AN INTRODUCTION TO PROGRAMMING AND COMPUTER SCIENCE WITH PYTHON

Second Edition

CLAYTON CAFIERO

An Introduction to Programming and Computer Science with Python

Second edition

Clayton Cafiero
The University of Vermont

This book is for use under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 License: https://creativecommons.org/licenses/by-nc-sa/4.0.

Book style has been adapted from the Memoir class for TEX, copyright © 2001–2011 Peter R. Wilson, 2011–2022 Lars Madsen, and is thus excluded from the above licence.

Images from Matplotlib.org in Chapter 15 are excluded from the license for this material. They are subject to Matplotlib's license at https://matplotlib.org/stable/users/project/license.html. Photo of Edsger Dijkstra by Hamilton Richards, University Texas at Austin, available under a Creative Commons CC BY-SA 3.0 license: https://creativecommons.org/licenses/by-sa/3.0/.

No generative AI was used in writing this book.

Manuscript prepared by the author with Quarto, Pandoc, Poetry, and XƎLATEX.

Illustrations, diagrams, and cover artwork by the author, except for the graph in Chapter 17, Exercise 2, which is adapted from an earlier version by Harry Sharman.

Version: 0.5.1

ISBN: 979-8-9887092-1-3
Library of Congress Control Number: 2025912312

Second edition
10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

For the Bug and the Bull

Table of contents

Table of contents i

Preface vii

Preface to the second edition ix

To the student xiii

Acknowledgements xv

1 Introduction 1

2 Programming and the Python Shell 11
- 2.1 Why learn a programming language? . . . 12
- 2.2 Compilation and interpretation . . . 14
- 2.3 The Python shell . . . 16
- 2.4 Hello, Python! . . . 18
- 2.5 Syntax and semantics . . . 19
- 2.6 Introduction to binary numbers . . . 21
- 2.7 Exercises . . . 25

3 Types and literals 27
- 3.1 What are *types*? . . . 28
- 3.2 Dynamic typing . . . 31
- 3.3 Types and memory . . . 33
- 3.4 More on string literals . . . 35
- 3.5 Representation error of numeric types . . . 37
- 3.6 Exercises . . . 41

4 Variables, statements, and expressions 43
- 4.1 Variables and assignment . . . 44
- 4.2 Expressions . . . 49
- 4.3 Augmented assignment operators . . . 54
- 4.4 Euclidean or “floor” division . . . 54
- 4.5 Modular arithmetic . . . 59
- 4.6 Exponentiation . . . 66
- 4.7 Exceptions . . . 68
- 4.8 Exercises . . . 71

5 Functions **75**
5.1 Introduction to functions 76
5.2 A deeper dive into functions 81
5.3 Passing arguments to a function 87
5.4 Scope 88
5.5 Pure and impure functions 89
5.6 The `math` module 91
5.7 Free variables, scope, and LEGB 94
5.8 Exercises 100

6 Style **103**
6.1 The importance of style 104
6.2 Constants 107
6.3 Comments in code 108
6.4 Dealing with long lines 111
6.5 Exercises 113

7 Console I/O **115**
7.1 Motivation 116
7.2 Command line interface 116
7.3 The `input()` function 116
7.4 Converting strings to numeric types 118
7.5 Some ways to format output 123
7.6 Python f-strings and string interpolation 124
7.7 Format specifiers 125
7.8 Scientific notation 126
7.9 Formatting tables 126
7.10 Example: currency converter 129
7.11 Format specifiers: a quick reference 132
7.12 Exceptions 133
7.13 Exercises 135

8 Branching, comparisons, and conditions **137**
8.1 Boolean logic and Boolean expressions 138
8.2 Comparison operators 141
8.3 Branching 143
8.4 `if`, `elif`, and `else` 144
8.5 Truthy and falsey 146
8.6 Identity and equivalence 147
8.7 Input validation 149
8.8 Some string methods 150
8.9 Decision trees 154
8.10 Exceptions 156
8.11 Exercises 157

9 Structure, development, and testing **161**
9.1 `main` the Python way 162
9.2 Program structure 168
9.3 Iterative and incremental development 168
9.4 Testing your code 174
9.5 The origin of the term “bug” 180
9.6 Using assertions to test your code 182

9.7 Rubberducking . . . 184
9.8 Exceptions . . . 185
9.9 Exercises . . . 185

10 Sequences 189
10.1 Lists . . . 190
10.2 Tuples . . . 197
10.3 Mutability and immutability . . . 201
10.4 Subscripts are indices . . . 204
10.5 Concatenating lists and tuples . . . 206
10.6 Copying lists . . . 206
10.7 Finding an element within a sequence . . . 207
10.8 Counting elements within a sequence . . . 209
10.9 Sequence unpacking . . . 212
10.10 Strings are sequences . . . 214
10.11 Some more string methods . . . 215
10.12 The Python built-in `sorted()` . . . 218
10.13 Sequences: a quick reference guide . . . 219
10.14 Slicing . . . 221
10.15 Passing mutables to functions . . . 223
10.16 The `global` keyword . . . 224
10.17 Exceptions . . . 228
10.18 Exercises . . . 229

11 Loops, iteration, and iterables 233
11.1 Loops: an introduction . . . 234
11.2 `while` loops . . . 235
11.3 Input validation with `while` loops . . . 240
11.4 An ancient algorithm with a `while` loop . . . 243
11.5 `for` loops . . . 245
11.6 Iterables . . . 250
11.7 Iterating over strings . . . 252
11.8 The list method `.extend()` . . . 252
11.9 Calculating a sum in a loop . . . 254
11.10 Loops and summations . . . 255
11.11 Products . . . 256
11.12 `enumerate()` . . . 256
11.13 Tracing a loop . . . 259
11.14 Nested loops . . . 263
11.15 Stacks and queues . . . 265
11.16 A deeper dive into iteration in Python . . . 268
11.17 Exercises . . . 270

12 Randomness, games, and simulations 275
12.1 The `random` module . . . 276
12.2 Pseudo-randomness in more detail . . . 282
12.3 Using the seed . . . 283
12.4 Exercises . . . 285

13 File I/O 289
13.1 Context managers . . . 290
13.2 Reading from a file . . . 290

13.3 Writing to a file . . . 291
13.4 Keyword arguments . . . 293
13.5 More on printing strings . . . 294
13.6 The `csv` module . . . 295
13.7 Exceptions . . . 299
13.8 Exercises . . . 300

14 Data analysis and presentation 303
14.1 Some elementary statistics . . . 303
14.2 Python's `statistics` module . . . 309
14.3 A brief introduction to plotting with Matplotlib . . . 310
14.4 The basics of Matplotlib . . . 312
14.5 Exceptions . . . 317
14.6 Exercises . . . 317

15 Exception handling 321
15.1 Exceptions . . . 322
15.2 Handling exceptions . . . 327
15.3 Exceptions and flow of control . . . 330
15.4 Exercises . . . 331

16 Dictionaries, sets, and structured data 333
16.1 Introduction to dictionaries and structured data . . . 334
16.2 Iterating over dictionaries . . . 339
16.3 Deleting dictionary keys . . . 341
16.4 Hashables . . . 342
16.5 Counting letters in a string . . . 344
16.6 Sets . . . 345
16.7 Named tuples . . . 351
16.8 Structured data with JSON . . . 356
16.9 Exceptions . . . 364
16.10 Exercises . . . 367

17 Graphs 371
17.1 Introduction to graphs . . . 371
17.2 Searching a graph: breadth-first search . . . 373
17.3 Exercises . . . 377

Appendices 379

A Glossary 379

B Mathematical notation 411

C `pip` and `venv` 413

D File systems 417

E Flow charts 421

F Code for cover artwork 427

G Code smells for beginners **431**

H The call stack **445**

I The joy of Unicode **457**

J A brief introduction to databases with SQLite **463**

Preface

This book has been written for use in University of Vermont's CS1210 Introduction to Programming. This is a semester long course which covers much of the basics of programming, and an introduction to some fundamental concepts in computer science. Not being happy with any of the available textbooks, I endeavored to write my own. Drafting began in August 2022, essentially writing a chapter a week over the course of the semester, delivered to students via UVM's learning management system. The text was revised, edited, and expanded in the following semester.

UVM's CS1210 carries "QR" (quantitative reasoning) and "QD" (quantitative and data literacy) designations. Accordingly, there's some mathematics included:

- writing functions to perform calculations,
- writing programs to generate interesting integer sequences,
- demonstrating the connection between pure functions and mathematical functions,
- demonstrating the connection between list indices and subscript notation,
- demonstrating that summations are loops,

and so on, to address the QR requirement. To address the QD requirement, we include some simple plotting with Matplotlib. Other aspects of these requirements are addressed in programming assignments, lab exercises, and lecture.

Nevertheless, despite this book's primary objective as instructional material for a specific course at UVM, others may find this material useful.

–CC, July 2023

Preface to the second edition

This book was always intended to remain a work-in-progress, and the second edition includes new topics which address frequent questions we've encountered in teaching CS 1210 Introduction to Programming at UVM:

- Chapter 5 (Functions) now includes discussion of free variables and Python's LEGB (local, enclosing, global, built-in) resolution strategy for finding matching identifiers, `UnboundLocalError`, and new exercises.
- Chapter 6 (Style) now includes strategies for dealing with long lines and PEP 8 conformance.
- Chapter 8 has been retitled "Branching, comparisons, and conditions," and now includes coverage of using the keyword `is` to test identity and `UnboundLocalError` which often occurs when we have poorly-written branching structures within functions.
- Chapter 10 (Sequences) includes new material on selected string methods, the sequence method `.count()`, the built-in `sorted()`, and use and misuse of the `global` keyword (introduced here because it is only when we get to this chapter that we enter into a discussion of mutability and immutability).
- Chapter 11 has been retitled "Loops, iteration, and iterables," and now covers the list method `.extend()` (since in this chapter we can speak of iterables).
- Chapter 12 (Randomness, games, and simulations) has been expanded to include `.sample()` and `.gauss()` on account of their abundant applications still accessible to beginners.
- Chapter 13 (File I/O) now includes coverage of `UnicodeDecodeError`.
- Chapter 15 (Exception handling) now includes coverage of `UnboundLocalError`, `UnicodeDecodeError`, and `JSONDecodeError`.
- Chapter 16 has been retitled "Dictionaries, sets, and structured data," and now includes sets, named tuples, and basic use of the JSON module.
- There are four new appendices:
  - Appendix G: Code smells for beginners,
  - Appendix H: The call stack,
  - Appendix I: The joy of Unicode,
  - Appendix J: An introduction to databases with SQLite.

Since UVM's CS2100 introduces object-oriented programming in Java, I've included, either in new chapter material or in appendices, transitional material like named tuples and `dataclass`. These are lightweight

alternatives to full-fledged OOP which also introduce the concept of defining a class.

There are other minor revisions and corrections, a few new examples, and some additional exercises.

I have been on the fence about `all()`, `any()`, `map()`, `zip()`, `filter()`, `reduce()`, and a few other built-in features. I've also considered more coverage of the `collections` module, and introducing portions of `argparse`, `datetime`, `functools`, `glob`, `itertools`, `os`, `pathlib`, `re`, `shutil`, `subprocess`, and `sys` modules (or some subset thereof). I've also haggled with myself over including doctests, `raise`, generators (with `yield`), decorators, lambdas, closures, and recursion. These are all widely used in everyday practice, but I've held off on these for the time being. Needless to say, what can be covered in an introductory text is, of necessity, only the tip of a very large iceberg. Nevertheless, if you have suggestions as to what else might be included in this book, I'd love to hear from you.

I don't expect everything here can be covered in a semester, but I think this will serve as a reference for students as they continue to learn Python. Also it will give instructors some flexibility if they wish to cover a few additional topics in more detail.

–CC, July 2025

Errata and suggestions

Once again, I'm aware that this text isn't quite "ready for prime time," but, as it's been said, "time and tide wait for no one," and a new semester approaches. So we push this unfinished work out of the nest, and hope for the best. If you have errata (which I'm sure are present) or suggestions, I'm all ears and I welcome your feedback—bouquets or brickbats or anything in between.

Contact

Clayton Cafiero
The University of Vermont
College of Engineering and Mathematical Sciences
Department of Computer Science
Innovation E309
82 University Place
Burlington, VT 05405-0125 (USA)

cbcafier@uvm.edu
https://www.uvm.edu/~cbcafier

To the student

Learning how to program is fun and rewarding, but it demands a rather different, structured approach to problem solving. This comes with time and practice. While I hope this book will help you learn how to solve problems with code, the only way to learn programming is by doing it. There's more than a little trial and error involved. If you find yourself struggling, don't despair—it just takes time and practice.

You will make mistakes—that's part of the process. As John Dewey once said, "Failure is instructive. The person who really thinks learns quite as much from their failures as from their successes."

You'll notice in this book that there are abundant examples given using the Python shell. The Python shell is a great way to experiment and deepen your understanding. I encourage you to follow along with the examples in the book, and enter them into the shell yourself. Unlike writing programs and then running them, interacting with the Python shell gives you immediate feedback. If you don't understand something as well as you'd like, turn to the shell. Experiment there, and then go back to writing your program.

If you take away only one thing from a first course in programming, it should not be the details of the syntax of the language. Rather, it should be an ability to *decompose* a problem into smaller units, to solve the smaller problems in code, and then to build up a complete solution from partial solutions of smaller problems. The primary vehicle for this approach is *functions*. So make sure you gain a firm grasp of functions (see in particular Chapter 5).

Some might say that in the age of AI a strong foundation in programming is no longer necessary. I'd argue quite the opposite. Now more than ever, students need a strong foundation in order to work effectively with new tools, judge the quality of model output, and to keep a firm hand on the rudder. It's also crucial that we continue to develop and exercise our critical thinking and problem solving skills. Without these, we are lost.

Good luck and happy coding!

Acknowledgements

Thanks to my colleagues, students, and teaching assistants in the Department of Computer Science at the University of Vermont for motivation, encouragement, suggestions, feedback, and corrections. Without you, this book would not exist. Thanks to Chris Skalka for support and encouragement, for the opportunity to teach, and for luring me back to UVM. Thanks to Isaac Levy for stimulating conversations on Python, and for feedback on early drafts that he used in his own teaching. Thanks to Jackie Horton, particularly for helpful comments on Chapter 3. Thanks to Jim Eddy for helping me through my first semester of teaching CS1210 (back when it was CS021). Thanks to Sami Connolly for using a preliminary draft in her own teaching and providing feedback. Thanks to Lisa Dion for morning check-ins and many helpful suggestions. Thanks to Joan "Rosi" Rosebush for regular chocolate deliveries. Thanks to Harry Sharman for helping with much of the painstaking work of turning words into a book, and for contributing a few of the exercises and an early version of Appendix D. Thanks to Deborah Cafiero for proofreading and patience. Thanks to Jim Hefferon who has served as a role model without knowing it.

Since the release of the first print edition, the following people have reported defects and provided corrections: Murat Güngör, Daniel Triplett, Anna Gale, Nina Holm, Colin Menuchi, Shiloh Chiu, Ted Pittman, Milan Chirag Shah, Andrew Slowman, Joey Donohue, Gale Stone, Darby Lane, Lillian Wyckoff, Angelique Macie, Alex Marshall, Ryan Stailey, Darrian Michaelides, Jackson Francis, Matthew Jackmore, Sami Connolly, Gabriella DiGiovanni, Luke Ste. Marie, Tyler Berman, and Prakash Pant. Thank you all.

Chapter 1

Introduction

> Computer science is a science of abstraction—creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.
>
> –Alfred V. Aho

> Abstraction is deliverance.
>
> –James Brandon Lewis

The goal of this book is to provide an introduction to computer programming with Python. This includes

- functional decomposition and a structured approach to programming,
- writing idiomatic Python,
- understanding the importance of abstraction,
- practical problem-solving exercises, and
- understanding and using structured data.

When you get to know it, Python is a peculiar programming language.[1] Much of what's peculiar about Python is concealed by its seemingly simple syntax. This is part of what makes Python a great first language—and it's fun!

Organization of this book

The book is organized into chapters which roughly correspond to a week's worth of material (with some deviations). Some chapters, particularly the first few, should be consumed at a rate of two a week. We present below a brief description of each chapter, followed by mention of some conventions used in the book.

[1] It's not quite *sui generis*—Python is firmly rooted in the tradition of ALGOL-influenced programming languages.

Programming and the Python shell

This chapter provides some motivation for why programming languages are useful, and gives a general outline of how a program is executed by the Python interpreter. This chapter also introduces the two modes of using Python. The *interactive mode* allows the user to interact with the Python interpreter using the Python shell. Python statements and expressions are entered one at a time, and the interpreter evaluates or executes the code entered by the user. This is an essential tool for experimentation and learning the details of various language features. *Script mode* allows the user to write, save, and execute Python programs. This is convenient since in this mode we can save our work, and run it multiple times without having to type it again and again at the Python shell.

This chapter also includes a brief introduction to binary numbers and binary arithmetic.

Types and literals

The concept of *type* is one of the most important in all computer science (indeed there's an entire field called "type theory").

This chapter introduces the most commonly used Python types, though in some cases, complete presentation of a given type will come later in the text—lists and dictionaries, for example. Other types are introduced later in the text (for example, `function`, `range`, `enumerate`, *etc.*). As types are introduced, examples of literals of each type are given.

Since representation error is a common cause of bewilderment among beginners, there is some discussion of why this occurs with `float` objects.

Variables, statements, and expressions

This chapter introduces much of the machinery that will be used throughout the remainder of the text: variables, assignment, expressions, operators, and evaluation of expressions.

On account of its broad applicability, a substantial account of modular arithmetic is presented as well.

Functions

Functions are the single most important concept a beginning programmer can acquire. *Functional decomposition* is a crucial requirement of writing reliable, robust, correct code.

This chapter explains why we use functions, how functions are defined, how functions are called, and how values are returned. We've tried to keep this "non-technical" and so there's no discussion of a call stack, though there is discussion of scope.

Because beginning programmers often introduce side effects into functions where they are undesirable or unnecessary, this chapter makes clear the distinction between *pure* functions (those without side effects) and *impure* functions (those with side effects, including mutating mutable objects).

Because the `math` module is so widely used and includes many useful functions, we introduce the `math` module in this chapter. In this way, we

also reinforce the idea of information hiding and good functional design. Do we need to know how the `math` module implements its `sqrt()` function? Of course not. Should we have to know how a function is implemented in order to use it? Apart from knowing what constitutes a valid input and what it returns as an output, no, we do not!

Style

Our goal here is to encourage the writing of idiomatic Python. Accordingly, we address the high points of PEP 8—the *de facto* style guide for Python—and provide examples of good and bad style.

Students don't always understand how important style is for the readability of one's code. By following style guidelines we can reduce the cognitive load that's required to read code, thereby making it easier to reason about and understand our code.

Console I/O (input/output)

This chapter demonstrates how to get input from the user (in command line programs) and how to format output using f-strings. Because f-strings have been around so long now, and because they allow for more readable code, we've avoided presentation of older, and now seldom used, C-style string formatting.

Branching, comparisons, and conditions

Branching is a programming language's way of handling *conditional execution* of code. In this chapter, we cover conditions (Boolean expressions) which evaluate to a true or false (or a value which is "truthy" or "falsey"—like true or like false). Python uses these conditions to determine whether a block of code should be executed. In many cases we have multiple branches—multiple paths of execution that might be taken. These are implemented with `if`, `elif` (a *portmanteau* of "else if"), and often `else`.

One common confusion that beginners face is understanding which branch is executed in an `if`/`elif`/`else` structure, and hopefully the chapter makes this clear.

Also covered are nested `if` statements, and two ways of visually representing branching (each appropriate to different use cases)—decision trees and flow charts.

Structure, development, and testing

Beginners often struggle with how to structure their code—both for proper flow of execution and for readability. This chapter gives clear guidelines for structuring code based on common idioms in Python. It also addresses how we can incrementally build and test our code.

Unlike many introductory texts, we present *assertions* in this chapter. Assertions are easy to understand and their use has great pedagogical value. In order to write an assertion, a programmer must understand clearly what behavior or output is expected for a given input. Using

assertions helps you reason about what should be happening when your code is executed.

Sequences

Sequences—lists, tuples, and strings—are presented in this chapter. It makes sense to present these before presenting loops for two reasons. First, sequences are *iterable*, and as such are used in `for` loops, and without a clear understanding of what constitutes an iterable, understanding such loops may present challenges. Second, we often do work within a loop which might involve constructing or filtering a list of objects.

Common features of sequences—for example, they are all indexed, support indexed reads, and are iterable—are highlighted throughout the chapter.

As this chapter introduces our first *mutable* type, the Python list, we present the concepts of mutability and immutability in this chapter.

Loops, iteration, and iterables

Loops allow for repetitive work or calculation. In this chapter we present the two kinds of loop supported by Python—`while` loops and `for` loops. At this point, students have seen iterables (in the form of sequences) and Boolean expressions, which are a necessary foundation for a proper presentation of loops.

Also, this chapter introduces two new types—`range` and `enumerate`—and their corresponding constructors. Presentation of `range` entails discussion of *arithmetic sequences*, and presentation of `enumerate` works nicely with tuple *unpacking* (or more generally, sequence unpacking), and so these are presented first in this chapter.

This chapter also provides a brief introduction to stacks and queues, which are trivially implemented in Python using `list` as an underlying data structure.

I've intentionally excluded treatment of *comprehensions* since beginners have difficulty reading and writing comprehensions without a prior, solid foundation in `for` loops.

Randomness, games, and simulations

There are many uses for randomness. Students love to write programs which implement games, and many games involve some chance element or elements—rolling dice, spinning a wheel, tossing a coin, shuffling a deck, and so on. Another application is in simulations, which may also include some chance elements. All manner of physical and other phenomena can be simulated with some randomness built in.

This chapter presents Python's `random` module, and some of the more commonly used methods within this module—`random.random()`, `random.randint()`, `random.choice()`, and `random.shuffle()`. Much of this is done within the context of games of chance, but we also include some simulations (for example, random walk and Gambler's Ruin). There is also some discussion of pseudo-random numbers and how Python's pseudo-random number generator is seeded.

File I/O (input/output)

This chapter shows you how to read data from and write data to a file. File I/O is best accomplished using a context manager. Context managers were introduced with Python 2.5 in 2006, and are a much preferred idiom (as compared to using try/finally). Accordingly, all file I/O demonstrations make use of context managers created with the Python keyword `with`.

Because so much data is in CSV format (or can be exported to this format easily), we introduce the `csv` module in this chapter. Using the `csv` module reduces some of the complexity we face when reading data from a file, since we don't have to parse it ourselves.

Data analysis and presentation

This chapter is motivated in large part by the University of Vermont's QD (quantitative and data literacy) designation for the course for which this textbook was written. Accordingly, we present some very basic descriptive statistics and introduce Python's `statistics` module including `statistics.mean()`, `statistics.pstdev()`, and `statistics.quantiles()`. The presentation component of this chapter is done using Matplotlib, which is the *de facto* standard for plotting and visualization with Python. This covers only the rudiments of Matplotlib's Pyplot interface (line plot, bar plot, *etc.*), and is not intended as a complete introduction.

Exception handling

In this chapter, we present simple exception handling (using try/except, but not `finally`), and explain that some exceptions should not be handled since in doing so, we can hide programming defects which should be corrected. We also demonstrate the use of exception handling in input validation. When you reach this chapter, you'll already have seen `while` loops for input validation, so the addition of exception handling represents only an incremental increase in complexity in this context.

Dictionaries, sets, and structured data

Dictionaries are the last new type we present in the text. Dictionaries store information using a key/value model—we look up values in a dictionary by their keys. Like sequences, dictionaries are iterable, but since they have keys rather than indices, this works a little differently. We'll see three different ways to iterate over a dictionary.

We'll also learn about *hashability* in the context of dictionary keys.

Graphs

Since graphs are so commonplace in computer science, it seems appropriate to include a basic introduction to graphs in this text. Plus, graphs are really fun!

A *graph* is a collection of *vertices* (also called *nodes*) and *edges*, which connect the vertices of the graph. The concrete example of a highway map is used, and an algorithm for breadth-first search (BFS) is demonstrated.

Since queues were introduced in chapter 11, the conceptual leap here—using a queue in the BFS algorithm—shouldn't be too great.

Assumptions regarding prior knowledge of mathematics

This text assumes a reasonable background in high-school algebra and a little geometry (for example, the Pythagorean theorem and right triangles). Prior exposure to summations and subscripts would help the reader but is not essential, as these are introduced in the text. The same goes for mean, standard deviation, and quantiles. You might find it helpful if you've seen these before, but these, too, are introduced in the text.

The minimum expectation is that you can add, subtract, multiply and divide; that you understand exponents and square roots; and that you understand the precedence of operations, grouping of expressions with parentheses, and evaluating expressions with multiple terms and operations.

Assumptions regarding prior knowledge of computer use

While this book assumes no prior knowledge whatsoever when it comes to programming, it does assume that you have some familiarity with using a computer and have a basic understanding of your computer's file system (a hierarchical system consisting of files and directories). If you don't know what a file is, or what a directory is, see Appendix D, or consult documentation for your operating system. Writing and running programs requires that you understand the basics of a computer file system.

Typographic conventions used in this book

Names of functions, variables, and modules are rendered in `fixed-pitch typeface`, as are Python keywords, code snippets, and sample output.

```
print("Hello, World!")
```

When referring to structures which make use of multiple keywords we render these keywords separated by slashes but do not use fixed-pitch typeface. Examples: `if/else`, `if/elif`, `if/elif/else`, try/except, try/finally, *etc.*

File names, for example, `hello_world.py`, and module names, for example, `math`, are also rendered in fixed-pitch typeface.

Where it is understood that code is entered into the Python shell, the interactive Python prompt `>>>` is shown. Wherever you see this, you should understand we're working in Python shell. `>>>` should never appear in your code.[2] Return values and evaluation of expressions are indicated just as they are in the Python shell, without the leading `>>>`.

[2]Except in the case of doctests, which are not presented in this text.

```
>>> 1 + 2
3
>>> import math
>>> math.sqrt(36)
6
```

In a few places, items which are placeholders for actual values or variable names are given in angle brackets, thus `<foo>`. For example, when describing the three-argument syntax for the `range()` function, we might write `range(<start>, <stop>, <stride>)` to indicate that three arguments must be supplied—the first for the start value, the second for the stop value, and the last for the stride. It's important to understand that the angle brackets are not part of the syntax, but are merely a typographic convention to indicate where an appropriate substitution must be made.

All of these conventions are in accord with the typographical conventions used in the official Python documentation at python.org. Hopefully, this will make it easier for students when they consult the official documentation.

Note that this use of angle brackets is a little different when it comes to traceback messages printed when exceptions occur. There you may see things like `<stdin>` and `<module>`, and in this context, they are not placeholders requiring substitution by the user.

Other conventions

When referring to functions, whether built-in, from some imported module, or otherwise, without any other context or specific problem instance, we write function identifiers along with parentheses (as a visual indicator that we're speaking of a function) but without formal parameters. Example: "The `range()` function accepts one, two, or three arguments." This should not be read as suggesting that `range()` takes no arguments.

Entry point / top-level code environment

As noted in the text, unlike many other languages such as C, C++, Java, *etc.*, a function named `main()` has *no special meaning in Python whatsoever*. The correct way to specify the entry point of your code in Python is with

```
if __name__ == '__main__':
    # the rest of your code here
```

This is explained fully in Chapter 9.

In code samples in the book, we do, however, avoid using this *if there are no function definitions included in the code*. We do this for space and conciseness of the examples. The same could reasonably apply to your code. In most cases, if there are no function definitions in your module, there's no need for this `if` statement (though it's fine to include it). However, if there are any function definitions in your module, then `if __name__ == '__main__':` is the correct, Pythonic way to segregate your driver code from your function definitions.

Origin of Python

Python has been around a long time, with the first release appearing in 1991 (four years before Java). It was invented by Guido van Rossum, who is now officially Python's benevolent dictator for life (BDFL).

Figure 1.1: Guido van Rossum, BDFL

Python gets its name from the British comedy troupe *Monty Python's Flying Circus* (Guido is a fan).

Nowadays, Python is one of the most widely used programming languages on the planet and is supported by an immense ecosystem and thriving community. See: https://python.org/ for more.

Python version

As this book is written (second edition), the current version of Python is 3.13. However, no new language features introduced since version 3.9 are presented in this book (as most are not appropriate or even useful for beginners). This book does cover f-strings, which were introduced in version 3.6. Accordingly, if you have Python version 3.9 or higher, you should be able to follow along with all code samples and exercises.

Using the Python documentation

For beginners and experts alike, a language's documentation is an essential resource. Accordingly, it's important that you know how to find and consult Python's online documentation.

There are many resources available on the internet and the quality of these resources varies from truly awful to superb. The online Python documentation falls toward the good end of that spectrum.

Pros

- Definitive and up-to-date
- Documentation for different versions clearly specified
- Thorough and accurate
- Includes references for all standard libraries
- Available in multiple languages
- Includes a comprehensive tutorial for Python beginners
- Coding examples (where given) conform to good Python style (PEP 8)

Cons

- Can be lengthy or technical—not always ideal for beginners
- Don't always appear at top of search engine results.

python.org

Official Python documentation, tutorials, and other resources are hosted at https://python.org.

- Documentation: https://docs.python.org/3/
- Tutorial: https://docs.python.org/3/tutorial/
- Beginner's Guide: https://wiki.python.org/moin/BeginnersGuide

I recommend these as the first place to check for online resources.

> ⚠ Warning
>
> There's a lot of incorrect, dated, or otherwise questionable code on the internet. Be careful when consulting sources other than official documentation.

Chapter 2

Programming and the Python Shell

Our objective for this chapter is to lay the foundations for the rest of the course. If you've done any programming before, some of this may seem familiar, but read carefully nonetheless. If you haven't done any programming before that's OK.

Learning objectives

- You will learn how to interact with the Python interpreter using the *Python shell*.
- You will learn the difference between *interactive mode* (in the shell) and *script mode* (writing, saving, and running programs).
- You will learn a little about computers, how they are structured, and that they use binary code.
- You will understand why we wish to write code in something other than just zeros and ones, and you'll learn a little about how Python translates high-level code (written by you, the programmer) into binary instructions that a computer can execute.
- You will write, save, and run your first Python program—an ordered collection of statements and expressions.

Terms introduced

- binary code
- bytecode
- compilation *vs* interpretation
- compiler
- console
- integrated development environment (IDE)
- interactive mode
- low-level *vs* high-level programming language
- Python interpreter / shell
- read-evaluate-print loop (REPL)
- semantics
- script mode

- syntax
- terminal

2.1 Why learn a programming language?

Computers are powerful tools. Computers can perform all manner of tasks: communication, computation, managing and manipulating data, modeling natural phenomena, and creating images, videos, and music, just to name a few. However, computers don't read minds (yet), and thus we have to provide instructions to computers so they can perform these tasks.

Computers don't speak natural languages (yet)—they only understand binary code. Binary code is unreadable by humans.

For example, a portion of an executable program might look like this (in binary):

```
0110101101101011 1100000000110101 1011110100100100
1010010100100100 0010100100010011 1110100100010101
1110100100010101 0001110110000000 1110000111100000
0000100000000001 0100101101110100 0000001000101011
0010100101110000 0101001001001001 1010100110101000
```

This is unintelligible. It's bad enough to try to read it, and it would be even worse if we had to write our computer programs in this fashion.

Computers don't speak human language, and humans don't speak computer language. That's a problem. The solution is *programming languages*.

Programming languages allow us, as humans, to write instructions in a form we can understand and reason about, and then have these instructions converted into a form that a computer can read and execute.

There is a tremendous variety of programming languages. Some languages are *low-level*, like assembly language, where there's roughly a one-to-one correspondence between machine instructions and assembly language instructions. Here's a "Hello World!" program in assembly language (for ARM64 architecture):[1]

```
.equ STDOUT, 1
.equ SVC_WRITE, 64
.equ SVC_EXIT, 93

.text
.global _start

_start:
    stp x29, x30, [sp, -16]!
    mov x0, #STDOUT
    ldr x1, =msg
    mov x2, 13
```

[1] Assembly language code sample from Rosetta Code: https://www.rosettacode.org/wiki/Hello_world

```
    mov x8, #SVC_WRITE
    mov x29, sp
    svc #0 // write(stdout, msg, 13);
    ldp x29, x30, [sp], 16
    mov x0, #0
    mov x8, #SVC_EXIT
    svc #0 // exit(0);

msg:    .ascii "Hello World!\n"
.align 4
```

Now, while this is a lot better than a string of zeros and ones, it's not so easy to read, write, and reason about code in assembly language.

Fortunately, we have *high-level* languages. Here's the same program in C++:

```
#include <iostream>

int main () {
  std::cout << "Hello World!" << std::endl;
}
```

Much better, right?

In Python, the same program is even more succinct:

```
print('Hello World!')
```

Notice that as we progress from machine code to Python, we're increasing abstraction. Machine code is the least abstract. These are the actual instructions executed on your computer. Assembly code uses human-readable symbols, but still retains (for the most part) a one-to-one correspondence between assembly instructions and machine instructions. In the case of C++, we're using a library `iostream` to provide us with an abstraction of an output stream, `std::cout`, and we're just sending strings to that stream. In the case of Python, we simply say "print this string" (more or less). This is the most abstract of these examples—we needn't concern ourselves with low-level details.

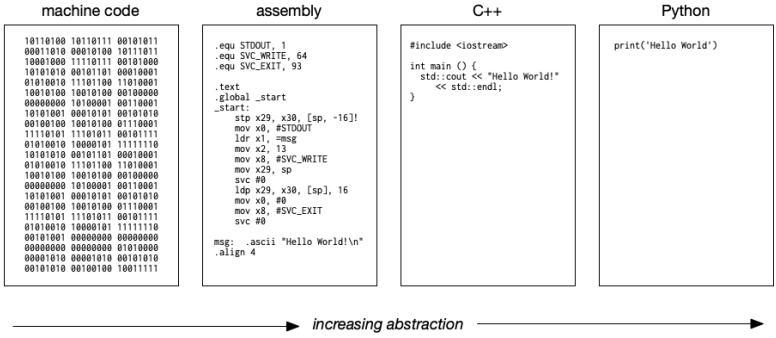


Figure 2.1: Increasing abstraction

Now, you may be wondering: How is it that we can write programs in such languages when computers only understand zeros and ones? There are programs which convert high-level code into machine code for execution. There are two main approaches when dealing with high-level languages, *compilation* and *interpretation*.

2.2 Compilation and interpretation

Generally speaking, *compilation* is a process whereby source code in some programming language is converted into binary code for execution on a particular architecture. The program which performs this conversion is called a *compiler*. The compiler takes source code (in some programming language) as an input, and yields binary machine code as an output.

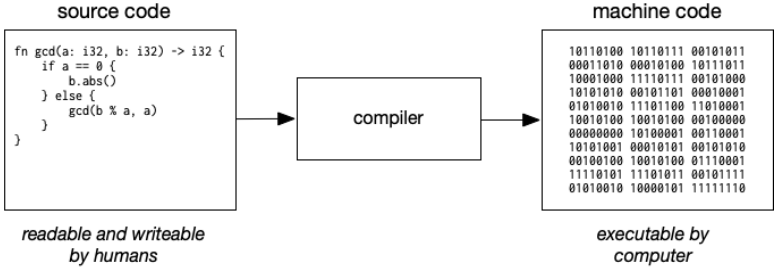


Figure 2.2: Compilation (simplified)

Interpreted languages work a little differently. Python is an interpreted language. In the case of Python, intermediate code is generated, and then this intermediate code is read and executed by another program. The intermediate code is called *bytecode*.

While the difference between compilation and interpretation is not quite as clear-cut as suggested here, these descriptions will serve for the present purposes.

The Python interpreter

Python is an interpreted language with intermediate bytecode. While you don't need to understand all the details of this process, it's helpful to have a general idea of what's going on.

Say you have written this program and saved it as `hello_world.py`.

```
print('Hello World!')
```

You may run this program from the terminal (command prompt), thus:

```
$ python hello_world.py
```

where `$` indicates a command prompt (your prompt may vary). When this runs, the following is printed to the console:

```
Hello World!
```

When we run this program, Python first reads the source code, then produces the intermediate bytecode, then executes each instruction in the bytecode.

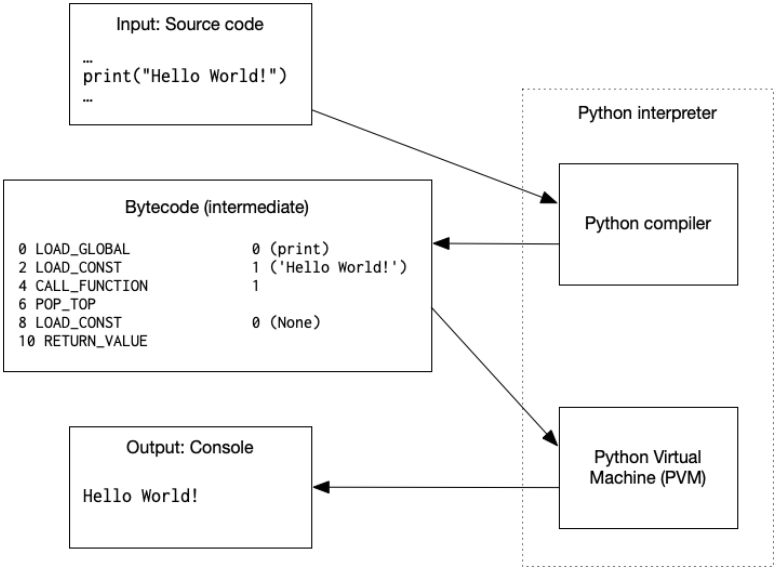


Figure 2.3: Execution of a Python program

1. By issuing the command `python hello_world.py`, we invoke the Python interpreter and tell it to read and execute the program `hello_world.py` (`.py` is the file extension used for Python files).
2. The Python interpreter reads the file `hello_world.py`.

3. The Python interpreter produces an intermediate, bytecode representation of the program in `hello_world.py`.
4. The bytecode is executed by the Python Virtual Machine.
5. This results in the words "Hello World!" being printed to the console.

So you see, there's a lot going on behind the scenes when we run a Python program.[2] However, this allows us to write programs in a high-level language that we as humans can understand.

Supplemental reading

- *Whetting Your Appetite*, from The (Official) Python Tutorial.[3]

2.3 The Python shell

The Python interpreter provides you with an environment for experimentation and observation—the *Python shell*, where we work in *interactive mode*. It's a great way to get your feet wet.

When working with the Python shell, you can enter *expressions* and Python will *read* them, *evaluate* them, and *print* the result. (There's more you can do, but this is a start.)

There are several ways to run the Python shell: in a terminal (command prompt) by typing `python`, `python3`, or `py` depending on the version(s) of Python installed on your machine. You can also run the Python shell through your chosen IDE (details will vary).

```
terminal
~
$ python
Python 3.10.11 …
Type "help", "copyright", "credits" or "license" for
more information.
>>> 'Hello, World!'
'Hello, World!'
>>>
```

Figure 2.4: The Python shell in a terminal (above)

[2] Actually, there's quite a bit more going on behind the scenes, but this should suffice for our purposes. If you're curious and wish to learn more, ask!

[3] https://docs.python.org/release/3.10.4/tutorial/appetite.html

The first thing you'll notice is this symbol: `>>>`. This is the Python prompt (you don't type this bit, this is Python telling you it's ready for new input).

We'll start with some simple examples (open the shell on your computer and follow along):

```
>>> 1
1
```

Here we've entered `1`. This is interpreted by Python as an integer, and Python responds by printing the *evaluation* of what you've just typed: `1`.

When we enter numbers like this, we call them "integer literals"—in the example above, what we entered was *literally* a `1`. Literals are special in that *they evaluate to themselves*.

Now let's try a simple *expression* that's not a mere literal:

```
>>> 1 + 2
3
```

Python understands arithmetic and when the operands are numbers (integers or floating-point) then `+` works just like you'd expect. So here we have a simple *expression*—a syntactically valid sequence of symbols that evaluates to a *value*. What does this expression evaluate to? `3` of course!

We refer to the `+` operator as a *binary infix operator*, since it takes two operands (hence, "binary") and the operand is placed *between* the operands (hence, "infix").

Here's another familiar binary infix operator: `-`. You already know what this does.

```
>>> 17 - 5
12
```

Yup. Just as you'd expect. The Python shell evaluates the expression `17 - 5` and returns the result: `12`.

REPL

This process—of entering an expression and having Python evaluate it and display the result—is called *REPL* which is an acronym for *read-evaluate-print loop*. Many languages have REPLs, and obviously, Python does too. REPLs were invented (back in the early 1960s) to provide an environment for *exploratory programming*. This is facilitated by allowing the programmer to see the result of each portion of code they enter. Accordingly, I encourage you to experiment with the Python shell. Do some tinkering and see the results. You can learn *a lot* by working this way.

Saving your work

Entering expressions into the Python shell *does not save anything.* In order to save your code, you'll want to work outside the shell (we'll see more on this soon).

Exiting the interpreter

If you're using an IDE there's no need to exit the shell. However, if you're using a terminal, and you wish to return to your command prompt, you may exit the shell with `exit()`.

```
>>> exit()
```

2.4 Hello, Python!

It is customary—a nearly universal ritual, in fact—when learning a new programming language, to write a program that prints "Hello World!" to the console. This tradition goes back as at least as far as 1974, when Brian Kernighan included such a program in his tutorial for the C programming language at Bell Labs, perhaps earlier.

So, in keeping with this fine tradition, our first program will do the same—print "Hello World!" to the console.

Python provides us with simple means to print to the console: a function named `print()`. If we wish to print something to the console, we write `print()` and place what we wish to print within the parentheses.

```
print("Hello World!")
```

That's it!

If we want to run a program in script mode we must write it and save it. Let's do that.

In your editor or IDE open a new file, and enter this one line of code (above). Save the file as `hello_world.py`.

Now you can run your program. If you're using an IDE, you can run the file within your IDE. You can also run the file from the command line, for example,

```
$ python hello_world.py
```

where `$` is the command line prompt (this will vary from system to system). The `$` isn't something you type, it's just meant to indicate a command prompt (like `>>>` in the Python shell). When you run this program it should print:

```
Hello World!
```

Next steps

The basic steps above will be similar for each new program you write. Of course, as we progress, programs will become more challenging, and it's likely you may need to test a program by running it multiple times as you make changes before you get it right. That's to be expected. But now you've learned the basic steps to create a new file, write some Python code, and run your program.

2.5 Syntax and semantics

In this text we'll talk about *syntax* and *semantics*, so it's important that we understand what these terms mean, particularly in the context of computer programming.

Syntax

In a (natural) language course—say Spanish, Chinese, or Latin—you'd learn about certain rules of *syntax*, that is, how we arrange words and choose the correct forms of words to produce a valid sentence or utterance. For example, in English,

> My hovercraft is full of eels.

is a syntactically valid sentence.[4] While it may or may not be true, and may not even make sense, it is certainly a *well-formed* English sentence. By contrast, the sequence of words

> Is by is is and cheese for

is *not* a well-formed English sentence. These are examples of valid and invalid syntax. The first is syntactically valid (*well-formed*); the second is not.

Every programming language has rules of syntax—rules which govern what is and is not a valid statement or expression in the language. For example, in Python

```
>>> 2 3
```

is not syntactically valid. If we were to try this using the Python shell, the Python interpreter would complain.

```
>>> 2 3
File "<stdin>", line 1
    2 3
    ^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

That's a clear-cut example of a syntax error in Python. Here's another:

[4] "My hovercraft is full of eels" originates in a famous sketch by Monty Python's Flying Circus.

```
>>> = 5
File "<stdin>", line 1
    = 5
    ^
SyntaxError: invalid syntax
```

Python makes it clear when we have syntax errors in our code. Usually it can point to the exact position within a line where such an error occurs. Sometimes, it can even provide suggestions, for example, "Perhaps you forgot a comma?"

Semantics

On the other hand, semantics is about *meaning*. In English we may say

> The ball is red.

We know there's some object being referred to—a ball—and that an assertion is being made about the color of the ball—red. This is fairly straightforward.

Of course, it's possible to construct ambiguous sentences in English. For example (with apologies to any vegetarians who may be reading):

> The turkey is ready to eat.

Does this mean that someone has cooked a turkey and that it is ready to be eaten? Or does this mean that there's a hungry turkey who is ready to be fed? This kind of ambiguity is quite common in natural languages. Not so with programming languages. If we've produced a syntactically valid statement or expression, it has only one "interpretation." There is no ambiguity in programming.

Here's another famous example, devised by the linguist Noam Chomsky:[5]

> Colorless green ideas sleep furiously.

This is a perfectly valid English sentence with respect to syntax. However, it is meaningless, nonsensical. How can anything be colorless and green at the same time? How can something abstract like an idea have color? What does it mean to "sleep furiously"? Syntax: A-OK. Semantics: nonsense.

Again, in programming, every syntactically valid statement or expression has a meaning. It is our job as programmers to write code which is syntactically valid but also semantically correct.

What happens if we write something which is syntactically valid and also semantically incorrect? It means that we've written code that *does not do what we intend for it to do*. There's a word for that: a *bug*.

Here's an example. Let's say we know the temperature in degrees Fahrenheit, but we want to know the equivalent in degrees Celsius. You may know the formula

[5] https://en.wikipedia.org/wiki/Noam_Chomsky

$$C = \frac{F - 32}{1.8}$$

where F is degrees Fahrenheit and C is degrees Celsius.

Let's say we wrote this Python code.

```
f = 68.0              # 68 degrees Fahrenheit
c = (f - 32) * 1.8    # attempt conversion to Celsius
print(c)              # print the result
```

This prints 64.8 which is incorrect! What's wrong? We're multiplying by 1.8 when we should be dividing by 1.8! This is a problem of *semantics*. Our code is syntactically valid. Python interprets it, runs it, and produces a result—but the result is *wrong*. Our code does not do what we intend for it to do. Call it what you will—a defect, an error, a bug—but it's a semantic error, not a syntactic error.

To fix it, we must change the *semantics*—the meaning—of our code. In this case the fix is simple.

```
f = 68.0              # 68 degrees Fahrenheit
c = (f - 32) / 1.8    # correct conversion to Celsius
print(c)              # print the result
```

and now this prints `20.0` which is correct. Now our program has the semantics we intend for it.

2.6 Introduction to binary numbers

You may know that computers use binary code to represent, well … *everything*. Everything stored on your computer's disk or solid-state drive is stored in binary form, a sequence of zeros and ones. All the programs your computer runs are sequences of zeros and ones. All the photos you save, all the music you listen to, even your word processing documents are all zeros and ones. Colors are represented with binary numbers. Audio waveforms are represented with binary numbers. Characters in your word processing document are represented with binary numbers. All the instructions executed and data processed by your computer are represented in binary form.

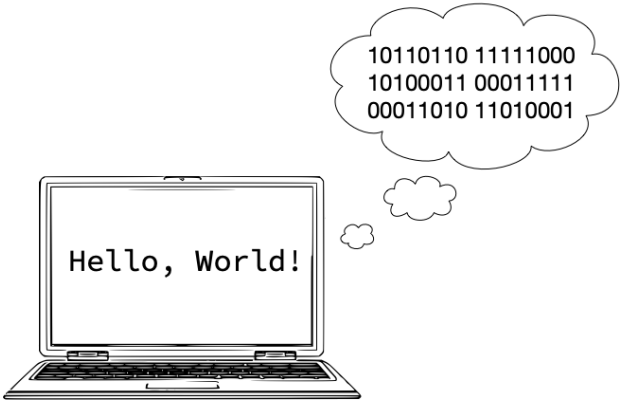


Figure 2.5

Accordingly, as computer scientists, we need to understand how we represent numbers in binary form and how we can perform arithmetic operations on such numbers.

However, first, let's review the familiar *decimal system*.

The decimal system

We've all used the decimal system.

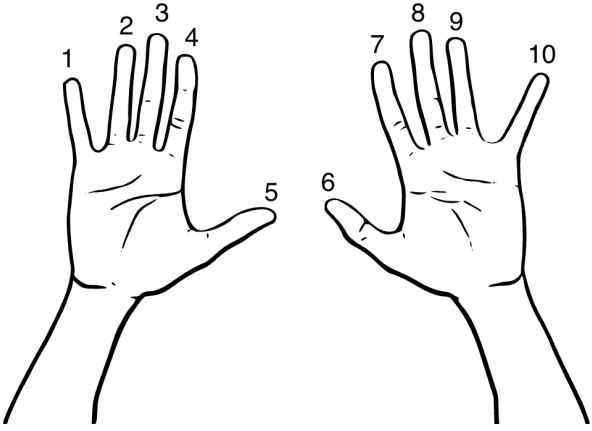


Figure 2.6

The decimal system is a *positional numeral system* based on powers of ten.[6] What do we mean by that? In the decimal system, we represent

[6]In fact, the first positional numeral system, developed in ancient Babylonia around 2000 BCE, used 60 as a base. Our base 10 system is an extension of the Hindu-Arabic numeral system. Other cultures have used other bases. For example,

numbers as coefficients in a sequence of powers of ten, where each coefficient appears in a position which corresponds to a certain power of ten. (That's a mouthful, I know.) This is best explained with an example.

Take the (decimal) number 8,675,309. Each digit is a coefficient in the sequence

$$8 \times 10^6 + 6 \times 10^5 + 7 \times 10^4 + 5 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 9 \times 10^0$$

Recall that anything to the zero power is one—so, $10^0 = 1$. If we do the arithmetic we get the correct result:

$$\begin{aligned}
8 \times 10^6 &= 8{,}000{,}000 \\
6 \times 10^5 &= \phantom{8{,}}600{,}000 \\
7 \times 10^4 &= \phantom{8{,}6}70{,}000 \\
5 \times 10^3 &= \phantom{8{,}675}5{,}000 \\
3 \times 10^2 &= \phantom{8{,}675{,}0}300 \\
0 \times 10^1 &= \phantom{8{,}675{,}00}00 \\
9 \times 10^0 &= \phantom{8{,}675{,}300}9
\end{aligned}$$

and all that adds up to 8,675,309.

This demonstrates the power and conciseness of a positional numeral system.

Notice that if we use base 10 for our system we need ten numerals to use as coefficients. For base 10, we use the numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

However, apart from the fact that most of us conveniently have ten fingers to count on, the choice of 10 as a base is arbitrary.

Computers and the binary system

As noted, computers use the binary system. This choice was originally motivated by the fact that electronic components which can be in one of two states are generally easier to design and implement than components that can be in one of more than two states.

So how does the binary system work? It, too, is a *positional numeral system*, but instead of using 10 as a base we use 2.

When using base 2, we need only two numerals: 0 and 1.

In the binary system, we represent numbers as coefficients in a sequence of powers of *two*. As with the decimal system, this is best explained with an example.

Take the decimal number 975. In binary this is 1111001111. That's

$$\begin{aligned}
&1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 \\
&\qquad + \ 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0
\end{aligned}$$

Again, doing the arithmetic

the Kewa counting system in Papua New Guinea is base 37—counting on fingers and other parts of the body: heel of thumb, palm, wrist, forearm, *etc*, up to the top of the head, and then back down the other side! See: Wolfers, E. P. (1971). "The Original Counting Systems of Papua and New Guinea", *The Arithmetic Teacher*, 18(2), 77-83, https://www.jstor.org/stable/41187615.

$$
\begin{aligned}
1 \times 2^9 &= 1000000000 \\
1 \times 2^8 &= 100000000 \\
1 \times 2^7 &= 10000000 \\
1 \times 2^6 &= 1000000 \\
0 \times 2^5 &= 000000 \\
0 \times 2^4 &= 00000 \\
1 \times 2^3 &= 1000 \\
1 \times 2^2 &= 100 \\
1 \times 2^1 &= 10 \\
1 \times 2^0 &= 1
\end{aligned}
$$

and that all adds up to 1111001111. To verify, let's represent these values in decimal format and check our arithmetic.

$$
\begin{aligned}
1 \times 2^9 &= 512 \\
1 \times 2^8 &= 256 \\
1 \times 2^7 &= 128 \\
1 \times 2^6 &= 64 \\
0 \times 2^5 &= 0 \\
0 \times 2^4 &= 0 \\
1 \times 2^3 &= 8 \\
1 \times 2^2 &= 4 \\
1 \times 2^1 &= 2 \\
1 \times 2^0 &= 1
\end{aligned}
$$

Indeed, this adds to 975.

Where in the decimal system we have the ones place, the tens place, the hundreds place, and so on, in the binary system we have the ones place, the twos place, the fours place, and so on.

How would we write, in binary, the decimal number 3? 11. That's one two, and one one.

How about the decimal number 10? 1010. That's one eight, zero fours, one two, and zero ones.

How about the decimal number 13? 1101. That's one eight, one four, zero twos, and one one.

Binary arithmetic

Once you get the hang of it, binary arithmetic is straightforward. Here's the most basic example: adding 1 and 1.

$$
\begin{array}{rr}
 & 1 \\
+ & 1 \\
\hline
 & 1\ 0
\end{array}
$$

In the ones column we add one plus one, that's two—binary 10—so we write 0, carry 1 into the twos column, and then write 1 in the twos column, and we're done.

Now let's add 1011 (decimal 11) and 11 (decimal 3).

$$\begin{array}{r} 1\ 0\ 1\ 1 \\ +\quad\ \ 1\ 1 \\ \hline 1\ 1\ 1\ 0 \end{array}$$

In the ones column we add one plus one, that's two—binary 10—so we write 0 and carry 1 into the twos column. Then in the twos column we add one (carried) plus one, plus one, that's three—binary 11—so we write 1 and carry 1 into the fours column. In the fours column we add one (carried) plus zero, so we write 1, and we have nothing to carry. In the eights column we have only the single eight, so we write that, and we're done. To verify (in decimal):

$$\begin{aligned} 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 &= 1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 14 \end{aligned}$$

That checks out.

2.7 Exercises

Exercise 01

Write a line of Python code that prints your name to the console.

Exercise 02

Multiple choice: Python is a(n) _____ programming language.

a. compiled
b. assembly
c. interpreted
d. binary

Exercise 03

True or false? Code that you write in the Python shell is saved.

Exercise 04

How do you exit the Python shell?

Exercise 05

Python can operate in two different modes. What are these modes and how do they differ?

Exercise 06

The following is an example of what kind of code?

```
1001011011011011 1110010110110001 1010101010101111
1111000011110010 0000101101101011 0110111000110110
```

Exercise 07

Calculate the following sums in binary:

a. 10 + 1
b. 100 + 11
c. 11 + 11
d. 1011 + 10

After you've worked these out in binary, convert to decimal form and check your arithmetic.

Exercise 08 (challenge!)

Try binary subtraction. What is 11011 - 1110? After calculating in binary, convert to decimal and check your answer.

Chapter 3

Types and literals

This chapter will expand our understanding of programming by introducing types and literals. All objects in Python have a *type*, and *literals* are fixed values of a given type. For example, the literal `1` is an integer and is of type `int` (short for "integer"). Python has many different types.

Learning objectives

- You will learn about many commonly used types in Python.
- You will understand why we have different types.
- You will be able to write literals of various types.
- You will learn different ways to write string literals which include various quotation marks within them.
- You will learn about representation error as it applies to numeric types (especially floating-point values).

Terms introduced

- dynamic typing
- escape sequence
- empty string, empty tuple, and empty list
- heterogeneous
- literal
- representation error
- static typing
- "strong" vs "weak" typing
- type (including `int`, `float`, `str`, `list`, `tuple`, `dict`, *etc.*)
- type inference
- Unicode

3.1 What are *types*?

Consider the universe of wheeled motor vehicles. There are many types: motorcycles, mopeds, automobiles, sport utility vehicles, busses, vans, tractor-trailers, pickup trucks, all-terrain vehicles, *etc.*, and agricultural vehicles such as tractors, harvesters, *etc.* Each type has characteristics which distinguish it from other types. Each type is suited for a particular purpose (you wouldn't use a moped to do the work of a tractor, would you?).

Similarly, everything in Python has a *type*, and every type is suited for a particular purpose. Python's types include numeric types such as integers and floating-point numbers; sequences such as strings, lists, and tuples; Booleans (true and false); and other types such as sets, dictionaries, ranges, and functions.

Why do we have different types in a programming language? Primarily for three reasons.

First, different types have different requirements regarding how they are stored in the computer's memory (we'll take a peek into this when we discuss *representation*).

Second, certain operations may or may not be appropriate for different types. For example, we can't raise 5 to the `'pumpkin'` power, or divide `'illuminate'` by 2.

Third, some operators behave differently depending on the types of their operands. For example, we'll see in the next chapter how `+` is used to add numeric types, but when the operands are strings `+` performs concatenation. How does Python know what operations to perform and what operations are permitted? It checks the type of the operands.

What's a *literal*?

A **literal** is simply fixed values of a given type. For example, `1` is a literal. It means, literally, the integer 1. Other examples of literals follow.

Some commonly used types in Python

Here are examples of some types we'll see. Don't worry if you don't know what they all are now—all will become clear in time.

Type	Description	Example(s) of literals
`int`	integer	`42`, `0`, `-1`
`float`	floating-point number	`3.14159`, `2.7182`, `0.0`
`str`	string	`'Python'`, `'badger'`, `'hovercraft'`
`bool`	Boolean	`True`, `False`
`NoneType`	none, no value	`None`
`tuple`	tuple	`()`, `('a', 'b', 'c')`, `(-1, 1)`

Type	Description	Example(s) of literals
`list`	list	`[]`, `[1, 2, 3]`, `['foo', 'bar', 'baz']`
`dict`	dictionary (key: value)	`{'cheese': 'stilton'}`, `{'age': 99}`
`function`	function	(see: Chapter 5)

`int`

The `int` type represents *integers*, that is, whole numbers, positive or negative, and zero. Examples of `int` literals: `1`, `42`, `-99`, `0`, `10000000`, *etc.* For readability, we can write integer literals with underscores in place of thousands separators. For example, `1_000_000` is rather easier to read than `1000000`, and both have the same value.

`float`

Objects of the `float` type represent floating-point numbers, that is, numbers with decimal (radix) points. These approximate real numbers (to varying degrees; see the section on representation error). Examples of `float` literals: `1.0`, `3.1415`, `-25.1`, *etc.*

`str`

A *string* is an ordered sequence of characters. Each word on this page is a string. So are `"abc123"` and `"@&)z)$"`—the symbols of a string needn't be alphabetic. In Python, objects of the `str` (string) type hold zero or more symbols in an ordered sequence. Strings must be *delimited* to distinguish them from variable names and other identifiers which we'll see later. Strings may be delimited with single quotation marks, double quotation marks, or "triple quotes." Examples of `str` literals: `"abc"`, `"123"`, `"vegetable"`, `"My hovercraft is full of eels."`, `"""What nonsense is this?"""`, *etc.*

Single and double quotation marks are equivalent when delimiting strings, but you must be consistent in their use—starting and ending delimiters must be the same. `"foo"` and `'foo'` are both valid string literals; `"foo'` and `'foo"` are not.

```
>>> "foo'
  File "<stdin>", line 1
    "foo'
    ^
SyntaxError: unterminated string literal (detected at line 1)
```

It is possible to have a string without any characters at all! We call this the *empty string*, and we write it `''` or `""` (just quotation marks with nothing in between).

Triple-quoted strings have special meaning in Python, and we'll see more about that in Chapter 6, on style. These can also be used for creating multi-line strings. Multi-line strings are handy for things like email templates and longer text, but in general it's best to use the single- or double-quoted versions.

`bool`

`bool` type is used for two special values in Python: `True` and `False`. `bool` is short for "Boolean", named after George Boole (1815–1864), a largely self-taught logician and mathematician, who devised Boolean logic—a cornerstone of modern logic and computer science (though computers did not yet exist in Boole's day).

There are only two literals of type `bool`: `True` and `False`. Notice that these are not strings, but instead are special literals of this type (so there aren't any quotation marks, and capitalization is significant).[1]

`NoneType`

`NoneType` is a special type in Python to represent the absence of a value. This may seem a little odd, but this comes up quite often in programming. There is *exactly one* literal of this type: `None` (and indeed there is exactly one instance of this type).

Like `True` and `False`, `None` is not a string, but rather a special literal.

`tuple`

A *tuple* is an *immutable sequence* of zero or more values. If an object is **immutable**, this means it cannot be changed once it's been created. Tuples are constructed using the comma to separate values. The *empty tuple*, `()`, is a tuple containing no elements.

The elements of a tuple can be of any type—including another tuple! The elements of a tuple needn't be the same type. That is, tuples can be *heterogeneous*.

While not strictly required by Python syntax (except in the case of the empty tuple), it is conventional to write tuples with enclosing parentheses. Examples of tuples: `()`, `(42, 71, 99)`, `(x, y)`, `('cheese', 11, True)`, *etc.*

A complete introduction to tuples appears in Chapter 10.

`list`

A *list* is a *mutable sequence* of zero or more values. If an object is **mutable**, then it can be changed after it is created (we'll see how to mutate lists later). Lists must be created with square brackets and elements

[1] In some instances, it might be helpful to interpret these as "on" and "off" but this will vary with context.

within a list are separated by commas. The *empty list*, `[]`, is a list containing no elements.

The elements of a list can be of any type—including another list! The elements of a list needn't be the same type. That is, like tuples, lists can be *heterogeneous*.

Examples of lists: `[]`, `['hello']`, '`['Larry', 'Moe', 'Curly']`, `[3, 6, 9, 12]`, `[a, b, c]`, `[4, 'alpha', ()]`, *etc.*

A complete introduction to lists appears in Chapter 10.

`dict`

`dict` is short for *dictionary*. Much like a conventional dictionary, Python dictionaries store information as pairs of keys and values. We write dictionaries with curly braces. Keys and values come in pairs, and are written with a colon separating key from value.

There are significant constraints on dictionary keys (which we'll see later in Chapter 16). However, dictionary values can be just about anything—including lists, tuples, and other dictionaries! Like lists, dictionaries are mutable. Example:

```
{'Egbert': 19, 'Edwina': 22, 'Winston': 35}
```

A complete introduction to dictionaries appears in Chapter 16.

The first few types we'll investigate are `int` (integer), `float` (floating-point number), `str` (string), and `bool` (Boolean). As noted, we'll learn more about other types later.

For a complete reference of built-in Python types, see: https://docs.python.org/3/library/stdtypes.html

3.2 Dynamic typing

You may have heard of "strongly typed" languages or "weakly typed" languages. These terms do not have precise definitions, and they are of limited utility. However, it's not uncommon to hear people referring to Python as a weakly typed language. This is *not* the case. If we're going to use these terms at all, Python exists toward the *strong* end of the spectrum. Python prevents most type errors at runtime, and performs very few implicit conversions between types—hence, it's more accurately characterized as being strongly typed.

Static and dynamic typing

Much more useful—and precise—are the concepts of *static typing* and *dynamic typing*. Some languages are *statically typed*, meaning that types are known at compile time—and types of objects (variables) cannot be changed at *runtime*—the time when the program is run.

Python, however, is *dynamically typed*. This means that the types of variables can change at runtime. For example, this works just fine in Python:

```
>>> x = 1
>>> print(type(x))
<class 'int'>
>>> x = 'Hey! Now I am a string!'
>>> print(type(x))
<class 'str'>
```

This demonstrates dynamic typing. When we first create the variable `x`, we assign to it the literal value `1`. Python understands that `1` is an integer, and so the result is an object of type `'int'` (which is short for "integer"). On the next line, we print the type of `x`, and Python prints: `<class 'int'>` as we'd expect. Then, we assign a new value to `x`, and Python doesn't miss a beat. Since Python is dynamically typed, we can change a variable's type at runtime. When we assign to `x` the value `'Hey! Now I am a string!'`, the type of `x` becomes `'str'` (which is short for "string").

In statically typed languages (say, C or Java or Rust) if we were to attempt something similar, we'd receive an error at compile time.

For example, in Java:

```
int x = 1;
x = "Hey! Now I am a string!";
```

would result in a compile-time error: "`incompatible types: java.lang.String cannot be converted to int`".

Notice that, unlike Python, when declaring a variable in Java a type annotation *must* be supplied, and once something is declared as a given type, that type *cannot* be changed.

It's important to note that it's not the type annotation

```
int x = 1
```

that makes Java statically typed. For example, other languages have *type inference* but are still statically typed (Python has limited type inference). *Type inference* is when the compiler or interpreter can *infer* something's type without having to be told explicitly "this is a string" or "this is an integer." For example, in Rust:

```
let x = 1;
x = "Hey! Now I am a string!";
```

would, again, result in a compile-time error: "`mismatched types... expected integer, found &str`".

While dynamic typing is convenient, this does place additional responsibility on you the programmer. This is particularly true since Python, unlike many other languages, doesn't care a whit about the types of formal parameters or return values of functions. Some languages ensure that programmers can't write code that calls a function with arguments of the wrong type, or return the wrong type of value from a function.

Python does not. Python won't enforce the correct use of types—that's up to you!

3.3 Types and memory

The details of how Python stores objects in memory is outside the scope of this text. Nevertheless, a little peek can be instructive.

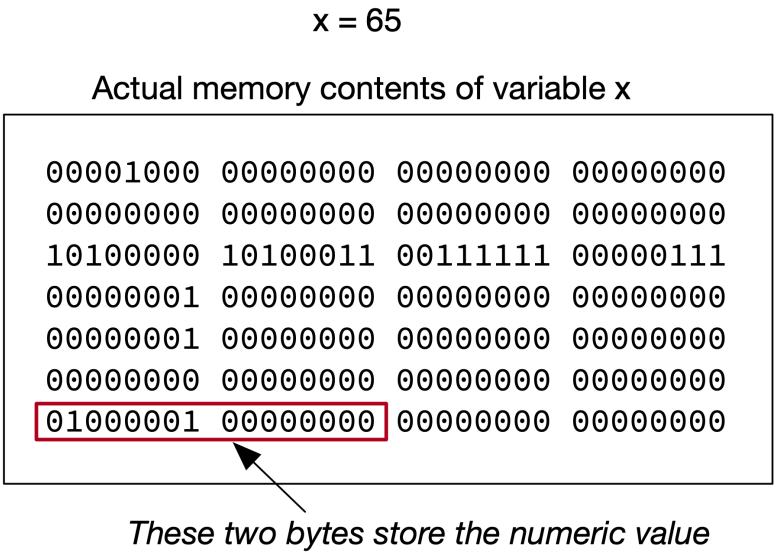


Figure 3.1: A sneak peek into an `int` object

Figure 3.1 includes a representation of an integer with value (decimal) 65. In binary, decimal 65 is represented as `01000001`. That's

$$0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Find `01000001` within the bitstring[2] shown in Figure 3.1. That's the integer value.[3]

Figure 3.2 shows the representation of the string `'A'`.[4] The letter 'A' is represented with the value (code point) of 65.

[2] A *bitstring* is just a sequence of zeros and ones.

[3] Actually, the value is stored in two bytes `01000001 00000000` as shown within the box in Figure 3.1. This layout in memory will vary with the particular implementation on your machine.

[4] Python uses Unicode encoding for strings. For reading on character encodings, don't miss Joel Spolsky's "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)".

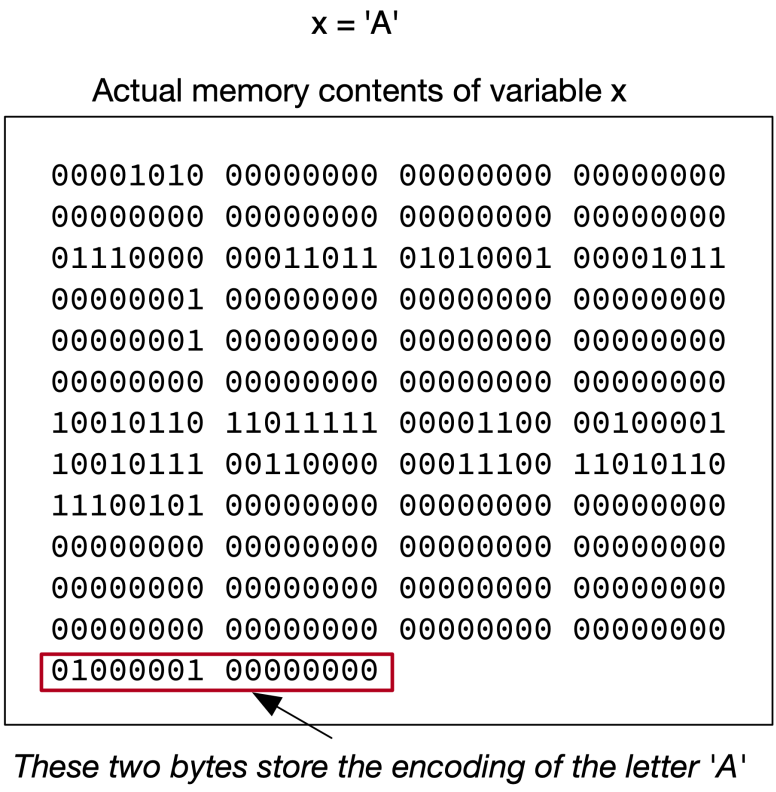


Figure 3.2: A sneak peek into a `str` object

Again, find `01000001` within the bitstring Figure 3.2—that's the encoding of `'A'`.

Apart from both representations containing the value 65 (`01000001`), notice how different the representations of an integer and a string are! How does Python know to interpret one as an integer and the other as a string? The type information is encoded in this representation (that's a part of what all the other ones and zeros are). That's how Python knows. That's one reason types are crucial!

Note: Other languages do this differently, and the representations (above) will vary somewhat depending on your machine's architecture.

Now, you don't need to know all the details of how Python uses your computer's memory in order to write effective programs, but this should give you a little insight into one reason why we need types. What's important for you as a programmer is to understand that different types have different behaviors. There are things that you can do with an integer that you can't do with a string, and *vice versa* (and that's a good thing).

3.4 More on string literals

Strings as ordered collections of characters

As we've seen, strings are ordered collections of characters, delimited by quotation marks. But what kind of characters can be included in a string?

Since Python 3.0, strings are composed of *Unicode* characters.[5]

> Unicode, formally The Unicode Standard, is an information technology standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. The standard, which is maintained by the Unicode Consortium, defines as of the current version (15.0) 149,186 characters covering 161 modern and historic scripts, as well as symbols, thousands of emoji (including in colors), and non-visual control and formatting codes.[6]

That's a lot of characters!

We won't dive deep into Unicode, but you should be aware that Python uses it, and that `"hello"`, `"Γειά σου"`, and `"привіт"` are all valid strings in Python. Strings can contain emojis too!

Strings containing quotation marks or apostrophes

You've learned that in Python, we can use either single or double quotation marks to delimit strings.

```
>>> 'Hello World!'
'Hello World!'
>>> "Hello World!"
'Hello World!'
```

Both are syntactically valid, and Python does not differentiate between the two.

It's not unusual that we have a string which contains quotation marks or apostrophes. This can motivate our choice of delimiters.

For example, given the name of a local coffee shop, Speeder and Earl's, there are two ways we could write this in Python. One approach would be to *escape* the apostrophe within a string delimited by single quotes:

```
>>> 'Speeder and Earl\'s'
"Speeder and Earl's"
```

Notice what's going on here. Since we want an apostrophe within this string, if we use single quotes, we precede the apostrophe with `\`. This

[5] You may have heard of Unicode, or perhaps ASCII (American Standard Code for Information Interchange). ASCII was an early standard and in Python was superseded in 2008 with the introduction of Python 3.

[6] https://en.wikipedia.org/wiki/Unicode

is called *escaping*, and it tells Python that what follows should be interpreted as an apostrophe and not a closing delimiter. We refer to the string `\'`, as an *escape sequence*.[7]

What would happen if we left that out?

```
>>> 'Speeder and Earl's'
Traceback (most recent call last):
  ...
  File "<input>", line 1
    'Speeder and Earl's'
                       ^
SyntaxError: unterminated string literal (detected at line 1)
```

What's going on here? Python reads the second single quote as the ending delimiter, so there's an extra—syntactically invalid—trailing `s'` at the end.

Another approach is to use double quotations as delimiters.

```
>>> "Speeder and Earl's"
"Speeder and Earl's"
```

The same applies to double quotes within a string. Let's say we wanted to print

> "Medium coffee, please", she said.

We could escape the double quotes within a string delimited by double quotes:

```
>>> "\"Medium coffee, please\", she said."
'"Medium coffee, please", she said.'
```

However, it's a little tidier in this case to use single quote delimiters.

```
>>> '"Medium coffee, please", she said.'
'"Medium coffee, please", she said.'
```

What happens if we have a string with both apostrophes and double quotes?

Say we want the string

> "I'll have a Speeder's Blend to go", she said.

What now? Now we must use escapes. Either of the following work:

[7] *Escape sequence* is a term whose precise origins are unknown. It's generally understood to mean that we use these sequences to "escape" from the usual meaning of the symbols used. In this particular context, it means we don't treat the apostrophe following the slash as a string delimiter (as it would otherwise be treated), but rather as a literal apostrophe.

```
>>> '"I\'ll have a Speeder\'s Blend to go", she said.'
'"I\'ll have a Speeder\'s Blend to go", she said.'
>>> print('"I\'ll have a Speeder\'s Blend to go", she said.')
"I'll have a Speeder's Blend to go", she said.
```

or

```
>>> "\"I'll have a Speeder's Blend to go\", she said."
'"I\'ll have a Speeder\'s Blend to go", she said.'
>>> print("\"I'll have a Speeder's Blend to go\", she said.")
"I'll have a Speeder's Blend to go", she said.
```

Not especially pretty, but there you have it.

More on escape sequences

We've seen how we can use the escape sequences `\'` and `\"` to avoid having the apostrophe and quotation mark treated as string delimiters, thereby allowing us to use these symbols within a string literal.

There are other escape sequences which work differently. The escape sequences `\n` and `\t` are used to insert a newline or tab character into a string, respectively. The escape sequence `\\` is used to insert a single backslash into a string.

Escape sequence	meaning
`\n`	newline
`\t`	tab
`\\`	backslash
`\'`	single quote / apostrophe
`\"`	double quote

Python documentation for strings

For more, see the Python documentation for strings, including *An Informal Introduction to Python*[8] and *Lexical Analysis*.[9]

3.5 Representation error of numeric types

Representation error occurs when we try to represent a number using a finite number of bits or digits which cannot be accurately represented in the system chosen. For example, in our familiar decimal system:

[8] https://docs.python.org/3/tutorial/introduction.html#strings
[9] https://docs.python.org/3/reference/lexical_analysis.html#literals

number	decimal representation	representation error
1	1	0
1/3	0.3333333333333333	0.0000000000000000333...
1/7	0.1428571428571428	0.00000000000000000571428...

Natural numbers, integers, rational numbers, and real numbers

You probably know that the set of all natural numbers

$$\mathbb{N} = \{0, 1, 2, 3, ...\}$$

is infinite.

From there it's not a great leap to see that the set of all integers

$$\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$$

is infinite too.

The rational numbers, $\mathbb{Q}$, and set of all real numbers, $\mathbb{R}$, also are infinite.

This fact—that these sets are of infinite size—has implications for numeric representation and numeric calculations on computers.

When we work with computers, numbers are given integer or floating-point representations. For example, in Python, we have distinct types, `int` and `float`, for holding integer and floating-point numbers, respectively.

I won't get into too much detail about how these are represented in binary, but here's a little bit of information.

Integers

Representation of integers is relatively straightforward. Integers are represented as binary numbers with a position set aside for the sign. So, 12345 would be represented as `00110000 00111001`. That's

$$\begin{aligned}0 \times 2^{15} + 0 \times 2^{14} + 1 \times 2^{13} + 1 \times 2^{12} \\ + 0 \times 10^{11} + 0 \times 10^{10} + 0 \times 10^{9} + 0 \times 10^{8} \\ + 0 \times 10^{6} + 1 \times 2^{5} + 1 \times 2^{4} + 1 \times 2^{3} \\ + 0 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0}\end{aligned}$$

This works out to

$$8192 + 4096 + 32 + 16 + 8 + 1 = 12345.$$

Negative integers are a little different. If you're curious about this, see the Wikipedia article on *Two's Complement*.

Floating-point numbers and IEEE 754

Floating-point numbers are a little tricky. Stop and think for a minute: How would *you* represent floating-point numbers? (It's not as straightforward as you might think.)

Floating-point numbers are represented using the IEEE 754 standard (IEEE stands for "Institute of Electrical and Electronics Engineers").[10] There are three parts to this representation: the sign, the exponent, and the fraction (also called the *mantissa* or *significand*)—and all of these must be expressed in binary. IEEE 754 uses either 32 or 64 bits for representing floating point numbers. The issue with representation lies in the fact that there's a fixed number of bits available: one bit for the sign of a number, eight bits for the exponent, and the rest for the fractional portion. With a finite number of bits, there's a finite number of values that can be represented without error.

Examples of representation error

Now we have some idea of how integers and floating-point numbers are represented in a computer. Consider this: We have some fixed number of bits set aside for these representations.[11] So we have a limited number of bits we can use to represent numbers on a computer. Do you see the problem?

The set of all integers, $\mathbb{Z}$, is infinite. The set of all real numbers, $\mathbb{R}$, is infinite. Any computer we can manufacture is finite. Now do you see the problem?

There exist infinitely more integers, and infinitely more real numbers, than we can represent in *any* system with a fixed number of bits. Let that soak in.

For any given machine or finite representation scheme, there are *infinitely many numbers that cannot be represented in that system*! This means that many numbers are represented by an approximation only.

Let's return to the example of 1/3 in our decimal system. We can never write down enough digits to the right of the decimal point so that we have the exact value of 1/3.

$$0.33333333333333333333\ldots$$

No matter how far we extend this expansion, the value will only be an *approximation of 1/3*. However, the fact that its decimal expansion is non-terminating is determined by the choice of base (10).

What if we were to represent this in base 3? In base 3, decimal 1/3 is 0.1. In base 3, it's easy to represent!

Of course our computers use binary, and so in that system (base 2) there are some numbers that can be represented accurately, and an infinite number that can only be approximated.

Here's the canonical example, in Python:

[10] For more, see: https://en.wikipedia.org/wiki/IEEE_754

[11] That's not entirely true for integers in Python, but it's reasonable to think of it this way for the purpose at hand.

```
>>> 0.1
0.1
>>> 0.2
0.2
>>> 0.1 + 0.2
0.30000000000000004
```

Wait! What? Yup. Something strange is going on. Python rounds values when displaying in the shell. Here's proof:

```
>>> print(f'{0.1:.56f}')
0.10000000000000000555111512312578270211815834045410156250
>>> print(f'{0.2:.56f}')
0.20000000000000001110223024625156540423631668090820312500
>>> print(f'{0.1 + 0.2:.56f}')
0.30000000000000004440892098500626161694526672363281250000
```

The last, `0.1 + 0.2`, is an example of representation error that accumulates to the point that it is no longer hidden by Python's automatic rounding, hence

```
>>> 0.1 + 0.2
0.30000000000000004
```

Remember we only work with powers of two. So there's no way to accurately represent these numbers in binary with a fixed number of decimal places.

What's the point?

1. The subset of real numbers that can be accurately represented within a given positional system depends on the base chosen (1/3 cannot be represented without error in the decimal system, but it can be in base 3).
2. It's important that we understand that *no finite machine can represent all real numbers without error*.
3. Most numbers that we provide to the computer and which the computer provides to us in the form of answers are *only approximations*.
4. Perhaps most important from a practical standpoint, *representation error can accumulate with repeated calculations*.
5. Understanding representation error can prevent you from chasing bugs when none exist.

For more, see:

- *Floating Point Arithmetic: Issues and Limitations*: https://docs.python.org/3.10/tutorial/floatingpoint.html.

3.6 Exercises

Exercise 01

Give the type of each of the following literals:

a. `42`
b. `True`
c. `"Burlington"`
d. `-17.45`
e. `"100"`
f. `"3.141592"`
g. `"False"`

You may check your work in the Python shell, using the built-in function `type()`. For example,

```
>>> type(777)
<class 'int'>
```

This tells us that the type of `777` is `int`.

Exercise 02

What happens when you enter the following in the Python shell?

a. `123.456.789`
b. `123_456_789`
c. `hello`
d. `"hello'`
e. `"Hello" "World!"` (this one may surprise you!)
f. `1,000` (this one, too, may surprise you!)
g. `1,000.234`
h. `1,000,000,000`
i. `'1,000,000,000'`

Exercise 03

The following all result in `SyntaxError`. Fix them!

a. `'Still the question sings like Saturn's rings'`
b. `"When I asked him what he was doing, he said "That isn't any business of yours.""`
c. `'I can't hide from you like I hide from myself.'`
d. `What's up, doc?`

Exercise 04 (challenge!)

We've seen that representation error occurs for most floating-point decimal values. Can you find values in the interval $[0.0, 1.0)$ that do *not* have representation error? Give three or four examples. What do all these examples have in common?

Chapter 4

Variables, statements, and expressions

In this chapter, we'll learn about variables and assignment, and the difference between *statements* (code which has no evaluation) and *expressions* (which have evaluations). We'll also learn about two additional arithmetic operators: floor division using the `//` operator (also called Euclidean division or integer division), and the modulo operator `%` (also called the remainder operator). Please note that the modulo operator has nothing to do with calculating percentages—this is a common confusion for beginners.

Learning objectives

- You will learn how to use the assignment operator and how to create and name variables.
- You will learn how to use the addition, subtraction, multiplication, division, and exponentiation operators.
- You will learn the difference between and use cases of division and Euclidian division (integer division).
- You will learn about modular arithmetic and how to use the remainder or "modulo" operator and floor division operator.
- You will learn operator precedence in Python.

Terms introduced

- absolute value
- assignment
- congruence
- dividend
- divisor
- Euclidean division
- evaluation
- exception
- expression
- floor function
- identifier

- modulus
- name
- operator
- quotient
- remainder
- statement
- variable

4.1 Variables and assignment

You have already written a "Hello, World!" program. As you can see, this isn't very flexible—you provided the exact text you wanted to print. However, more often than not, we don't know the values we want to use in our programs when we write them. Values may depend on user input, database records, results of calculations, and other sources that we cannot know in advance when we write our programs.

Imagine writing a program to calculate the sum of two numbers and print the result. We could write,

```
print(1 + 1)
print(2 + 2)
...
```

but that's really awkward. For every sum we want to calculate, we'd have to write another statement.

So when we write computer programs we use *variables*. In Python, a **variable** is the combination of a *name* (*identifier*) and an associated *value* which has a specific *type*.[1]

It's important to note that variables in a computer program are *not* like variables you've learned about in mathematics. For example, in mathematics we might write $a+b = 5$ and, of course, there's an infinite number of possible pairs of values which sum to five.

When writing computer programs, variables are rather different. While the same name can refer to different values at different times, a name can refer to only *one* value at a time.

Assignment statements

In Python, we use the `=` to assign a value to a variable, and we call `=` the *assignment operator*. The variable name is on the left-hand side of the assignment operator, and the expression (which evaluates to some value) is on the right-hand side of the assignment operator.

[1] Python differs from most other programming languages in this regard. In many other programming languages, variables refer to memory locations which hold values. (Yes, deep down, this is what goes on "under the hood" but the paradigm from the perspective of those writing programs in Python is that variables are *names* attached to *values*.) Feel free to check the entry in the glossary for more.

```
a = 3        # the variable named `a` has the value 3
print(a)     # prints 3 to the console
a = 17       # now the variable named `a` has the value 17
print(a)     # prints 17 to the console
```

Assignment is a kind of *statement* in Python. Assignment statements associate a name with a value.

Beginners often get confused about the assignment operator. You may find it helpful to think of it as a left-pointing arrow.[2] When reading your code, for example

```
a = 42
```

it may help to say, "*Let* a equal 42", or "a *gets* 42", rather than "a equals 42" (which sounds more like a claim or assertion about the value of `a`). This can reinforce the concept of assignment.[3]

Dynamic typing

In Python, all values have a *type*, and Python knows the type of each value at every instant. However, Python is a *dynamically typed* language. This means that any given *name* can refer to values of different types at different points in a program. So this is valid Python:

```
a = 42      # now `a` is of type int
print(a)    # prints 42 to the console
a = 'abc'   # now `a` is of type str
print(a)    # prints 'abc' to the console
```

Evaluation and assignment

Sometimes we can use a variable in some calculation and reassign the result. For example:

```
x = 0
print(x)    # prints 0 to the console
x = x + 1
print(x)    # prints 1 to the console
x = x + 1
print(x)    # prints 2 to the console
```

[2] In fact, the left-facing arrow is commonly used to indicate assignment in *pseudocode*—descriptions of algorithms outside the context of any particular programming language.

[3] Later on, we'll see the comparison operator `==`. This is used to compare two values to see if they are identical. For example, `a == b` would be *true* if the values of `a` and `b` were the same. So it's important to keep the assignment (`=`) and comparison (`==`) operators straight in your mind.

What's going on here? Remember, `=` is the *assignment* operator. So in the code snippet above, we're not making assertions about equivalence; instead, we're assigning values to `x`. With:

```
x = 0
```

we're assigning the literal value zero to `x`. At this point we can say the value of `x` is zero.

Consider what happens here:

```
x = x + 1
```

So first, Python will evaluate the expression on the right, and then it will assign the result to `x`. At the start, the value of `x` is still zero, so we can think of Python substituting the value of `x` for the object `x` on the right hand side.

```
x = 0 + 1
```

and then evaluating the right-hand side:

```
x = 1
```

and assigning the result to `x`. Now the value of `x` is one. If we do it again,

```
x = x + 1
```

now the `x` on the right has the value one, and one plus one is two, so the variable `x` has the value two.

Variables are *names* associated with values

What are variables in Python? Variables work differently in Python than they do in many other languages. Again, in Python, a variable is a name (identifier) associated with a value.

Consider this code:

```
>>> x = 1001
>>> y = x
```

What we've done here is give two different names to the same value. This is A-OK in Python. What does `x` refer to? The value 1001. What does `y` refer to? *The exact same 1001.*[4] It is *not* the case that there are two different locations in memory both holding the value 1001 (as might be the case in a different programming language).

Now what happens if we assign a new value to `x`? Does `y` "change"? What do you think?

[4] We can verify this by inspecting the identity number of the object(s) in question using Python's built-in `id()` function.

```
>>> x = 2001
>>> x
2001
>>> y
1001
```

No. Even though `x` now has the new value of 2001, `y` is unchanged and still has the value of 1001.

When we assign a value to a variable,

```
>>> x = 1001
```

what's really going on is that we're associating a *name* with a value. In the above example, 1001 is the value, and `x` is a name we've given to it.

Values can have more than one name associated with them. In fact, we can give any number of names to the same value.

```
>>> x = 1001
>>> y = x
>>> z = y
```

Now what happens if we assign a new value to `x`?

```
>>> x = 500
>>> x
500
>>> y
1001
>>> z
1001
```

`y` and `z` are still names for 1001, but now the name `x` is associated with a new value, 500.

While it's true that values can have more than one name associated with them, it's important to understand that each name can only refer to a single value (or object). `x` can't have two different values at the same time.

```
>>> x = 3
>>> x
3
>>> x = 42  # What happened to 3? Gone forever.
>>> x
42
```

Comprehension check

Given the following snippets of Python code, determine the resulting value `x`:

1.

```
x = 1
```

2.

```
x = 1
x = x + 1
```

3.

```
y = 200
x = y
```

4.

```
x = 0
x = x * 200
```

5.

```
x = 1
x = 'hello'
```

6.

```
x = 5
y = 3
x = x + 2 * y - 1
```

Constants

A lot of the time in programming, we want to use a specific value or calculation multiple times. Instead of repeating that same value or calculation over and over again, we can just assign the value to a variable and reuse it throughout a program. We call this *constant*. A **constant** is a variable that has a value that will be left unchanged throughout a program. Using constants improves the readability of programs because they provide meaningful and recognizable names for fixed values. Let's look at an example:

```
HOURS_IN_A_DAY = 24
```

Here we have assigned the variable `HOURS_IN_A_DAY` to `24`. This variable is a constant because the number of hours in a day will always be `24` (at least for the foreseeable future). Now if we need to do some calculation using the number of hours in a day, we can just use this variable. Note that constants are uppercase. This isn't enforced by Python, but it's good common practice.

4.2 Expressions

In programming—and computer science in general—an **expression** is something which can be *evaluated*—that is, a syntactically valid combination of constants, variables, functions, and operators which yields a *value*.

Let's try out a few expressions with the Python shell.

Literals and types revisited

The simplest possible expression is a single *literal*.

```
>>> 1
1
```

What just happened? We typed a simple expression—a single literal—and Python replied with its value. Literals are special in that they evaluate to themselves!

Here's another:

```
>>> 'Hello, Python!'
'Hello, Python!'
```

Once again, we've provided a single literal, and again, Python has replied with its value.

You may notice that `'Hello, Python!'` is rather different from `1`. You might say these are literals of different *types*—and you'd be correct! Literals come in different types. Here are four different literals of four different types.

`'Hello, Python'`	string (`str`)
`1`	integer (`int`)
`3.141592`	floating-point (`float`)
`True`	Boolean (`bool`)

`'Hello, Python!'` is a *string* literal. The quotation marks *delimit* the string. They let Python know that what's between them is to be interpreted as a string, but *they are not part of the string itself*. Python allows single-quoted or double-quoted strings, so `"Hello, Python!"` and `'Hello, Python!'` are both syntactically correct. Note that if you start with a single quote (`'`), you must end with a single quote. Likewise, if you start with a double quote (`"`), you must end with a double quote.

`1` is different. It is an *integer* literal. Notice that there are no quotation marks around it.

Given all this, the latter two examples work as you'd expect.

```
>>> 3.141592
3.141592
>>> True
True
```

What types are these? `3.141592` is a *floating point* literal (that's a number that has something to the right of the decimal point). `True` is what's called a *Boolean* literal. Notice there are no quotation marks around it and the first letter is capitalized. `True` and `False` are the only two Boolean literals.

Expressions with arithmetic operators

Let's try some more complex expressions. In order to construct more complex expressions we'll use some simple arithmetic operators, specifically some *binary infix operators*. These should be very familiar. A *binary operator* is one which operates on two *operands*. The term *infix* means that we put the operator *between* the operands.

```
>>> 1 + 2
3
```

Surprised? Probably not. But let's consider what just happened anyway.

At the prompt, we typed `1 + 2` and Python responded with `3`. `1` and `2` are integer literals, and `+` is the operator for addition. `1` and `2` are the operands, and `+` is the operator. This combination `1 + 2` is a syntactically valid Python expression which evaluates to… you guessed it, `3`.

Some infix arithmetic operators in Python are:

`+`	addition
`-`	subtraction
`*`	multiplication (notice we use `*` and not `x`)
`/`	division
`//`	integer or "floor" division
`%`	remainder or "modulo"
`**`	exponentiation

There are other operators, but these will suffice for now. Here we'll present examples of the first four, and we'll present the others later—floor division, modulo, and exponentiation. Let's try a few (I encourage you follow along and try these out in the Python shell as we go).

```
>>> 40 + 2
42
>>> 3 * 5
15
```

```
>>> 5 - 1
4
>>> 30 / 3
10.0
```

Notice that in the last case, when performing division, Python returns a floating-point number and not an integer (Python does support what's called *integer division* or *floor division*, but we'll get to that later). So even if we have two integer operands, division yields a floating-point number.

What do you think would be the result if we were to add the following?

```
>>> 1 + 1.0
```

In a case like this, Python performs *implicit type conversion*, essentially promoting `1` to `1.0` so it can add like types. Accordingly, the result is:

```
>>> 1 + 1.0
2.0
```

Python will perform similar type conversions in similar contexts:

```
>>> 2 - 1.0
1.0
>>> 3 * 5.0
15.0
```

Precedence of operators

No doubt you've learned about *precedence of operations*, and Python respects these rules.

```
>>> 40 + 2 * 3
46
>>> 3 * 5 - 1
14
>>> 30 - 18 / 3
24.0
```

Multiplication and division have higher precedence than addition and subtraction. We also say multiplication and division *bind* more strongly than addition and subtraction—this is just a different way of saying the same thing.

As you might expect, we can use parentheses to group expressions. We do this to group operations of lower precedence—either in order to perform the desired calculation, or to disambiguate or make our code easier to read, or both.

```
>>> 40 + (2 * 3)
46
>>> 3 * (5 - 1)
12
>>> (30 - 18) / 3
4.0
```

So what happens here? The portions within the parentheses are evaluated first, and then Python performs the remaining operation.

We can construct expressions of arbitrary complexity using these arithmetic operators and parentheses.

```
>>> (1 + 1) * (1 + 1 + 1) - 1
5
```

Python also has *unary operators*. These are operators with a single operand. For example, we negate a number by prefixing `-`.

```
>>> -1
-1
>>> -1 + 3
2
>>> 1 + -3
-2
```

We can also negate expressions within parentheses.

```
>>> -(3 * 5)
-15
```

Summary of operator precedence

`**`	exponentiation
`+`, `-`	unary positive or negative (`+x`, `-x`)
`*`, `/`, `//`, `%`	multiplication, and various forms of division
`+`, `-`	addition and subtraction (`x - y`, `x + y`)

Expressions grouped within parentheses are evaluated first, so the rule you might have learned in high school and its associated mnemonic—PEMDAS (parentheses, exponentiation, multiplication and division, addition and subtraction)—apply.

Comprehension check

1. When evaluating expressions, do you think Python proceeds left-to-right or right-to-left? Can you think of an experiment you might perform to test your hypothesis? Write down an expression that might provide some evidence.

2. Why do you think `1 / 1` evaluates to `1.0` (a float) and not just `1` (an integer)?

More on operations

So far, we've seen some simple expressions involving literals, operators, and parentheses. We've also seen examples of a few types: integers, floating-point numbers ("floats" for short), strings, and Booleans.

We've seen that we can perform arithmetic operations on numeric types (integers and floats).

The operators `+` and `*` applied to strings

Certain arithmetic operators behave differently when their operands are strings. For example,

```
>>> 'Hello' + ', ' + 'World!'
'Hello, World!'
```

This is an example of *operator overloading*, which is just a fancy way of saying that an operator behaves differently in different contexts. In this context, with strings as operands, `+` doesn't perform addition, but instead performs *concatenation*. **Concatenation** is the joining of two or more strings, like the coupling of railroad cars.

We can also use the multiplication operator `*` with strings. In the context of strings, this operator concatenates multiple copies of a string together.

```
>>> 'Foo' * 1
'Foo'
>>> 'Foo' * 2
'FooFoo'
>>> 'Foo' * 3
'FooFooFoo'
```

What do you think would be the result of the following?

```
>>> 'Foo' * 0
```

This gives us `''` which is called the *empty string* and is the result of concatenating zero copies of `'Foo'` together. Notice that the result is still a string, albeit an empty one.

4.3 Augmented assignment operators

As a shorthand, Python provides what are called *augmented assignment* operators. Here are some (but not all):

augmented assignment	simila to
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`

A common example is incrementing or decrementing by one.

```
>>> a = 0
>>> a += 1
>>> a
1
>>> a += 1
>>> a
2
>>> a -= 1
>>> a
1
>>> a -= 1
>>> a
0
```

You can use these or not, depending on your own preference.[5]

4.4 Euclidean or "floor" division

When presenting expressions, we saw examples of common arithmetic operations: addition, subtraction, multiplication, and division. Here we will present two additional operations which are closely related: the modulo or "remainder" operator, and what's variously called "quotient", "floor division", "integer division" or "Euclidean division."[6]

Chances are, when you first learned division in primary school, you learned about Euclidean (floor) division. For example, $17 \div 5 = 3$ r 2, or $21 \div 4 = 5$ r 1. In the latter example, we'd call 21 the dividend, 4 the divisor, 5 the quotient, and 1 the remainder.

[5] The table above says "similar to" because, for example, `a += b` isn't *exactly* the same as `a = a + b`. In augmented assignment, the left-hand side is evaluated *before* the right-hand side, then the right-hand side is evaluated and the result is assigned to the variable on the left-hand side. There are some other minor differences.

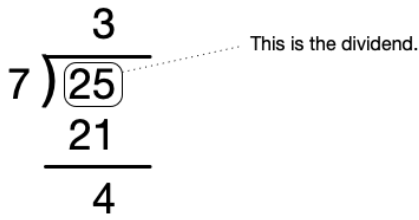
[6] Euclid had many things named after him, even if he wasn't the originator (I guess fame begets fame). Anyhow, Euclid was unaware of the division algorithm you're taught in primary school. Similar division algorithms depend on the positional system of Hindu-Arabic numerals, and these date from around the 12th Century CE. The algorithm you most likely learned, called "long division", dates from around 1600 CE.

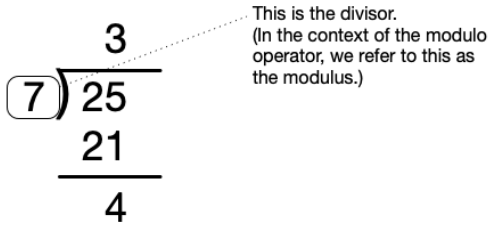
Obviously the operations of finding the quotient and the remainder are closely related. For any two integers a, b, with $b \neq 0$ there exist unique integers q and r such that

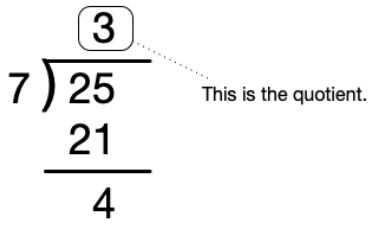
$$a = bq + r$$

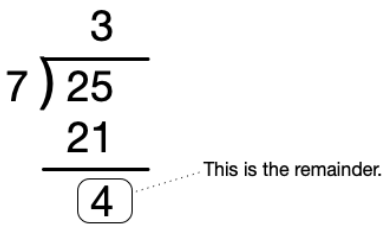
where q is the Euclidean quotient and r is the remainder.
Furthermore, $0 \leq r < |b|$, where $|b|$ is the absolute value of b. This should be familiar.

Just in case you need a refresher:









Python's `//` and `%` operators

Python provides us with operators for calculating quotient and remainder. These are `//` and `%`, respectively. Here are some examples:

```
>>> 17 // 5  # calculate the quotient
3
>>> 17 % 5   # calculate the remainder
2
>>> 21 // 4  # calculate the quotient
5
>>> 21 % 4   # calculate the remainder
1
```

You may ask: What's the difference between the division we saw earlier, `/`, and floor division with `//`? The difference is that `/` calculates the quotient as a decimal expansion. Here's a simple comparison:

```
>>> 4 / 3
1.3333333333333333   # three goes into four 1 and 1/3 times
>>> 4 // 3           # calculates Euclidean quotient
1
>>> 4 % 3            # calculates remainder
1
```

Common questions

What happens when the divisor is zero?

Just as in mathematics, we cannot divide by zero in Python either. So all of these operations will fail if the right operand is zero, and Python will complain: `ZeroDivisionError`.

What happens if we supply floating-point operands to `//` or `%`?

In both cases, operands are first converted to a common type. So if one operand is a `float` and the other an `int`, the `int` will be implicitly converted to a `float`. Then the calculations behave as you'd expect.

```
>>> 7 // 2    # if both operands are ints, we get an int
3
>>> 7.0 // 2  # otherwise, we get a float...
3.0
>>> 7 // 2.0
3.0
>>> 7.0 // 2.0
3.0
>>> 7 % 2     # if both operands are ints, we get an int
1
>>> 7.0 % 2   # otherwise, we get a float...
1.0
>>> 7 % 2.0
1.0
>>> 7.0 % 2.0
1.0
```

What if the dividend is zero?

What are `0 // n` and `0 % n`?

```
>>> 0 // 5    # five goes into zero zero times
0
>>> 0 % 5     # the remainder is also zero
0
```

What if the dividend is less than the divisor?

What are `m // n` and `m % n`, when $m < n$, with m and n both integers, and $m \geq 0$, $n > 0$?

The first one's easy: if $m < n$?, then `m // n` yields zero. The other trips some folks up at first.

```
>>> 5 % 7
5
```

That is, seven goes into five zero times and leaves a remainder of five. So if $m < n$, then `m % n` yields `m`.

What if the divisor is negative?

What is `m % n`, when $n < 0$? This might not work the way you'd expect at first.

```
>>> 15 // -5
-3
>>> 15 % -5
0
```

So far, so good. Now consider:

```
>>> 17 // -5
-4
>>> 17 % -5
-3
```

Why does `17 // -5` yield -4 and not -3? Remember that this is what's called "floor division." What Python does, is that it calculates the (floating-point) quotient and then applies the `floor` function.

The floor function is a mathematical function which, given some number x, returns the greatest integer less than or equal to x. In mathematics this is written as:

$$\lfloor x \rfloor.$$

So in the case of `17 // -5`, Python first converts the operands to `float` type, then calculates the (floating-point) quotient, which is -3.4 and then applies the `floor` function, to yield -4 (since -4 is the largest integer less than or equal to -3.4).

This also makes clear why `17 % -5` yields -3. This preserves the equality

$$\begin{aligned} a &= bq + r \\ 17 &= (-5 \times -4) + (-3) \\ 17 &= 20 - 3. \end{aligned}$$

What if the dividend is negative?

```
>>> -15 // 5
-3
>>> -15 % 5
0
```

So far so good. Now consider:

```
>>> -17 // 5
-4
>>> -17 % 5
3
```

Again, Python preserves the equality

$$\begin{aligned} a &= bq + r \\ -17 &= (5 \times -4) + 3 \\ -17 &= -20 + 3. \end{aligned}$$

Yeah. I know. These take a little getting used to.

What if dividend and divisor both are negative?

Let's try it out—having seen the previous examples, this should come as no surprise.

```
>>> -43 // -3
14
>>> -43 % -3
-1
```

Check this result:

$$\begin{aligned} a &= bq + r \\ -43 &= (-3 \times 14) + (-1) \\ -43 &= -42 - 1 \end{aligned}$$

The `%` operator will always yield a result with the same sign as the second operand (or zero).

You are encouraged to experiment with the Python shell. It is a great tool to further your understanding.

4.5 Modular arithmetic

Now, in the Python documentation[7], you'll see `//` referred to as floor division. You'll also see that `%` is referred to as the *modulo* operator.

It's fine to think about `%` as the remainder operator (with the provisos noted above), but what is a "modulo operator"?

Let's start with the example of clocks.

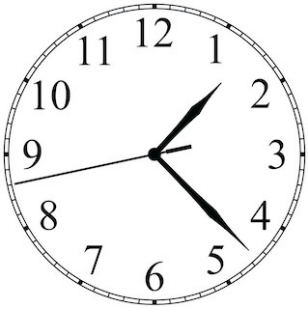


Figure 4.1: Clock face

Perhaps you don't realize it, but you do modular arithmetic in your head all the time. For example, if you were asked what time is 5 hours after 9 o'clock, you'd answer 2 o'clock. You wouldn't say 14 o'clock.[8] This is an example of modular arithmetic. In fact, modular arithmetic is sometimes called "clock arithmetic."

[7] https://docs.python.org/3/reference/expressions.html

[8] OK. Maybe in the military or in Europe you might, but you get the idea. We have a clock with numbers 12–11, and 12 hours brings us back to where we started (at least as far as the clock face is concerned). Notice also that the arithmetic is the same for an analog clock face with hands and a digital clock face. This difference in interface doesn't change the math at all, it's just that *visually* things work out nicely with the analog clock face.

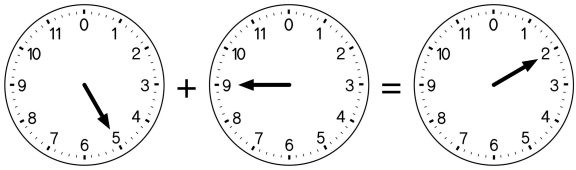


Figure 4.2: Clock arithmetic: $5 + 9 \equiv 2 \pmod{12}$

In mathematics, we would say that $5+9$ is *congruent* to 2 *modulo* 12, and we'd write

$$5 + 9 \equiv 2 \pmod{12}$$

So $5 + 9 = 14$ and $14 \div 12$ has a remainder of 2.

Here's another example:

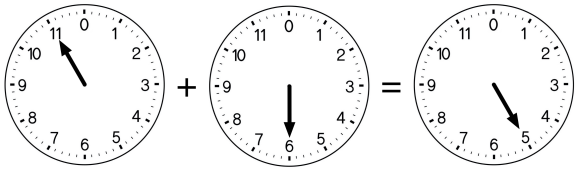


Figure 4.3: Clock arithmetic: $11 + 6 \equiv 5 \pmod{12}$

Similarly, we'd say that $11+6$ is congruent to 5 modulo 12, and we'd write

$$11 + 6 \equiv 5 \pmod{12}$$

Let's think of this a little more formally. Suppose we have some positive integer, n, which we call the *modulus*. We can perform arithmetic with respect to this integer in the following way. When counting, when we reach this number we start over at zero. Now in the case of clocks, this positive integer is 12, but it needn't be—we could choose any positive integer.

For example, with $n = 5$, we'd count

$$0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, ...$$

Notice that we never count to five, we start over at zero. You'll see that the clocks in the figures above don't have twelve on their face but instead have zero. If $n = 5$, then we'd have five positions on our "clock", numbered zero through four.

Under such a system, addition and subtraction would take on a new meaning. For example, with $n = 5$, $4 + 1 \equiv 0 \pmod{5}$,
$4 + 2 \equiv 1 \pmod{5}$,
$4 + 3 \equiv 2 \pmod{5}$,
and so on.

Things work similarly for subtraction, except we proceed anti-clockwise. For example $1 - 3 \equiv 3 \pmod{5}$.

The same principle applies to multiplication: $2 \times 4 \equiv 3 \pmod{5}$ and $3 \times 3 \equiv 4 \pmod{5}$.

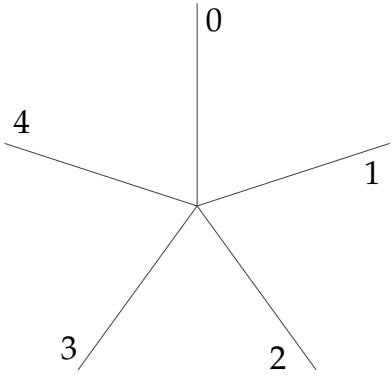


Figure 4.4: A "clock" for (mod 5)

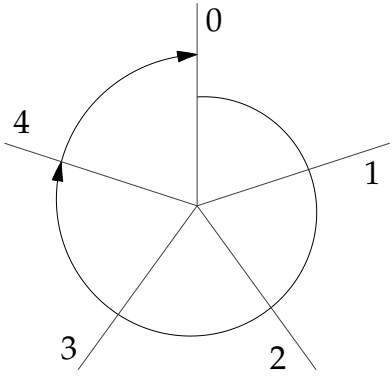


Figure 4.5: $4 + 1 \equiv 0 \pmod 5$

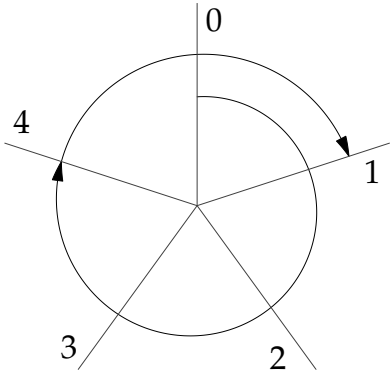


Figure 4.6: $4 + 2 \equiv 1 \pmod 5$

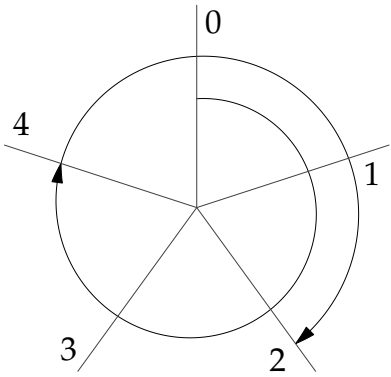


Figure 4.7: $4 + 3 \equiv 2 \pmod 5$

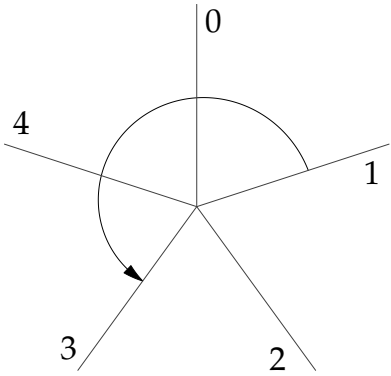


Figure 4.8: $1 - 3 \equiv 3 \pmod 5$

Negative modulus

We've seen that when we add we go clockwise, and when we subtract we go anti-clockwise. What happens when the modulus is negative?

To preserve the "direction" of addition (clockwise) and subtraction (anti-clockwise), if our modulus is negative we number the face of the clock anti-clockwise.

Examples:

$1 \equiv -4 \pmod{-5}$

$2 \equiv -3 \pmod{-5}$

$2 + 4 \equiv -4 \pmod{-5}$

We can confirm these agree with Python's evaluation of these expressions:

```
>>> 1 % -5
-4
>>> 2 % -5
```

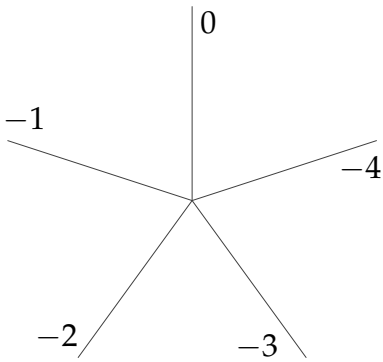


Figure 4.9: A "clock" for (mod -5)

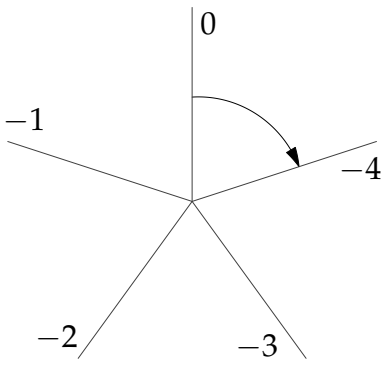


Figure 4.10: $1 \equiv -4 \pmod{-5}$

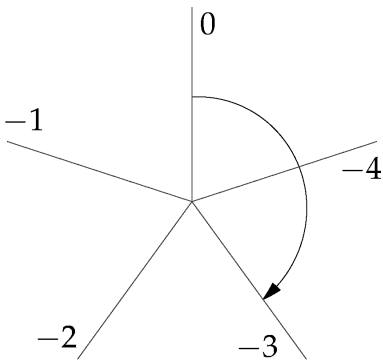


Figure 4.11: $2 \equiv -3 \pmod{-5}$

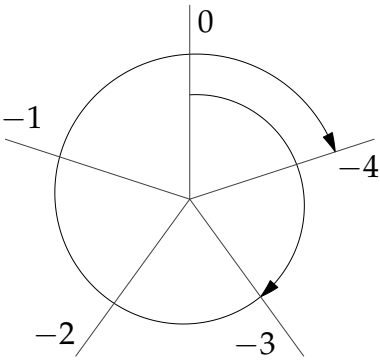


Figure 4.12: $2 + 4 \equiv -4 \pmod{-5}$

```
-3
>>> (4 + 2) % -5
-4
```

Why did I put `(4 + 2)` in parentheses? Because `%` has higher precedence than `+`. Again, try inputting your own expressions into the Python shell.

Some things to note

- If the modulus is an integer, $n > 0$, then the only possible remainders are $[0, 1, ..., n - 1]$.
- If the modulus is an integer, $n < 0$, then the only possible remainders are $[n + 1, ..., -1, 0]$.

What now?

This is actually a *big* topic, involving equivalence classes, remainders (in this context, called "residues"), and others. That's all outside the scope of this textbook. Practically, however, there are abundant applications for modular arithmetic, and this is something we'll see again and again in this text.

Some applications for modular arithmetic include:

- Hashing function
- Cryptography
- Primality and divisibility testing
- Number theory

Here are a couple of simple examples.

Example: eggs and cartons

Jenny has n eggs. If a carton holds 12 eggs, how many complete cartons can she make and how many eggs will be left over?

```
EGGS_PER_CARTON = 12
cartons = n // EGGS_PER_CARTON
leftover = n % EGGS_PER_CARTON
```

Example: even or odd

Given some integer n is n even or odd?

```
if n % 2 == 0:
    print(f'{n} is even')
else:
    print(f'{n} is odd')
```

(Yes, there's an even simpler way to write this. We'll get to that in due course.)

Comprehension check

1. Given some modulus, n, an integer, and some dividend, d, also an integer, what are the possible values of `d % n` if

   a. $n = 5$
   b. $n = -4$
   c. $n = 2$
   d. $n = 0$

2. Explain why, if the modulus is positive, the remainder can never be greater than the modulus.

3. The planet Zorlax orbits its sun every 291 1/3 Zorlaxian days. Thus, starting from the year one, every third year is a leap year on Zorlax. So the year three is a leap year. The year six is a leap year. The year 273 is a leap year. Write a Python expression which, given some integer y greater than or equal to one representing the year, will determine whether y represents a Zorlaxian leap year.

4. There's a funny little poem: Solomon Grundy— / Born on a Monday, / Christened on Tuesday, / Married on Wednesday, / Took ill on Thursday, / Grew worse on Friday, / Died on Saturday, / Buried on Sunday. / That was the end / Of Solomon Grundy.[9] How could this be? Was Solomon Grundy married as an infant? Did he die before he was a week old? What does this have to do with modular arithmetic? What if I told you that Solomon Grundy was married at age 28, and died at age 81? Explain.

[9] First recorded by James Orchard Halliwell and published in 1842. Minor changes to punctuation by the author.

4.6 Exponentiation

Exponentiation is a ubiquitous mathematical operation. However, the syntax for exponentiation varies between programming languages. In some languages, the caret (`^`) is the exponentiation operator. In other languages, including Python, it's the double-asterisk (`**`). Some languages don't have an exponentiation operator, and instead they provide a library function, `pow()`.

The reasons for these differences are largely historical. In mathematics, we write an exponent as a superscript, for example, x^2. However, keyboards and character sets don't know about superscripts,[10] and so the designers of programming languages had to come up with different ways of writing exponentiation.

`**` was first used in Fortran, which first appeared in 1957. This is the operator which Python uses for exponentiation.

For the curious, here's a table with some programming languages and the operators or functions they use for exponentiation.

`**`	Algol, Basic, Fortran, JavaScript, OCaml, Pascal, Perl, **Python**, Ruby, Smalltalk
`^`	J, Julia, Lua, Mathematica
`pow()`	C, C++, C#, Dart, Go, Java, Kotlin, Rust, Scala
`expt`	Lisp, Clojure

Exponentiation in Python

Now we know that `**` is the exponentiation operator in Python. This is an *infix* operator, meaning that the operator appears *between* its two operands. As you'd expect, the first operand is the *base*, and the second operand is the *exponent* or *power*. So,

```
b ** n
```

implements b^n.

Here are some examples,

Area of a circle of a given radius	`3.14159 * radius ** 2`
Kinetic energy, given mass and velocity	`(1 / 2) * m * v ** 2`

Also, as you'd expect, `**` has precedence over `*` so in the above examples, `radius ** 2` and `v ** 2` are calculated before multiplication with other terms.

[10] Picking nits, that's not entirely true, since nowadays there are some character sets that include superscript 2 and 3. But they're not understood as numbers and aren't useful in programming.

But wait! There's more!

You're going to find out sooner or later, so you might as well know now that Python also has a built-in function `pow()`. For our purposes, `**` and `pow()` are equivalent, so you may use either. Here's a session at the Python shell:

```
>>> 3 ** 2
9
>>> 3.0 ** 2
9.0
>>>
>>> pow(3, 2)
9
>>> pow(3.0, 2)
9.0
```

With two operands or arguments, `**` and `pow()` behave identically. If both operands are integers, and the exponent is positive, the result will be an integer. If one or more of the operands is a float, the result will be a float.

Negative or fractional exponents behave as you'd expect.

```
>>> 3 ** 0.5    # Calculates the square root of 3
1.7320508075688772
>>> 3 ** -1     # Calculates 1 / 3
0.3333333333333333
```

No surprises here.

$x^0 = 1$

Remember from algebra that any non-zero number raised to the zero power is one. If that weren't the case, what would become of this rule?

$$b^{m+n} = b^m \times b^n$$

So $x^0 = 1$ for all non-zero x. Python knows about that, too.

```
>>> 1 ** 0
1
>>> 2 ** 0
1
>>> 0.1 ** 0
1.0
```

What about 0^0? Many mathematics texts state that this should be undefined or indeterminate. Others say $0^0 = 1$. What do you think Python does?

```
>>> 0 ** 0
1
```

So, Python has an opinion on this.

Now, go forth and exponentiate!

A little puzzle

Consider the following Python shell session:

```
>>> pow(-1, 0)
1
>>> -1 ** 0
-1
```

What's going on here? The answer we get using `pow()` is what we'd expect. Shouldn't these both produce the same result? Can you guess why these yield different answers?

4.7 Exceptions

Exceptions are errors that occur at run time, that is, when you run your code. When such an error occurs Python raises an exception, prints a message with information about the exception, and then halts execution. Exceptions have different types, and this tells us about the kind of error that's occurred.

If there is a syntax error, an exception of type `SyntaxError` is raised. If there is an indentation error (a more specific kind of syntax error), an `IndentationError` is raised. These errors occur before your code is ever run—they are discovered as Python is first reading your file.

Most other exceptions occur as your program is run. In these cases, the message will include what's called a *traceback*, which provides a little information about where in your code the error occurred. The last line in an exception message reports the type of exception that has occurred. It's often helpful to read such messages from the bottom up.

What follows are brief summaries of the first types of exceptions you're likely to encounter, and in each new chapter, we'll introduce new exception types as appropriate.

`SyntaxError`

If you write code which does not follow the rules of Python syntax, Python will raise an exception of type `SyntaxError`. Example:

```
>>> 1 + / 1
  File "<stdin>", line 1
    1 + / 1
        ^
SyntaxError: invalid syntax
```

Notice that the ^ character is used to indicate the point at which the error occurred.

Here's another:

```
>>> True False
  File "<stdin>", line 1
    True False
         ^
SyntaxError: invalid syntax
```

When you encounter a syntax error, it means that some portion of your code does not follow the rules of syntax. Code which includes a syntax error cannot be executed by the Python interpreter, and syntax errors must be corrected before your code will run.

`NameError`

A `NameError` occurs when we try to use a name which is undefined. There must be a value assigned to a name before we can use the name.

Here's an example of a `NameError`:

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Notice that Python reports the `NameError` and informs you of the name you tried to use but which is undefined (in this case `x`).

These kinds of errors most often occur when we've made a typo in a name.

Depending on the root cause of the error, there are two ways to correct these errors.

- If the cause is a typo, just correct your typo.
- If it's not just a typo, then you must *define* the name by making an assignment with the appropriate name.

```
>>> pet = 'rabbit'
>>> print(pot)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pot' is not defined
>>> print(pet)
rabbit
```

```
>>> age = age + 1  # Happy birthday!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'age' is not defined
```

```
>>> age = 100
>>> age = age + 1  # Happy birthday!
>>> print(age)
101
```

TypeError

A `TypeError` occurs when we try to perform an operation on an object which does not support that operation.

The Python documentation states: "[`TypeError` is] raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch."[11]

For example, we can perform addition with operands of type `int` using the `+` operator, and we can concatenate strings using the same operator, but we cannot add an `int` to a `str`.

```
>>> 2 + 2
4
>>> 'fast' + 'fast' + 'fast'
'fastfastfast'
>>> 2 + 'armadillo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

When you encounter a `TypeError`, you must examine the operator and operands and determine the best fix. This will vary on a case-by-case basis.

Here are some other examples of `TypeError`:

```
>>> 'hopscotch' / 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

```
>>> 'barbequeue' + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

ZeroDivisionError

Just as we cannot divide by zero in mathematics, we cannot divide by zero in Python either. Since the remainder operation (`%`) and integer (a.k.a. floor) division (`//`) depend on division, the same restriction applies to these as well.

[11] https://docs.python.org/3/library/exceptions.html#TypeError

```
>>> 1000 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

4.8 Exercises

Exercise 01

Without typing these at a Python prompt first, determine the value of each of the following expressions. Once you've worked out what you think the evaluation should be, check your answer using the Python shell.

a. `13 + 6 - 1 * 7`

b. `(17 - 2) / 5`

c. `-5 / -1`

d. `42 / 2 / 3`

e. `3.0 + 1`

f. `1.0 / 3`

g. `2 ** 2`

h. `2 ** 3`

i. `3 * 2 ** 8 + 1`

Exercise 02

For each of the expressions in exercise 01, give the *type* of the result of evaluation. Example: `1 + 1` evaluates to `2` which is of type `int`.

Exercise 03

What is the evaluation of the following expressions?

a. `10 % 2`

b. `19 % 2`

c. `24 % 5`

d. `-8 % 3`

Exercise 04

What do you think would happen if we were to use the operands we've just seen with non-numeric types? For example, what do you think would happen if we were to enter the following. Then check your expectations using the Python interpreter. Any surprises?

a. `'Hello' + ', ' + 'World!'`

b. `'Hello' * 3`

c. `True * True`

d. `True * False`

e. `False * 42`

f. `-True`

g. `True + True`

Exercise 05

What is the difference between the following statements?

```
it_is_cloudy_today = True
it_will_rain_tomorrow = 'True'
```

Exercise 06

Some operands don't work with certain types. For example, the following will result in errors. Try these out at a prompt, and observe what happens. Make a note of the *type* of error which occurs.

a. `'Hello' / 3`

b. `-'Hello'`

c. `'Hello' - 'H'`

Exercise 07

a. Write a statement that assigns the value `79.95` to a variable name `subtotal`.

b. Write a statement that assigns the value `0.06` to a variable name `tax_rate`.

c. Write a statement that multiplies `subtotal` by `tax_rate` and assigns the result to a variable name `sales_tax`.

d. Write a statement that adds `subtotal` and `sales_tax` and assigns the result to a variable name `total`.

Exercise 08

What do you think would happen if we were to evaluate the expression `1 / 0`? Why? Does this result in an error? What type of error results?

Exercise 09

a. Now that we've learned a little about modular arithmetic, reconsider the numerals we use in our decimal (base 10) system. In that system, why do we have only the numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9?

b. What numerals would we need in a base 7 system? How about base 5?

Chapter 5

Functions

> Functions are the first forms of reusable code.
>
> –source unknown

> A function should do one thing, and do it well.
>
> –Robert C Martin[1]

This chapter introduces *functions*. Functions are a fundamental building block for code in all programming languages (though they may go by different names, depending on context).

Learning objectives

- You will learn how to write simple functions in Python, using `def` and `return`.
- You will learn how to use functions in your code.
- You will learn the difference between *pure* and *impure functions*, and *side effects*.
- You will learn how arguments are passed to Python functions (pass by assignment).
- You will learn that indentation is syntactically meaningful in Python (unlike many other languages).
- You will learn how to use functions (and constants) in Python's `math` module, such as square root and sine.
- You will expand on and solidify your understanding of topics presented in earlier chapters.

Terms and keywords introduced

- argument
- call or invoke
- `def` keyword
- dot notation

[1] I don't always agree with Uncle Bob, but certainly I agree with this.

- formal parameter
- free variable
- function
- import
- keyword
- lexical scoping
- LEGB (local, enclosing, global, built-in)
- local variable
- module
- pass by assignment
- pure and impure functions
- `return` keyword
- return value
- scope
- shadowing
- side effect
- `UnboundLocalError`

5.1 Introduction to functions

Among the most powerful tools we have as programmers—perhaps the most powerful tools—are *functions*.

We've already seen some built-in Python functions, for example, `print()` and `type()`. We'll see many others soon.

Essentially, a function is a sub-program which we can "call" or "invoke" from within our larger program. For example, consider this code which prints a string to the console.

```
print('Do you want a cookie?')
```

Here we're making use of built-in Python function `print()`. The developers of Python have written this function for you, so you can use it within your program. When we use a function, we say we are "calling" or "invoking" the function.

In the example above, we *call* the `print()` function, supplying the string `'Do you want a cookie?'` as an argument.

As a programmer using this function, you don't need to worry about what goes on "under the hood" (which is quite a bit, actually). How convenient!

When we call a function, the flow of control within our program passes to the function, the function does its work, and then returns a value. All Python functions return a value, though in some cases, the value returned is `None`.[2]

[2] Unlike C or Java, there is no such thing as a *void* function in Python. All Python functions return a value, even if that value is `None`.

Defining a function

Python allows us to define our own functions.[3] A **function** is a unit of code which performs some calculation or some task. A function may take zero or more **arguments** (inputs to the function). The definition of a function may include zero or more **formal parameters** which are, essentially, variables that will take on the values of the arguments provided. When called, the body of the function is executed. In most, but not all cases, a value is explicitly returned. Returned values might be the result of a calculation or some status indicator—this will vary depending on the purpose of the function. If a value is not explicitly returned, the value `None` is returned implicitly.

Let's take the simple example of a function which squares a number. In your mathematics class you might write

$$f(x) = x^2$$

and you would understand that when we apply the function f to some argument x the result is x^2. For example, $f(3) = 9$. Let's write a function in Python which squares the argument supplied:

```
def square(x):
    return x * x
```

`def` is a Python keyword, short for "define", which tells Python we're defining a function. (**Keywords** are reserved words that are part of the syntax of the language. `def` is one such keyword, and we will see others soon.) Functions defined with `def` must have names (a.k.a., "identifiers"),[4] so we give our function the name "square".

Now, in order to calculate the square of something we need to know what that something is. That's where the `x` comes in. We refer to this as a *formal parameter* of the function. When we use this function elsewhere in our code we must supply a value for `x`. Values passed to a function are called *arguments* (however, in casual usage it's not uncommon to hear people use "parameter" and "argument" interchangeably).

At the end of the first line of our definition we add a colon. What follows after the colon is referred to as the *body* of the function. It is within the body of the function that the actual work is done. The body of a function must be indented as shown below—this is required by the syntax of the language. It is important to note that *the body of the function is only executed when the function is called, not when it is defined.*

In this example, we calculate the square, `x * x`, and we *return* the result. `return` is a Python keyword, which does exactly that: it returns some value from a function.

[3] Different languages have different names for sub-programs we can call within a larger program, for example, functions, methods, procedures, subroutines, *etc.*, and some of these designations vary with context. Also, these are defined and implemented somewhat differently in different languages. However, the fundamental idea is similar for all: these are portions of code we can call or invoke within our programs.

[4] Python does allow for anonymous functions, called "lambdas", but that's for another day. For the time being, we'll be defining and calling functions as demonstrated here.

Let's try this in the Python shell to see how it works:

```
>>> def square(x):
...     return x * x
...
>>>
```

Here we've defined the function `square()`. Notice that if we enter this in the shell, after we hit return after the colon, Python replies with `...` and indents for us. This is to indicate that Python expects the body of the function to follow. Remember: The body of a function must be indented. Indentation in Python is syntactically significant (which might seem strange if you've coded in Java, C, C++, Rust, JavaScript, C#, *etc.*; Python uses indentation rather than braces).

So we write the body—in this case, it's just a single line. Again, Python replies with `...`, essentially asking "Is there more?". Here we hit the return/enter key, and Python understands we're done, and we wind up back at the `>>>` prompt.

Now let's use our function by calling it. To call a function, we give the name, and we supply the required argument(s).

```
>>> square(5)   # call `square` with argument 5
25
>>> square(7)   # call `square` with argument 7
49
>>> square(10)  # call `square` with argument 10
100
```

Notice that once we define our function we can reuse it over and over again. This is one of the primary motivations for functions.

Notice also that in this case, there is no `x` outside the body of the function.

```
>>> x
Traceback (most recent call last):
  File "/blah/blah/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
NameError: name 'x' is not defined
```

In this case, `x` exists only within the body of the function.[5] The argument we supply within the parentheses becomes available within the body of the function as `x`. The function then calculates `x * x` and returns the *value* which is the result of this calculation.

If we wish to use the value returned by our function we can save it by assigning the value to some variable, or use it in an expression, or even include it as an argument to another function!

[5] This is what is called "scope", and in the example given `x` exists only within the scope of the function. It does not exist outside the function—for this we say "`x` is out of scope." We'll learn more about scope later.

Can we create new variables in our functions?

Yes. Of course.

```
def cube(x):
    y = x ** 3   # assign result local variable `y`
    return y     # return the value of `y`
```

We refer to such variable names (`y` in this example) as *local variables*, and like the formal parameters of a function, they exist only within the body of the function.

Storing a value returned by a function

Continuing with our example of `square()`:

```
>>> a = 17
>>> b = square(a)
>>> b
289
```

Notice that we can supply a variable as an argument to our function. Notice also that this object needn't be called `x`.[6]

Using the value returned by a function in an expression

Sometimes there's no need for assigning the value returned by a function to a variable. Let's use the value returned by the `square()` function to calculate the circumference of a circle of radius r.

```
>>> PI = 3.1415926
>>> r = 126.1
>>> PI * square(r)
49955.123667046
```

Notice we didn't assign the value returned to a variable first, but rather, we used the result directly in an expression.

Passing the value returned from a function to another function

Similarly, we can pass the value returned from a function to another function.

```
>>> print(square(12))
144
```

[6] In fact, even though it's syntactically valid for a variable in the outer scope to have the same name as a parameter to a function, or a local variable within a function, it's best if they don't have the same identifier. See the section on "shadowing" for more.

What happens here? We pass the value 12 to the `square()` function, this calculates the square and returns the result (144). This result becomes the value we pass to the `print()` function, and, unsurprisingly, Python prints 144.

Do all Python functions return a value?

This is a reasonable question to ask, and the answer is "yes."

But what about `print()`? Well, the point of `print()` is not to return some value but rather to display something in the console. We call things that a function does apart from returning a value *side effects*. The side effect of calling `print()` is that it displays something in the console.

```
>>> print('My hovercraft is full of eels!')
My hovercraft is full of eels!
```

But does `print()` return a value? How would you find out? Can you think of a way you might check this?

What do you think would happen here?

```
>>> mystery = print('Hello')
```

Let's see:

```
>>> mystery = print('Hello')
Hello
>>> print(mystery)
None
```

`None` is Python's special way of saying "no value." `None` is the default value returned by functions which don't otherwise return a value. *All* Python functions return a value, though in some cases that value is `None`.[7] So `print()` returns the `None`.

How do we return `None` (assuming that's something we want to do)? By default, in the absence of any `return` statement, `None` will be returned implicitly.

```
>>> def nothing():
...     pass  # `pass` means "don't do anything"
...
>>> type(nothing())
<class 'NoneType'>
```

Using the keyword `return` without any value will also return `None`.

[7] If you've seen `void` in C++ or Java you have some prior experience with functions that don't return anything. All Python functions return a value, even if that value is `None`.

```
>>> def nothing():
...     return
...
>>> type(nothing())
<class 'NoneType'>
```

Or, if you wish, you can explicitly return `None`.

```
>>> def nothing():
...     return None
...
>>> type(nothing())
<class 'NoneType'>
```

What functions can do for us

- Functions allow us to break a program into smaller, more manageable pieces.
- They make the program easier to debug.
- They make it easier for programmers to work in teams.
- Functions can be efficiently tested.
- Functions can be written once, and used many times.

Comprehension check

1. Write a function which calculates the successor of any integer. That is, given some argument `n` the function should return `n + 1`.
2. What's the difference between a formal parameter and an argument?
3. When is the body of a function executed?

5.2 A deeper dive into functions

Recall that we define a function using the Python keyword `def`. So, for example, if we wanted to implement the following mathematical function in Python:

$$f(x) = x^2 - 1$$

Our function would need to take a single argument and return the result of the calculation.

```
def f(x):
    return x ** 2 - 1
```

We refer to `f` as the *name* or *identifier* of the function. What follows the identifier, within parentheses, are the *formal parameters* of the function.

A function may have zero or more parameters. The function above has one parameter `x`.[8]

Let's take a look at a complete Python program in which this function is defined and then called twice: once with the argument 12 and once with the argument 5.

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

Here we define the function. At this point, x does not have a value, and this code is not executed.

Remember: Writing a function definition does not execute the function. A function is executed *only when it is called.*

Calling a function

Once we have written our function, we may *call* or *invoke* the function by name, supplying the necessary arguments. To call the function `f()` above, we must supply one argument.

```
y = f(12)
```

This calls the function `f()`, with the argument 12 and assigns the result to a variable named `y`. Now what happens?

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

Here we call the function, supplying 12 as an argument.

When we call a function with an argument, the argument is *passed* to the function. The formal parameter receives the argument—that is, *the*

[8] While Python does support optional parameters, we won't present the syntax for this here.

argument is assigned *to the formal parameter*. So when we pass the argument 12 to the function `f()`, then the first thing that happens is that the formal parameter `x` is assigned the argument. It's almost as if we performed the assignment `x = 12` as the first line within the body of the function.

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

When we call the function and pass the argument 12, x takes on the value 12.

Once the formal parameter has been assigned the value of the argument, the function does its work, executing the body of the function.

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

Now, with x = 12, the function evaluates the expression…

…12 × 12 - 1
…144 - 1
…143

Then, the function returns the result. Flow of control is returned *to the point at which the function was called.*

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

The function returns the calculated value (143)…

So f(12) has evaluated to 143, and the result is assigned to y.

For this example, we print the result, 143.

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

Prints 143 at the console.

Let's call the function again, this time with a different argument, 5.

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

Here we call the function again, supplying 5 as an argument.

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

When we call the function and pass the argument 5, x takes on the value 5.

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

Now, with x = 5, the function evaluates the expression…

…5 × 5 - 1
…25 - 1
…24

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

The function returns the calculated value (24)…

So f(5) has evaluated to 24, and the result is assigned to y.

```
"""
Demonstration of a function
"""

def f(x):
    return x ** 2 - 1

if __name__ == '__main__':

    y = f(12)
    print(y)

    y = f(5)
    print(y)
```

Prints 24 at the console.

What to pass to a function?

A function call must match the *signature* of the function. The signature of a function is its identifier and formal parameters. When we call a function, the number of arguments must agree with the number of formal parameters.[9]

[9] Again, we're excluding from consideration functions with optional arguments or keyword arguments.

A function should receive as arguments *all* of the information it needs to do its work. A function should *not* depend on variables that exist only in the outer scope. (In most cases, it's OK for a function to depend on a *constant* defined in the outer scope.)

Here's an example of how things can go wrong if we write a function which depends on some variable that exists in the outer scope.

```
y = 2

def square(x):
    return x ** y

print(square(3))  # prints "9"

y = 3

print(square(3))  # oops! prints "27"
```

This is a great way to introduce bugs and cause headaches. Better that the function should use a value passed in as an argument, or a value assigned within the body of the function (a local variable), or a literal. For example, this is OK:

```
def square(x):
    y = 2
    return x ** y
```

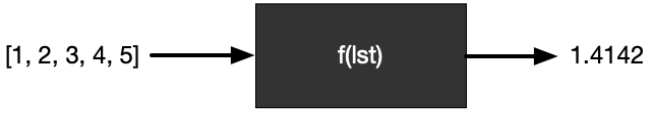
and this is even better:

```
def square(x):
    return x ** 2
```

Now, whenever we supply some particular argument, we'll always get the same, correct return value.

Functions should be "black boxes"

In most cases functions should operate like *black boxes* which take some input (or inputs) and return some output.



We should write functions in such a way that, once written, we don't need to keep track of what's going on within the function in order to use it correctly. A cardinal rule of programming: functions should *hide*

implementation details from the outside world. In fact, **information hiding** is considered a fundamental principle of software design.[10]

For example, let's say I gave you a function which calculates the square root of any real number greater than zero. Should you be required to understand the internal workings of this function in order to use it correctly? Of course not! Imagine if you had to initialize variables used internally by this function in order for it to work correctly! That would make our job as programmers much more complicated and error-prone.

Instead, we write functions that take care of their implementation details internally, without having to rely on code or the existence of variables outside the function body.

5.3 Passing arguments to a function

What happens when we pass arguments to a function in Python? When we call a function and supply an argument, the argument is *assigned* to the corresponding formal parameter. For example:

```
def f(z):
    z = z + 1
    return z

x = 5
y = f(x)

print(x)  # prints 5
print(y)  # prints 6
```

When we called this function supplying `x` as an argument we assigned the value `x` to `z`. It is just as if we wrote `z = x`. This assignment takes place automatically when we call a function and supply an argument.

If we had two (or more) formal parameters it would be no different.

```
def add(a, b):
    return a + b

x = 1
y = 2

print(add(x, y))  # prints 3
```

In this example, we have two formal parameters, `a` and `b`, so when we call the `add` function we must supply two arguments. Here we supply `x` and `y` as arguments, so when the function is called Python automatically makes the assignments `a = x` and `b = y`.

[10] If you're curious, check out David Parnas' seminal 1972 article: "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, 15(12) (https://dl.acm.org/doi/pdf/10.1145/361598.361623).

It would work similarly if we were to supply literals instead of variables.

```
print(add(12, 5))  # prints 17
```

In this example, the assignments that are performed are `a = 12` and `b = 5`.

You may have heard the terms "pass by value" or "pass by reference." These don't really apply to Python. *Python always passes arguments by assignment.* Always.

5.4 Scope

Names of formal parameters and any local variables created within a function have a limited lifetime—they exist only until the function is done with its work. We refer to this as *scope*.

The most important thing to understand here is that names of formal parameters and names of local variables we define within a function have local scope. They have a lifetime limited to the execution of the function, and then those names are gone.

Here's a trivial example.

```
>>> def foo():
...     x = 1
...     return x
...

>>> y = foo()
>>> y
1
```

The name `x` within the function `foo` only exists as long as `foo` is being executed. Once `foo` returns the value of `x`, `x` is no more.

```
>>> def foo():
...     x = 1
...     return x
...

>>> y = foo()
>>> y
1
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

So the name `x` lives only within the execution of `foo`. In this example, the scope of `x` is limited to the function `foo`.

Shadowing

It is, perhaps, unfortunate that Python allows us to use variable names within a function that exist outside the function. This is called *shadowing*, and it sometimes leads to confusion.

Here's an example:

```
>>> def square(x):
...     x = x * x
...     return x
...
>>> x = 5
>>> y = square(x)
>>> y
25
>>> x
5
```

What has happened? Didn't we set `x = x * x`? Shouldn't `x` also be 25?

No. Here we have two different variables with the same name, `x`. We have the `x` in the outer scope, created with the assignment `x = 5`. The `x` within the function `square` is local, within that function. Yes, it has the same name as the `x` in the outer scope, but it's a different `x`.

Generally, it's not a good idea to shadow variable names in a function. Python allows it, but this is more a matter of style and avoiding confusion. Oftentimes, we rename the variables in our functions, appending an underscore.

```
>>> def square(x_):
...     x_ = x_ * x_
...     return x_
...
>>> x = 5
>>> y = square(x)
>>> y
25
```

This is one way to avoid shadowing.

Another approach is to give longer, more descriptive names to variables in the outer scope, and leave the shorter or single-character names to the function. Which approach is best depends on context.

5.5 Pure and impure functions

So far, all the functions we've written are *pure*. That is, they accept some argument or arguments, and return a result, behaving like a black box with no interaction with anything outside the box. Example:

```
def successor(n):
    return n + 1
```

In this case, there's an argument, a simple calculation, and the result is returned. This is pure, in that there's nothing changed outside the function and there's no observable behavior of the function other than returning the result. This is just like the mathematical function

$$s(n) = n + 1.$$

Impure functions

Sometimes it's useful to implement an *impure* function. An impure function is one that has *side effects*. For example, we might want to write a function that prompts the user for an input and then returns the result.

```
def get_price():
    while True:
        price = float(input("Enter the asking price "
                            "for the item you wish "
                            "to sell: $"))
        if price > 1.00:
            break
        else:
            print("Price must be greater than $1.00!")
    return price
```

This is an impure function since it has side effects, the side effects being the prompts and responses displayed to the user. That is, we can observe behavior in this function other than its return value. It *does* return a value, but it exposes other behaviors as well.

Keep side effects to a minimum

Always, *always* consider what side effects your functions have, and whether such side effects are correct and desirable.

As a rule, it's best to keep side effects to a minimum (eliminating them entirely if possible). But sometimes it is appropriate to rely on side effects. Just make sure that if you are relying on side effects, that it is correct and by design, and not due to a defect in programming. That is, if you write a function with side effects, it should be because *you choose to do so* and *you understand how side effects may or may not change the state of your program*. It should always be a conscious choice and never inadvertent, otherwise you may introduce bugs into your program. For example, if you inadvertently mutate a mutable object, you may change the state of your program in ways you have not anticipated, and your program may exhibit unexpected and undesirable behaviors.

We will take this topic up again, when we get to mutable data types like `list` and `dict` in Chapters 10 and 16.

Comprehension check

1. Write an impure function which produces a side effect, but returns `None`.

2. Write a pure function which performs a simple calculation and returns the result.

5.6 The math module

We've seen that Python provides us with quite a few conveniences "right out of the box." Among these are *built-in* functions, that we as programmers can use—and reuse—in our code with little effort. For example, `print()`, and there are many others.

We've also learned about how to define constants in Python. For example, Newton's gravitational constant:

```
G = 6.67 * 10 ** -11  # N m^2 / kg^2
```

Here we'll learn a little about Python's `math` module. The Python `math` module is a collection of constants and functions that you can use in your own programs—and these are very, very convenient. For example, why write your own function to find the principal square root of a number when Python can do it for you?[11]

Unlike built-in functions, in order to use functions (and constants) provided by the Python `math` module, we must first *import* the module (or portions thereof).[12] You can think of this as importing features you need into your program.

Python's `math` module is rich with features. It provides many functions including

function	calculation
`sqrt(x)`	$\sqrt{x}$ (x non-negative)
`exp(x)`	e^x
`log(x)`	$\ln x$ (x positive)
`log2(x)`	$\log_2 x$ (log base two)
`log10(x)`	$\log_{10} x$ (log base ten)
`floor(x)`	$\lfloor x \rfloor$
`ceil(x)`	$\lceil x \rceil$
`sin(x)`	$\sin x$ (x in radians)
`cos(x)`	$\cos x$ (x in radians)
`degrees(x)`	$x \times \frac{180}{\pi}$
`radians(x)`	$x \times \frac{\pi}{180}$

and many others. The `math` module also supplies values for π and e.

Using the math module

To import a module, we use the `import` keyword. Imports should appear in your code immediately after the starting docstring, and you only need

[11] The only reasonable answer to this question is "for pedagogical purposes", and in fact, later on, we'll do just that—write our own function to find the principal square root of a number. But let's set that aside for now.

[12] We're going to ignore the possibility of importing portions of a module for now.

to import a module once. If the import is successful, then the imported module becomes available.

```
>>> import math
```

Now what? Say we'd like to use the `sqrt()` function provided by the `math` module. How do we access it?

If we want to access functions (or constants) within the `math` module we use the `.` operator.

```
>>> import math
>>> math.sqrt(25)
5.0
```

Let's unpack this. Within the `math` module, there's a function named `sqrt()`. Writing `math.sqrt()` is accessing the `sqrt()` function within the `math` module. This uses what is called *dot notation* in Python (and many other languages use this as well).

Let's try another function in the `math` module, `sin()`, which calculates the sine of an angle. You may remember from pre-calculus or trigonometry course that $\sin 0 = 0$, $\sin \frac{\pi}{2} = 1$, $\sin \pi = 0$, $\sin \frac{3\pi}{2} = -1$, and $\sin 2\pi = 0$. Let's try this out.

```
>>> import math
>>> PI = 3.14159
>>> math.sin(0)
0.0
```

So far, so good.

```
>>> math.sin(PI / 2)
0.999999998926914
```

That's close, but not quite right. What went wrong? (Hint: Representation error is *not* the problem.) Our approximation of π (defined as `PI = 3.14159`, above) isn't of sufficient precision. Fortunately, the `math` module includes high-precision constants for π and e.

```
>>> math.sin(math.pi / 2)
1.0
```

Much better.

It is left to the reader to test other arguments to the `sin()` function.

More examples using the `math` module

Here are examples of other functions provided by the `math` module.

Cosine

```
>>> math.cos(0)       # cosine of argument in radians
1.0
math.cos(3.141592653589793)
-1.0
```

Convert radians to degrees

```
>>> math.degrees(math.pi / 2.0)
90.0
>>> math.degrees(3 * math.pi / 4.0)
135.0
```

Convert degrees to radians

```
>>> math.radians(180.0)
3.141592653589793
>>> math.radians(45.0)
0.7853981633974483
```

Floor and ceiling

The *floor* function, written $\lfloor x \rfloor$, returns the greatest integer that's less than or equal to x. This is implemented with `math.floor()`.

```
>>> math.floor(4.2)
4
>>> math.floor(0.0)
0
>>> math.floor(-4.2)
-5
```

Notice that `math.floor()` always returns an object of type `int`. Also notice the result for $\lfloor -4.2 \rfloor$ is -5, because -5 is the greatest integer less than or equal to -4.2.

The *ceiling* function, written $\lceil \rceil$, returns the least integer that's greater than or equal to x. This is implemented with `math.ceil)`.

```
>>> math.ceil(5)
4
>>> math.ceil(0.0)
0
>>> math.ceil(-4.2)
-4
```

Like `math.floor()`, `math.ceil()` always returns an object of type `int`.

Natural logarithm (ln) and exponential function

```
>>> math.log(42)                    # ln of argument
3.7376696182833684
>>> math.exp(3.7376696182833684)    # exp of argument
42.00000000000001
```

Logarithm base 10 and base 2

```
>>> math.log10(1000)  # log base 10 of argument
3.0
>>> math.log2(4096)   # log base 2 of argument
12.0
>>> math.log2(64)
6.0
>>> math.log2(2)
1.0
```

Fun fact: When computer scientists speak of logarithms, they almost always mean log base 2. Can you think why this might be?

π and e

Here are two values provided by the `math` module. You should always use these in your code and never define these as constants yourself.

```
>>> math.e
2.718281828459045
>>> math.pi
3.141592653589793
```

`math` module documentation

As noted, the `math` module has many ready-made functions you can use, including many not covered here. For more information, see the `math` module documentation.

- https://docs.python.org/3/library/math.html.

5.7 Free variables, scope, and LEGB

Python has a behavior that is not universal among programming languages, and if you have experience with, say Java, you might find this surprising.

Let's say we have this function:

```
def f(a):
    return a + y
```

```
x = 5
y = 2

print(f(x))   # prints 7
```

Let's walk through what's happening. Why does this work? Importantly, when is it OK to rely on this behavior and when can it lead to insidious bugs?

In this instance, `y` is what we call a **free variable** within the function `f()`. Notice that `y` is *not* passed in as an argument. What happens here is that Python, seeing the `y` in the body of the function and not finding a matching argument or definition within the function *looks in the enclosing scope to see if it can find the identifier* `y`. It does so, and then uses the value of `y` in the computation and returns the result.

Now consider what happens if we execute this code?

```
def f(a):
    return a + y

x = 5
z = 2   # <-- name changed from y to z

print(f(x))
```

In this instance we get an exception: `NameError: name 'y' is not defined`. This illustrates the danger of free variables: We define a function and it seems OK, but it depends on a name that *must* exist in the outer scope in order for it to work. In this case, not such a good idea. Here's a fix:

```
def f(a, b):     # two formal parameters
    return a + b

x = 5
y = 2

print(f(x, y))   # pass two arguments
```

Now there are no free variables in `f()` and we pass two arguments to the function. Much better—this approach is safer and more predictable.

Now, when is it OK to have a free variable in a function? One case is with *constants*. Let's say we're writing a program to assist with common calculations in physics.

```
G = 9.80665   # m / s^2

def displacement(v_0, t):
    """Calculate distance traveled in free fall in time t """
    return v_0 * t + (1 / 2) * G * t ** 2
```

```
def final_velocity(v_0, t):
    """Calculate velocity at time t """
    return v_0 + G * t

def potential_energy(m, h):
    """Calculate potential energy given mass and height """
    return m * G * h
```

All these calculations require the constant, `G`, acceleration due to gravity (usually denoted g, but here we're indicating it's a constant). It makes sense to define a constant in one place and one place only. You wouldn't want to use 9.80665 in one calculation and 9.8 in another, so we define once and use this constant value as needed.

It's true we could redefine these functions to require `G` as an argument, but there's no need to do that. `G` is indeed free in each of these functions, and Python looks in the enclosing scope to find it. Because `G` is a constant, and because we've defined it *before* defining our functions, this is A-OK.

LEGB: local, enclosing, global, built-in

So how exactly does Python go about finding identifiers, including free variables in functions? Python uses the "LEGB" rule: local, enclosing, global and built-in. When looking for an identifier, Python will first look in the *local* scope. In the case of functions, that's within the body of the function. If it can't find the identifier in the local scope, it looks in the *enclosing* scope. If it can't find the identifier in the enclosing scope, it looks in the *global* scope. If it can't find the identifier in the global scope, it looks among *built-ins*. In the example above, the enclosing scope *is* the global scope, but this isn't always the case.

Here's an example you can experiment with:

```
def f():

    n = 2

    def g():  # yes, we can define functions within functions
        n = 3
        return n

    return g()

n = 1

print(f())
```

Run this code. What does it print? It prints `3`. Why? We call `f()` in `print(f())`. `f()` calls `g()` within its body. `g()` has `n = 3`, so `n` is local to `g()` and `g()` returns `3`, and `f()` returns the value returned by `g()`. The `n` defined in `f()` is ignored, as is the `n` defined in the outer scope. It's important that you understand that *these are three different* `n`*s!*

Now comment out the line `n = 3` within the body of `g()` and run again.

```
def f():

    n = 2

    def g():  # yes, we can define functions within functions
        # n = 3   <-- comment this line
        return n

    return g()

n = 1

print(f())
```

What does it print? It prints `2`. Why? Again, we call `f()` in `print(f())`. `f()` calls `g()` within its body. But now `g()` has no local definition of `n`, so Python looks in the enclosing scope—the body of `f()`—and finds a value there. So `n` within `g()` gets the value `2`, `g()` returns `2`, and `f()` returns the value returned by `g()`.

Now comment out the line `n = 2` within the body of `f()` and run again.

```
def f():

    # n = 2  <-- comment this line too

    def g():  # yes, we can define functions within functions
        # n = 3   <-- comment this line
        return n

    return g()

n = 1

print(f())
```

What does it print? It prints `1`. Why? Again, we call `f()` in `print(f())`. `f()` calls `g()` within its body. `g()` has no local definition of `n`, so Python looks in the enclosing scope—the body of `f()`—and *does not* find a value there either. So Python looks in the global scope, and finds it there: `n = 1`. So the `n` in `g()` gets the value `1`, `g()` returns `1`, and `f()` returns the value returned by `g()`.

It's enough to make your head spin! This is why we must be very careful about free variables!

So remember "LEGB":

- L is for *local*, *e.g.*, inside the current function
- E is for *enclosing*, *e.g.*, within any enclosing function, if nested
- G is for *global*, the top-level module scope
- B is for *built-in*, *e.g.*, identifiers like `len`, `sum`, and `print`

Lexical scoping

The LEGB resolution rule is possible because of what is called **lexical scoping**. When we say a language has lexical scoping it means that the scope of a variable depends on *where the variable is defined* (rather than when a function which uses it is called). The term "lexical" applies to the words or vocabulary of a language, so in the case of lexical scoping we're referring to how code is written and where variables are defined in the code. Without this, the idea of looking in the enclosing scope, for example, wouldn't quite make sense.

`IndentationError`

In Python, unlike many other languages, indentation is syntactically significant. We've seen when defining a function that the body of the function must be indented (we'll see other uses of indentation soon).

An exception of type `IndentationError` is raised if Python disagrees with your use of indentation—typically if indentation is expected and your code isn't, or if a portion of your code is over-indented. `IndentationError` is a more specific type of `SyntaxError`.

When you encounter an indentation error, you should inspect your code and correct the indentation.

Here's an example:

```
>>> def square(n):
... return n * n
  File "<stdin>", line 2
    return n * n
    ^
IndentationError: expected an indented block after function
                        definition on line 1
>>> def square(n):
...     return n * n  # now it's correctly indented!
...
>>> square(4)
16
```

`ValueError`

A `ValueError` is raised when the *type* of an argument or operand is valid, but the *value* is somehow unsuitable. We've seen how to import the `math` module and how to use `math.sqrt()` to calculate the square root of some number. However, `math.sqrt()` does not accept negative operands (it

doesn't know about complex numbers), and so, if you supply a negative operand to `math.sqrt()` a `ValueError` is raised.

Example:

```
>>> import math
>>> math.sqrt(4)  # this is A-OK
2.0
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

In a case like this, you need to ensure that the argument or operand which is causing the problem has a suitable value.

`ModuleNotFoundError`

We encounter `ModuleNotFoundError` if we try to import a module that doesn't exist, that Python can't find, or if we misspell the name of a module.

Example:

```
>>> import maath
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'maath'
```

`UnboundLocalError`

An *unbound local error* is a more specific kind of `NameError`—one which occurs within a function or method. This type of error is raised when we try to read a value from a variable before the variable has been assigned a value within the body (scope) of a function.

Example:

```
>>> def f(x):
    y = y + 1
    return x + y

>>>
>>> f(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'y' referenced before assignment
```

5.8 Exercises

Exercise 01

Identify the formal parameter(s) in each of the following functions.

a.

```
def add_one(n):
    n = n + 1
    return n
```

b.

```
def power(x, y):
    return x ** y
```

Exercise 02

There's something wrong with each of these function definitions. Identify the problem and suggest a fix.

a. Function to cube any real number.

```
def cube:
    return x ** 3
```

b. Function to print someone's name.

```
def say_hello():
    print(name)
```

c. Function to calculate $x^2 + 3x - 1$ for any real valued x.

```
def poly(x):
return x ** 2 + 3 * x - 1
```

d. Function which takes some number, x, subtracts 1, and returns the result.

```
def subtract_one(x):
    y = x - 1
```

Exercise 03

Write a function which takes any arbitrary string as an argument and prints the string to the console.

Exercise 04

Write a function which takes two numeric arguments (`float` or `int`) and returns their product.

Exercise 05

a. Write a function which take an integer as an argument and returns `0` if the integer is even and `1` if the integer is odd. Hint: The remainder (modulo) operator, `%`, calculates the remainder when performing integer division. For example, `17 % 5` yields 2, because 5 goes into 17 three times, leaving a remainder of two.

b. What did you name your function and why?

Exercise 06

Write a function which takes two numeric arguments, one named `subtotal` and the other named `tax_rate`, and calculates and *returns* the total including tax. For example, if the arguments supplied were `114.0` for `subtotal` and `0.05` for `tax_rate`, your function should return the value `119.7`. If the arguments were `328.0` and `0.045`, your function should return the value `342.76`.

This function should produce no side effects.

Exercise 07

Recall the Law of Sines. For any triangle with sides of lengths a, b, and c, with opposite angles α, β, and γ, respectively, we have

$$\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{a}{\sin\gamma}$$

Here's a program with a bug. This is intended to use the Law of Sines to solve for the length of a side of an arbitrary triangle, given the length of a side, its opposite angle, and the opposite angle of the side whose length we wish to find (SAA). Find and fix the defect.

```
import math

def saa(a, alpha_deg, beta_deg):
    """Given side a and opposite angles alpha and beta (in degrees),
    return side b using the Law of Sines."""
    alpha_rad = math.radians(alpha_deg)
    beta_rad = radians(beta_deg)
    return a * math.sin(beta_rad) / math.sin(alpha_rad)

a_length = 22
alpha = 107
beta = 45
b = saa(a_length, alpha, beta)
print("The length of side b is " + str(b))
```

Exercise 08

Each of the following functions has at least one free variable, which is not a constant (danger!). Identify the free variables in each instance.

```
def f(a, b):
    a = a ** 2
    b = b ** 2
    return (a + b) / c
```

```
def g(x):
    return a * x ** 2 + b * x + c
```

```
def deg_to_rad(d):
    return deg * (180 / pi)
```

Exercise 09

This code fails when run. It results in the exception: `RecursionError: maximum recursion depth exceeded`.

```
def print_x(x):
    print_x(x)

print_x(5)
```

Describe in words what you think happens when we call this function.

Exercise 10

When this code is run it prints `139` and then crashes with the exception `TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'`. What's wrong here and how would you fix it?

```
def calc_poly(x):
    y = 2 * x ** 3 - 5 * x ** 2 + 3 * x - 1
    print(y)

y = calc_poly(5)
y = y + 1
```

Chapter 6

Style

> Programs must be written for people to read, and only incidentally for machines to execute.
>
> –Abelson, Sussman, and Sussman

> Learn the rules so you know how to break them.
>
> –The 14th Dalai Lama

This chapter will introduce the concept of good Python style through the use of the PEP 8 style guide. We'll further our understanding of constants, learn about the benefits of using comments, and how/when to use them effectively.

Learning objectives

- You will learn about the importance of good Python style and the PEP 8 style guide.
- You will learn conventions for naming constants.
- You will learn about comments and docstrings in Python and their uses.
- You will learn how to deal with long lines using implicit concatenation and implicit line continuation.

Terms introduced

- `camelCase`
- docstring
- implicit concatenation
- implicit line continuation
- inline comment
- PEP 8
- single-line comment
- `snake_case`

6.1 The importance of style

As we learn to write programs, we will also learn how to use good style. Good coding style is more important than many beginners realize. Using good style:

- helps you read your code more quickly and accurately.
- makes it easier to identify syntax errors and common bugs.
- helps clarify your reasoning about your code.

Most workplaces and open-source projects require conformance to a coding standard. For example, *Google's style guide for Python.*[1]

When we're solving problems we don't want reading code or making decisions about formatting our code to consume more than their share of valuable cognitive load. So start off on the right foot and use good style from the beginning. That way, you can free your brain to solve problems—that's the fun part.

Every language has its conventions, and you should stick to them. This is part of writing readable, idiomatic code.

PEP 8

Fortunately, there is a long-standing, comprehensive style guide for Python called PEP 8.[2] The entire style guide *PEP 8 – Style Guide for Python Code* is available online[3]. Noted Python developer Kenneth Reitz has made a somewhat prettified version at https://pep8.org/. You should consult these resources for a complete guide to good style for Python. Many projects in industry and in the open source world enforce the use of this standard.

Don't think in terms of saving keystrokes—follow the style guide, and you'll find this helps you (and others!) read and understand what you write.

Whitespace

Yes! Whitespace is important! Using whitespace correctly can make your code *much* more readable.

Use whitespace between operators and after commas.[4] Example:

```
# Don't do this

x=3.4*y+9-17/2

joules=(kg*m**2)/(sec**2)

amount=priceperitem*items
```

[1] https://google.github.io/styleguide/pyguide.html
[2] PEP is short for Python Enhancement Proposal
[3] https://peps.python.org/pep-0008/
[4] One exception to this rule is keyword arguments in function calls, for example, the `end` keyword argument for the built-in `print()` function.

Instead, do this:

```
# Better

x = 3.4 * y + 9 - 17 / 2

joules = (kg * m ** 2) / (sec ** 2)

amount = price_per_item * items
```

Without the whitespace between operators, your eye and brain have to do much more work to divide these up for reading.

Sometimes, however, extra whitespace is unnecessary or may even hinder readability. You should avoid unnecessary whitespace before closing and after opening braces, brackets or parentheses.

```
# Don't do this

picas = inches_to_picas ( inches )

# Better

picas = inches_to_picas(inches)
```

Do not put whitespace after names of functions.

```
# Don't do this

def inches_to_points (in):
    return in * POINTS_PER_INCH

# Better

def inches_to_points(in):
    return in * POINTS_PER_INCH
```

In doing this, we create a tight visual association between the name of a function and its parameters.

Do not put whitespace before commas or colons.

```
# Don't do this

lst = [3  ,  2  ,  1]

# Better
lst = [3, 2, 1]
```

It's OK to slip a blank line within a block of code to logically separate elements but don't use more than one blank line. Exceptions to this are

function declarations which should be preceded by two blank lines, and function bodies which should be followed by two blank lines.

Names (identifiers)

Choosing good names is crucial in producing readable code. Use meaningful names when defining variables, constants, and functions. This makes your code easier to read and easier to debug!

Examples of good variable names:

- `velocity`
- `average_score`
- `watts`

Examples of bad variable names:

- `q`
- `m7`
- `rsolln`

Compare these with the good names (above). With a good name, you know what the variable represents. With these bad names, who knows what they mean? (Seriously, what the heck is `rsolln`?)

There are some particularly bad names that should be avoided no matter what. Never use the letter "O" (upper or lower case) or "l" (lowercase "L") or "I" (upper case "i") as a variable name. "O" and "o" look too much like "0". "l" and "I" look too much like "1".

While single letter variable names are, in general, to be avoided, it's OK sometimes (depending on context). For example, it's common practice to use `i`, `j`, *etc.* for loop indices (we'll get to loops later) and `x`, `y`, `z` for spatial coordinates, but only use such short names when it is 100% clear from context what these represent.

Similar rules apply to functions. Examples of bad function names:

- `i2m()`
- `sd()`

Examples of good function names:

- `inches_to_meters()`
- `standard_deviation()`

ALL_CAPS, lowercase, snake_case, camelCase, WordCaps

In Python, the convention for naming variables and function is that they should use lowercase or so-called *snake case*, in which words are separated by underscores.

In some other languages, *camelCase* (with capital letters in the middle) or *WordCaps* (where each word is capitalized) are the norm. Not so with Python. *camelCase* should be avoided (always). *WordCaps* are appropriate for class names (a feature of object-oriented programming—something that's not presented in this text).

- Good: `price_per_item`
- Bad: `Priceperitem` or `pricePerItem`

As noted earlier, ALL_CAPS is reserved for constants.

Line length

PEP 8 suggests that lines should not exceed 79 characters in length. There are several reasons why this is a good practice.

- It makes it feasible to print source code on paper or to view without truncation on various source code hosting websites (for example, GitHub, GitLab, *etc.*).
- It accommodates viewports (editor windows, *etc.*) of varying width (don't forget you may be collaborating with others).
- Even if you have a wide monitor, and can fit long lines in your viewport, long lines *slow your reading down*. This is well documented. If your eye has to scan too great a distance to find the beginning of the next line, readability suffers.

See: *Dealing with long lines* in this chapter.

Using underscore in large numeric literals

Python permits the use of underscores in numeric literals. Using underscores can help make large numeric literals easier to read.

```
x = 11111111111   # without underscores -- is this billions?

x = 11_111_111_111  # now we can see immediately
```

This is legal, if perhaps a bit peculiar, within the fractional part of float literals.

```
x = 1.234_567_89
```

In both cases, Python ignores the underscores—they're for human eyes!

```
>>> 11_111_111_111
11111111111
>>> 1.234_567_89
1.23456789
```

6.2 Constants

In most programming languages there's a convention for naming constants. Python is no different—and the convention is quite similar to many other languages.

In Python, we use ALL_CAPS for constant names, with underscores to separate words if necessary. Here are some examples:

```
# Physical constants
C = 299792458  # speed of light: meters / second ** -1
MASS_ELECTRON = 9.1093837015 * 10 ** -31  # mass in kg

# Mathematical constants
PI = 3.1415926535  # pi
PHI = 1.6180339887  # phi (golden ratio)

# Unit conversions
FEET_PER_METER = 3.280839895
KM_PER_NAUTICAL_MILES = 1.852

# Other constants
EGGS_PER_CARTON = 12
```

Unlike Java, there is no `final` keyword, which tells the compiler that a constant must not be changed (same for other languages like C++ or Rust which have a `const` keyword).

What prevents a user from changing a constant? In Python, nothing. All the more reason to make it immediately clear—visually—that we're dealing with a constant.

So the rule in Python is to use ALL_CAPS for constants and nothing else. Then it's up to you, the programmer, to ensure these remain unchanged.

6.3 Comments in code

Virtually all programming languages allow programmers to add *comments* to their code, and Python is no different. Comments are text within your code which is ignored by the Python interpreter.

Comments have many uses:

- explanations as to *why* a portion of code was written the way it was,
- reminders to the programmer, and
- guideposts for others who might read your code.

Comments are an essential part of your code. In fact, it's helpful to think of your comments as you do your code. By that, I mean that *the comments you supply should be of value to the reader*—even if that reader is *you*.

Some folks say that code should explain *how*, and comments should explain *why*. This is not always the case, but it's a very good rule in general. But beware: good comments cannot make up for opaque or poorly-written code.

Python also has what are called *docstrings*. While these are not ignored by the Python interpreter, it's OK for the purposes of this textbook to think of them that way.

Docstrings are used for:

- providing identifying information,
- indicating the purpose and correct use of your code, and
- providing detailed information about the inputs to and output from functions.

That said, here are some forms and guidelines for comments and docstrings.

Inline and single-line comments

The simplest comment is an *inline* or *single-line* comment. Python uses the `#` (call it what you will—pound sign, hash sign, number sign, or octothorpe) to start a comment. Everything following the `#` on the same line will be ignored by the Python interpreter. Here are some examples:

```
# This is a single-line comment

foo = 'bar'  # This is an inline comment
```

Docstrings

Docstring is short for *documentation string*. These are somewhat different from comments. According to PEP 257

> A *docstring* is a string literal that occurs as the first statement in a module, function, class, or method definition.

Docstrings are not ignored by the Python interpreter, but for the purposes of this textbook you may think of them that way. Docstrings are delimited with triple quotation marks. Docstrings may be single lines, thus:

```
def square(n):
    """Return the square of n."""
    return n * n
```

or they may span multiple lines:

```
"""
Egbert Porcupine <egbert.porcupine@uvm.edu>
CS 1210, section Z
Homework 5
"""
```

It's a good idea to include a docstring in every program file to explain who you are and what your code is intended to do.

```
"""
Distance converter
J. Jones

This is a simple program that
converts miles to kilometers.
We use the constant KM_PER_MILE
= 1.60934 for these calculations.
"""
```

Using comments as scaffolding

You may find it helpful to use comments as *scaffolding* for your code. This involves using temporary comments that serve as placeholders or outlines of your program. In computer science, a description of steps written in plain language embedded in code is known as *pseudocode*.

For example, if one were asked to write a program that prompts the user for two integers and then prints out the sum, one might sketch this out with comments, and then replace the comments with code. For example:

```
# Get first integer from user
# Get second integer from user
# Calculate the sum
# Display the result
```

and then, implementing the code one line at a time:

```
a = int(input('Please enter an integer: '))
# Get second integer from user
# Calculate the sum
# Display the result
```

```
a = int(input('Please enter an integer: '))
b = int(input('Please enter another integer: '))
# Calculate the sum
# Display the result
```

```
a = int(input('Please enter an integer: '))
b = int(input('Please enter another integer: '))
result = a + b
# Display the result
```

```
a = int(input('Please enter an integer: '))
b = int(input('Please enter another integer: '))
result = a + b
```

```
print(f'The sum of the two numbers is {result}')
```

This approach allows you to design your program initially without fussing with syntax or implementation details, and then, once you have the outline sketched out in comments, you can focus on the details one step at a time.

TODOs and reminders

While you are writing code it's often helpful to leave notes for yourself (or others working on the same code). `TODO` is commonly used to indicate a part of your code which has been left unfinished. Many IDEs recognize `TODO` and can automatically generate a list of unfinished to-do items.

Avoid over-commenting

While it is good practice to include comments in your code, well-written code often does not require much by way of comments. Accordingly, it's important not to over-comment your code. Here are some examples of over-commenting:

```
song.set_tempo(120)  # set tempo to 120 beats / minute  NO!

x = x + 1  # add one to x  NO!

# I wrote this code before I had any coffee  NO!
```

> **Note**
>
> It is often the case that code from textbooks or presented in lectures is over-commented. This is for pedagogical purposes and should be understood as such.

6.4 Dealing with long lines

PEP 8 requires lines that do not exceed 80 characters. Sometimes, in production-grade code we exceed this, but when we do so, we do so with good reason. You should get in the habit of working within this constraint, as long lines become hard to read. Here are some things you can do if you have long lines.

Implicit concatenation

Strings can be broken up quite nicely using implicit concatenation. Here's how implicit concatenation works:

```
>>> "Implicit " "concatenation"
'Implicit concatenation'
```

Look at that. Python concatenated them without the "+" operator. We can use this to deal with long strings.

```
print("Here I'm using implicit concatenation to deal with "
      "what otherwise might be an annoyingly long line "
      "which if all on one line would certainly exceed "
      "eighty characters, but here it does not. Yipee!")
```

This works with f-strings too. You only need to prefix the strings that have replacement fields.

```
name = "Egbert Porcupine"
species = "Erethizon dorsatum"
birth_year = 1960
home = "Porcupine Mountains Wilderness State Park"
current_year = 2025

print(f"{{name}} was born in {birth_year} which makes him "
      f"{current_year - birth_year} years old. This is "
      "an unusual age for a member of the species "
      f"{{species}} and it's quite likely that {{name}} is "
      "the world's oldest living porcupine. He hails "
      f"from {{home}}, one of the lovliest, and most "
      "unspoiled wildernesses in North America.")
```

Implicit line continuation by wrapping in parentheses

When we have a long expression we can wrap the entire thing in parentheses.

```
x = 3
# Wrap a long expression in parentheses!
y = (3 * x ** 14 + 4 * x ** 13 - 17 * x ** 12
    + 5 * x ** 11 - 22 * x ** 10 + 9 ** x * 9
    - 42 * x ** 8 + 2 * x ** 7 - 401 * x ** 6
    + 2 * x ** 5 + 3 * x ** 4 - 15 * x ** 3
    - 10 * x ** 2 + 6 * x - 7)
print(y)
```

The line continuation character

Python does have a line continuation character, the backslash (\). Using the backslash to explicitly continue a line is discouraged and use of implicit line continuation (within parentheses, for example) is much preferred.

Here's one example of a case where \ makes sense, just to satisfy your curiosity:

```
assert some_long_variable_name > another_long_variable_name \
    and some_function(x, y) != 42, \
    "Computation failed: result was 42 and value of \
    another_long_variable_name is too small"
```

(You won't see many examples in this book, though it is used here and there in code snippets in order to fit code within the width of the page and where using other approaches might be confusing).

6.5 Exercises

Exercise 01

Here are some awful identifiers. Replace them with better ones.

```
# Newton's gravitational constant, G
MY_CONSTANT = 6.674E-11

# Circumference of a circle
circle = rad ** 2 * math.pi

# Clock arithmetic
# Calculate 17 hours after 7 o'clock
thisIsHowWeDoItInJava = (7 + 17) % 12
```

Exercise 02

The following Python code runs perfectly fine but deviates from the PEP 8 style guide. Using your chosen IDE, fix the issues and check to make sure that the program still runs correctly.

```
def CIRCUMFERENCEOFCIRCLE(radius):
    c=2*pi*radius
    return c
pi=3.14159
CIRC=CIRCUMFERENCEOFCIRCLE(22)
print("The circumference of a circle with a radius of 22cm"
"is "+str(CIRC)+ "cm.")
```

Exercise 03

Here's the first paragraph of Edwin A. Abbot's *Flatland: A Romance of Many Dimensions.*[5]

[5] https://www.gutenberg.org/cache/epub/201/pg201.txt

> If my poor Flatland friend retained the vigour of mind which he enjoyed when he began to compose these Memoirs, I should not now need to represent him in this preface, in which he desires, firstly, to return his thanks to his readers and critics in Spaceland, whose appreciation has, with unexpected celerity, required a second edition of his work; secondly, to apologize for certain errors and misprints (for which, however, he is not entirely responsible); and, thirdly, to explain one or two misconceptions. But he is not the Square he once was. Years of imprisonment, and the still heavier burden of general incredulity and mockery, have combined with the natural decay of old age to erase from his mind many of the thoughts and notions, and much also of the terminology, which he acquired during his short stay in Spaceland. He has, therefore, requested me to reply in his behalf to two special objections, one of an intellectual, the other of a moral nature.

Without using a triple-quoted string, render this in Python without exceeding 79 characters on any given line.

Exercise 04

Render the following polynomial expression in Python without exceeding 79 characters on any given line.

$$2047x^7 - 1905x^6 + 1729x^5 - 1387x^4 + 1105x^3 - 645x^2 + 561x - 341$$

Exercise 05

Rewrite each of the following to conform to PEP 8.

a. `Hypotenuse=math.sqrt(a**2+b**2)`

b. `print ( "This lovely string" )`

c. `distance = v0+math.cos( Theta) *t`

d. `Torque = r * Force*   math.sin(theta)`

Chapter 7

Console I/O

So far, we've provided all the values for our functions in our code. Things get a lot more interesting when the user can provide such values.

In this chapter, we'll learn how to get input from the user and use it in our calculations. We'll also learn how to format the output displayed to the user.

We call getting and displaying data this way as *console I/O* ("I/O" is just short for input/output).

We'll also learn how to use Python's *f-strings* (short for *formatted string literals*) to format output. For example, we can use f-strings with *format specifiers* to display floating point numbers to a specific number of digits to the right of the decimal place. With f-strings we can align strings for displaying data in tabular format.

> ⚠ Warning
>
> In this text, we will use f-strings exclusively for formatting output. Beware! There's a lot of stale information out there on the internet showing how to format strings in Python. For example, you may see the so-called *printf-style* (inherited from the C programming language), or the `str.format()` method. These alternate methods have their occasional uses, but for general purpose string formatting, your default should be to use f-strings.

Learning objectives

- You will learn how to prompt the user for input, and handle input from the user, converting it to an appropriate type if necessary.
- You will learn how to format strings using f-strings and interpolation.
- You will learn how to use format specifiers within f-strings.
- You will write programs which receive user input, and produce output based on that input, often performing calculations.

Terms and built-in functions introduced

- command line interface (CLI)
- console
- constructor (`int()`, `float()`, and `str()`)
- f-string
- format specifier
- graphical user interface (GUI)
- I/O (input/output)
- `input()`
- replacement field
- string interpolation

7.1 Motivation

It's often the case that as we are writing code, we don't have all the information we need for our program to produce the desired result. For example, imagine you were asked to write a calculator program. No doubt, such a program would be expected to add, subtract, multiply, and divide. But *what* should it add? *What* should it multiply? You, as the programmer, would not know in advance. Thus, you'd need a way to get information from the user *into* the program.

Of course, there are many ways to get input from a user. The most common, perhaps, is via a *graphical user interface* or *GUI*. Most, or likely all, of the software you use on your laptop, desktop, tablet, or phone makes use of a GUI.

In this chapter, we'll see how to get input in the most simple way, without having to construct a GUI. Here we'll introduce getting user input from the console. Later, in Chapter 13, we'll learn how to read data from an external file.

7.2 Command line interface

We've seen how to use the Python shell, where we type expressions or other Python code and it's executed interactively.

What we'll learn now is how to write what are called *CLI* programs. That's short for *command line interface*. This distinguishes them from GUI or graphical user interface programs.

When we interact with a CLI program, we run it from the command line (or within your IDE) and we enter data and view program output in text format. We often refer to this interaction as taking place within the *console*. The console is just a window where we receive text prompts, and reply by typing at the keyboard.

This has a similar feel to the Python shell: prompt then reply, prompt then reply.

7.3 The `input()` function

Python makes it relatively easy to get input from the console, using the built-in function `input()`. The `input()` function takes a single, optional

parameter—a string—which, if supplied, is used as a prompt displayed to the user.

Here's a quick example at the Python shell:

```
>>> input("What is your name? ")
What is your name? Sir Robin of Camelot
'Sir Robin of Camelot'
>>> input("What is your favorite color? ")
What is your favorite color? Blue
'Blue'
>>> input("What is the capital of Assyria? ")
What is the capital of Assyria? I don't know that!
"I don't know that!"
```

`input()` takes a string as an argument. This string is displayed as a prompt to the user, like "How old are you?" or "How many cookies would you like to bake?" or "How long are your skis (in cm)?" After displaying a prompt, `input()` waits for the user to enter something at the keyboard. When the user hits the return key, `input()` returns what the user typed as a string.

Again, here's an example in the Python shell—notice we're going to store the value returned by the `input()` function using a variable.

```
>>> users_name = input("What is your name? ")
What is your name? Sir Robin of Camelot
>>> users_name
'Sir Robin of Camelot'
```

Try this out on your own in the Python shell.

Here's what just happened. On the first line (above) we called the `input()` function and supplied the string argument "What is your name? ". Then, on the second line, `input()` does its work. It prints "What is your name?" and then waits for the user to type their response. In this case, the user has typed "Sir Robin of Camelot". When the user hits the enter/return key, the `input()` function returns what the user typed (before hitting enter/return) as a string. In this example, the string returned by `input()` is assigned the name `users_name`. On the third line, we enter the expression `users_name` and Python obliges by printing the associated value: "Sir Robin of Camelot".

Here's a short program that prompts the user for their name and their quest, and just echoes back what the user has typed:

```
"""Prompts the user for their name and quest
and prints the results. """

name = input('What is your name? ')
quest = input('What is your quest? ')

print(name)
print(quest)
```

Try it out. Copy this code, paste it into an editor window in your text editor or IDE, and save the file as `name_and_quest.py`. Then run the program. When run, the program first will prompt the user with 'What is your name? ' and then it will assign the value returned by `input()` to the variable `name`. Then it will prompt for the user's quest and handle the result similarly, assigning the result to the variable `quest`. Once the program has gathered the information it needs, it prints the name and quest that the user provided.

A session for this might look like this:

```
What is your name? Galahad
What is your quest? To seek the Holy Grail!
Galahad
To seek the Holy Grail!
```

Notice that in order to use the strings returned by `input()` we assigned them to variables. Again, remember that the `input()` function returns a string.

That's pretty convenient, when what we want are strings, but sometimes we want numeric data, so we'll also learn how to convert strings to numeric data (where possible) with the functions `int()` and `float()`.

7.4 Converting strings to numeric types

The problem

Here's an example which illustrates a problem we encounter when trying to get numeric input from the user—the `input()` function *always* returns a string and we can't do math with strings.

```
"""
Prompts the user for weight in kilograms
and converts to (US customary) pounds.
"""

POUNDS_PER_KILOGRAM = 2.204623


def kg_to_pounds(kg_):
    return kg_ * POUNDS_PER_KILOGRAM
```

```
kg = input("Enter the weight in kilograms (kg): ")
lbs = kg_to_pounds(kg)
print(lbs)
```

If we save this code to file and try running it, it will fail when trying to convert kilograms to pounds. We'll get the message:

```
TypeError: can't multiply sequence by non-int of type 'float'
```

What happened? When the program gets input from the user:

```
kg = input("Enter the weight in kilograms (kg): ")
```

the value returned from the `input()` function is a string. The value returned from the `input()` function is *always* a string. Say the user enters "82" at the prompt. Then what gets saved with the name `kg` is the *string* `'82'` not the number `82` and we can't multiply a string by a floating point number—that makes no sense!

Happily, there's an easy fix. Python provides built-in functions that can be used to produce objects of numeric types from a string (if possible). These are the integer *constructor*, `int()`, and the float *constructor*, `float()`. These functions can take a string which *looks* like it ought to be convertible to a number, performs the conversion, and returns an object of the corresponding numeric type.

Let's try these out in the Python shell:

```
>>> int('82')
82
>>> float('82')
82.0
```

In the first instance, we pass the string `'82'` as an argument to the `int()` constructor to get an `int`. In the second instance, we pass the string `'82'` as an argument to the `float()` constructor to get a `float`.

What happens if have a string like `'82.5'`? This is a valid input to the `float()` constructor, but does not work with the `int()` constructor.

```
>>> float('82.5')  # this works OK
82.5
>>> int('82.5')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '82.5'
```

The error message is telling us that the string `'82.5'` cannot be converted to an `int`.

Now, returning to the problem at hand—converting user input to a floating point number—here's how we fix the code we started with.

```
"""
Prompts the user for weight in kilograms
and converts to (US customary) pounds.
"""

POUNDS_PER_KILOGRAM = 2.204623


def kg_to_pounds(kg_):
    return kg_ * POUNDS_PER_KILOGRAM


kg = float(input("Enter the weight in kilograms (kg): "))
lbs = kg_to_pounds(kg)
print(lbs)
```

Notice that we wrapped the call to `input()` within a call to the `float` constructor. This expression is evaluated from the inside out (as you might suspect). First the call to `input()` displays the prompt provided, waits for input, then returns a string. The value returned (say, `'82.5'`) is then passed to the `float` constructor as an argument. The `float` constructor does its work and the constructor returns a floating point number, `82.5`. Now, when we pass this value to the function `kg_to_pounds()`, everything works just fine.

If you've seen mathematical functions before this is no different from something like

$$f(g(x))$$

where we would first calculate $g(x)$ and then apply f to the result.

Another scenario with a nasty bug

Consider this program which has a nasty bug:

```
"""This program has a bug!
It does not add as you might expect. """

a = input("Enter an integer: ")
b = input("Enter another integer: ")
print(a + b)
```

Can you see what the bug is?

Imagine what would happen if at the first prompt the user typed "42" and at the second prompt the user typed "10". Of course, 42 plus 10 equals 52, but is that what this program would print?

No. Here's a trial run of this program:

```
Enter an integer: 42
Enter another integer: 10
4210
```

"4210" is not the correct result! What happened?

Remember, `input()` always returns a *string*, so in the program above, `a` is a string and `b` is a string. Thus, when we perform the operation `a + b` it's *not* addition, it's *string concatenation!*

What do we do in cases like this? In order to perform arithmetic with user-supplied values from `input()`, we first need to convert input strings to numeric types (as above).

```
"""This program fixes the bug in the earlier program."""

a = input("Enter an integer: ")
b = input("Enter another integer: ")
a = int(a)
b = int(b)
print(a + b)
```

We can make this a little more concise:

```
"""Prompt the user for two integers and display the sum."""

a = int(input("Enter an integer: "))
b = int(input("Enter another integer: "))
print(a + b)
```

Conversion may fail

While the code samples above work fine if the user follows instructions and enters numeric strings that can be converted to integers, users don't always read instructions and they aren't always well-behaved. For example, if a misbehaved user were to enter values that cannot be converted to integers, we might see a session like this:

```
Enter an integer: cheese
Enter another integer: bananas
Traceback (most recent call last):
  File "/myfiles/addition_fixed.py", line 5, in <module>
    a = int(a)
ValueError: invalid literal for int() with base 10: 'cheese'

Process finished with exit code 1
```

This occurs because `'cheese'` cannot be converted to an `int`. In this case, Python reports a `ValueError` and indicates the invalid literal `'cheese'`. We'll see how to handle problems like this later on in Chapter 15.

`input()` does not validate input

It's important to note that `input()` *does not validate the user's input.*

Validation is a process whereby we check to ensure that input from a user or some other source meets certain criteria. That's a *big* topic we'll touch on later, but for now, just keep in mind that the user can type just about anything at a prompt and `input()` will return whatever the user typed—without checking anything.

So a different session with the same program (above) might be

```
What is your name? -1
-1
```

Be warned.

Don't use names that collide with names of built-in Python functions!

As noted, `input`, `int`, and `float` are names of built-in Python functions. It's very important that you do not use such names as names for your own functions or variables. In doing so, for example, you'd be reassigning the name `input`, and thus the `input()` function would no longer be accessible. For example, this

```
input = input('What is your name? ')
print(input)  # so far, so good -- prints what the user typed
input = input('What is your quest? ')
print(input)
```

fails miserably, with

```
Traceback (most recent call last):
  File ".../3.10/lib/python3.10/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: 'str' object is not callable
```

What happened? We assigned the result to an object named `input`, so after the first line (above) is executed, `input` no longer refers to the function, but instead is now a string (hence the error message " 'str' object is not callable").

So be careful to choose good names and avoid collisions with built-ins.

Additional resources

The documentation for any programming language can be a bit technical. But it can't hurt to take a look at the documentation for `input()`, `int()`, and `float()`. If it makes your head spin, just navigate away and move on. But maybe the documentation can deepen your understanding of these functions. See relevant sections of:

- https://docs.python.org/3/library/functions.html

7.5 Some ways to format output

Say we want to write a program which prompts a user for some number and calculates the square of that number. Here's a program that does just that:

```
"""A program to square a number provided by the user
and display the result. """

x = input("Enter a number: ")
x = float(x)
result = x * x
print(result)
```

That's fine, but perhaps we could make this more friendly. Let's say we wanted to print the result like this.

```
17.0 squared is 289.0
```

How would we go about it? There are a few ways. One somewhat clunky approach would be to use string concatenation. Now, we cannot do this:

```
"""A program to square a number provided by the user
and display the result. """

x = input("Enter a number: ")
x = float(x)
result = x * x
print(x + ' squared is ' + result)
```

This program fails, with the error:

```
Traceback (most recent call last):
  File ".../squared_with_concatenation.py", line 4, in <module>
    print(x + ' squared is ' + result)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Why? Because we cannot concatenate floats and strings. One way to fix this would be to explicitly convert the floating-point values to strings. We can do this—explicitly—by using Python's built-in function `str()`.

```
"""A program to square a number provided by the user
and display the result. """

x = input("Enter a number: ")
x = float(x)
result = x * x
print(str(x) + ' squared is ' + str(result))
```

Now, this works. Here's a trial.

```
Enter a number: 17
17.0 squared is 289.0
```

Again, this works, but it's not a particularly elegant solution. If you were to think "there must be a better way" you'd be right!

7.6 Python f-strings and string interpolation

The approach described above is valid Python, but there's a better way. Earlier versions of Python offered a form of *string interpolation* borrowed from the C programming language and the string `format()` function (neither of which are presented here). These are still available, but are largely superseded. With Python 3.6 came *f-strings*.[1]

f-strings provide an improved approach to string interpolation. They are, essentially, template strings with placeholders for values we wish to interpolate into the string.

An f-string is prefixed with the letter `f`, thus:

```
f'I am an f-string, albeit a boring one.'
```

The prefix tells the Python interpreter that this is an f-string.

Within the f-string, we can include names of objects or expressions we wish to interpolate within curly braces (these are called *replacement fields*). For example,

```
>>> name = 'Carol'
>>> f'My name is {name}.'
My name is Carol.
>>> favorite_number = 498
>>> f'My favorite number is {favorite_number}.'
My favorite number is 498.
>>> f'My favorite number is {400 + 90 + 8}.'
My favorite number is 498.
```

Here's how we'd solve our earlier problem above using f-strings.

```
"""A program to square a number provided by the user
and display the result. """

x = input("Enter a number: ")
x = float(x)
result = x * x
print(f'{x} squared is {result}')
```

[1] "F-string" is short for "formatted string literal". These were introduced in Python 3.6. For details, see the *Input and Output* section of the official Python tutorial (https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals), and PEP 498 (https://peps.python.org/pep-0498/) for a complete rationale behind literal string interpolation.

7.7 Format specifiers

Let's say the result of some calculation was 1/3. The decimal expansion of this is 0.333… Let's print this.

```
>>> 1 / 3
0.3333333333333333
```

Now, in this example, it's unlikely that you'd need or want to display 0.3333333333333333 to quite so many decimal places. Usually, it suffices to print fewer digits to the right of the decimal point, for example, 0.33 or 0.3333. If we were to interpolate this value within an f-string, we'd get a similar result.

```
>>> x = 1 / 3
>>> f'{x}'
'0.3333333333333333'
```

Fortunately, we can tell Python how many decimal places we wish to use. We can include a *format specifier* within our f-string.

```
>>> x = 1 / 3
>>> f'{x:.2f}'
'0.33'
```

The syntax for this is to follow the interpolated element with a colon `:` and then a specifier for the desired format. In the example above, we used `.2f` meaning "display as a floating-point number to two decimal places precision". Notice what happens here:

```
>>> x = 2 / 3
>>> f'{x:.2f}'
'0.67'
```

Python has taken care of rounding for us, rounding the value 0.6666666666666666 to two decimal places. Unless you have very specific reasons not to do so, you should use f-strings for formatting output.

There are many format specifiers we can use. Here are a few:

Format a floating-point number as a percentage.

```
>>> x = 2 / 3
>>> f'{x:.2%}'
'66.67%'
```

Format an integer with comma-separated thousands

```
>>> x = 1234567890
>>> f'{x:,}'
'1,234,567,890'
```

Format a floating-point number with comma-separated thousands

```
>>> gdp = 22996100000000  # USA gross domestic product
>>> population = 331893745  # USA population, 2021 est.
>>> gdp_per_capita = gdp / population
>>> f'${gdp_per_capita:,.2f}'
'$69,287.54'
```

7.8 Scientific notation

Using `E` as a format specifier will result in *normalized* scientific notation.

```
>>> x = 1234567890
>>> f"{x:.4E}"
'1.2346E+09'
```

7.9 Formatting tables

It's not uncommon that we wish to print data or results of calculations in tabular form. For this, we need to be able to specify *width* of a field or column, and the *alignment* of a field or column. For example, say we wanted to print a table like this:[2]

```
                        GDP    Population       GDP per
Country       ($ billion)     (million)    capita ($)
-----------------------------------------------------
Chad                11.780        16.818           700
Chile              317.059        19.768        16,039
China           17,734.063     1,412.600        12,554
Colombia           314.323        51.049         6,157
```

We'd want to left-align the "Country" column, and right-align numbers (numbers, in almost all cases should be right-aligned, and if the decimal point is displayed, all decimal points should be aligned vertically). Let's see how to do that with f-strings and format specifiers. We'll start with the column headings.

[2]Sources: 2021 GDP from World Bank (https://data.worldbank.org/); 2021 population from the United Nations (https://www.un.org/development/desa/pd/).

```
print(f'{"":<12}'
      f'{"GDP":>16}'
      f'{"Population":>16}'
      f'{"GDP per":>16}')
print(f'{"Country":<12}'
      f'{"($ billion)":>16}'
      f'{"(million)":>16}'
      f'{"capita ($)":>16}')
```

This would have been a little long as two single lines, so it's been split into multiple lines.[3] This prints the column headings:

```
                      GDP    Population         GDP per
Country      ($ billion)     (million)      capita ($)
```

< is used for left-alignment (it points left). > is used for right-alignment (it points right). The numbers in the format specifiers (above) designate the width of the field (or column), so the country column is 12 characters wide, GDP column is 16 characters wide, *etc.*

Now let's see about a horizontal rule, dividing the column headings from the data. For that we can use repeated concatenation.

```
print('-' * 60)  # prints 60 hyphens
```

So now we have

```
print(f'{"":<12}'
      f'{"GDP":>16}'
      f'{"Population":>16}'
      f'{"GDP per":>16}')
print(f'{"Country":<12}'
      f'{"($ billion)":>16}'
      f'{"(million)":>16}'
      f'{"capita ($)":>16}')
print('-' * 60)
```

which prints:

```
                      GDP   Population        GDP per
Country       ($ billion)    (million)     capita ($)
------------------------------------------------------
```

Now we need to handle the data. Let's say we have the data in this form:

[3] This takes advantage of the fact that when Python sees two strings without an operator between them it will concatenate them automatically. Don't do this just to save keystrokes. It's best to reserve this feature for handling long lines or building long strings across multiple lines.

```
gdp_chad = 11.780
gdp_chile = 317.059
gdp_china = 17734.063
gdp_colombia = 314.323
pop_chad = 16.818
pop_chile = 19.768
pop_china = 1412.600
pop_colombia = 51.049
```

(Yeah. This is a little clunky. We'll learn better ways to handle data later.) We could print the rows in our table like this:

```
print(f'{"Chad":<12}'
      f'{gdp_chad:>16,.3f}'
      f'{pop_chad:>16,.3f}'
      f'{gdp_chad / pop_chad * 1000:>16,.0f}')

print(f'{"Chile":<12}'
      f'{gdp_chile:>16,.3f}'
      f'{pop_chile:>16,.3f}'
      f'{gdp_chile / pop_chile * 1000:>16,.0f}')

print(f'{"China":<12}'
      f'{gdp_china:>16,.3f}'
      f'{pop_china:>16,.3f}'
      f'{gdp_china / pop_china * 1000:>16,.0f}')

print(f'{"Colombia":<12}'
      f'{gdp_colombia:>16,.3f}'
      f'{pop_colombia:>16,.3f}'
      f'{gdp_colombia / pop_colombia * 1000:>16,.0f}')
```

(Yeah. This is a little clunky too. We'll see better ways soon.) Notice that we can combine format specifiers, so for values in the GDP column we have a format specifier

```
>16,.3f
```

The first symbol `>` indicates that the column should be right-aligned. The `16` indicates the width of the column. The `,` indicates that we should use a comma as a thousands separator. `.3f` indicates formatting as a floating point number, with three decimal places of precision. Other format specifiers are similar.

Putting it all together we have:

```
gdp_chad = 11.780
gdp_chile = 317.059
gdp_china = 17734.063
gdp_colombia = 314.323
```

```
pop_chad = 16.818
pop_chile = 19.768
pop_china = 1412.600
pop_colombia = 51.049

print(f'{"":<12}'
      f'{"GDP":>16}'
      f'{"Population":>16}'
      f'{"GDP per":>16}')
print(f'{"Country":<12}'
      f'{"($ billion)":>16}'
      f'{"(million)":>16}'
      f'{"capita ($)":>16}')
print('-' * 60)

print(f'{"Chad":<12}'
      f'{gdp_chad:>16,.3f}'
      f'{pop_chad:>16,.3f}'
      f'{gdp_chad / pop_chad * 1000:>16,.0f}')

print(f'{"Chile":<12}'
      f'{gdp_chile:>16,.3f}'
      f'{pop_chile:>16,.3f}'
      f'{gdp_chile / pop_chile * 1000:>16,.0f}')

print(f'{"China":<12}'
      f'{gdp_china:>16,.3f}'
      f'{pop_china:>16,.3f}'
      f'{gdp_china / pop_china * 1000:>16,.0f}')

print(f'{"Colombia":<12}'
      f'{gdp_colombia:>16,.3f}'
      f'{pop_colombia:>16,.3f}'
      f'{gdp_colombia / pop_colombia * 1000:>16,.0f}')
```

which prints:

```
                       GDP    Population       GDP per
Country       ($ billion)     (million)    capita ($)
-----------------------------------------------------
Chad               11.780        16.818           700
Chile             317.059        19.768        16,039
China          17,734.063     1,412.600        12,554
Colombia          314.323        51.049         6,157
```

7.10 Example: currency converter

We're starting with programs that take some input, perform some simple calculations, and display some output.

Here we'll demonstrate a currency converter. Consider how we'd like such a program to behave, assuming we need the exchange rate as a user-supplied input. What's the information we'd need to perform the calculation?

- The amount we'd like to convert.
- A label for the currency we'd like to convert, for example, USD, CAD, MXN, BRL, EUR, *etc.*[4] We'll call this the *source currency.*
- An exchange rate.
- A label for the currency we want to receive (as above). We'll call this the *target currency.*

Let's imagine how this might work:

1. The user is prompted for a source currency label.
2. The user is prompted for a target currency label.
3. The user is prompted for an amount (in the source currency).
4. The user is prompted for an exchange rate.
5. The program displays the result.

This raises the question: how do we express the exchange rate? One approach would be to express the rate as the ratio of the value of the source currency unit to the target currency unit. For example, as of this writing, one US dollar (USD) is equivalent to 1.3134 Canadian dollars (CAD).

Taking this approach, we'll multiply the source currency amount by the exchange rate to get the equivalent value in the target currency. Here's how a session might proceed:

```
Enter source currency label: USD
Enter target currency label: CAD
OK. We will convert USD to CAD.
Enter the amount in USD you wish to convert: 1.00
Enter the exchange rate (USD/CAD): 1.3134
1.00 USD is worth 1.31 CAD
```

At this point, we don't have all the tools we'd need to validate input from the user, so for this program we'll trust the user to be well-behaved and to enter reasonable labels and rates. (We'll see more on input validation and exception handling soon.) With this proviso, here's how we might start this program:

```
source_label = input('Enter source currency label: ')
target_label = input('Enter target currency label: ')
print(f'OK. We will convert {source_label} '
      f'to {target_label}.')
```

This code will prompt the user for source and target labels and then print out the conversion the user has requested. Notice that we use an f-string to interpolate the user-supplied labels.

[4]For three-character ISO 4217 standard currency codes, see: https://en.wikipedia.org/wiki/ISO_4217.

Now we need two other bits of information: the amount we wish to convert, and the exchange rate. (Here we'll use the \, which, when used as shown, signifies an explicit line continuation in Python. Python will automatically join the lines.)[5]

```
source_prompt = f'Enter the amount in {source_label} ' \
                f'you wish to convert: '

ex_rate_prompt = f'Enter the exchange rate ' \
                 f'({source_label}/{target_label}): '

source_amount = float(input(source_prompt))
exchange_rate = float(input(ex_rate_prompt))
```

At this point, we have both labels `source_label` and `target_label`, the amount we wish to convert stored as `source_amount`, and the exchange rate stored as `exchange_rate`. The labels are of type `str`. The source amount and the exchange rate are of type `float`.

Giving these objects significant names (rather than `x`, `y`, `z`, ...) makes the code easy to read and understand.

Now, on to the calculation. Since we have the rate expressed as a ratio of the value of a unit of the source currency to a unit of the target currency, all we need to do is multiply.

```
target_amount = source_amount * exchange_rate
```

See how choosing good names makes things so clear? You should aim for similar clarity when naming objects and putting them to use.

The only thing left is to print the result. Again, we'll use f-strings, but this time we'll include format specifiers to display the results to two decimal places of precision.

```
print(f'{source_amount:,.2f} {source_label} is worth '
      f'{target_amount:,.2f} {target_label}')
```

[5] From the Python documentation: Two or more physical lines may be joined into logical lines using backslash characters (), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character (https://docs.python.org/3/reference/lexical_analysis.html).

Putting it all together we get:

```
"""
Currency Converter
Egbert Porcupine <egbert.porcupine@uvm.edu>
CS 1210

Prompts users for two currencies, amount to convert,
and exchange rate, and then performs the conversion
and displays the result.
"""

source_label = input('Enter source currency label: ')
target_label = input('Enter target currency label: ')
print(f'OK. We will convert {source_label} '
      f'to {target_label}.')

source_prompt = f'Enter the amount in {source_label} ' \
                f'you wish to convert: '

ex_rate_prompt = f'Enter the exchange rate ' \
                 f'({source_label}/{target_label}): '

source_amount = float(input(source_prompt))
exchange_rate = float(input(ex_rate_prompt))


target_amount = source_amount * exchange_rate

print(f'{source_amount:,.2f} {source_label} is worth '
      f'{target_amount:,.2f} {target_label}')
```

Notice that we don't need any comments to explain our code. By choosing good names we've made comments unnecessary!

We'll revisit this program, making improvements as we acquire more tools.

Feel free to copy, save, and run this code. There are lots of websites which provide exchange rates and perform such conversions. One such website is the Xe Currency Converter (https://www.xe.com/currencyconverter/).

7.11 Format specifiers: a quick reference

Format specifiers actually constitute a "mini-language" of their own. For a complete reference, see the section entitled "Format Specification Mini-Language" in the Python documentation for strings (https://docs.python.org/3/library/string.html).

Remember that format specifiers are optionally included in f-string replacement fields. The format specifier is separated from the replacement expression by a colon. Examples:

```
>>> x = 0.16251`
>>> f'{x:.1%}'
'16.3%'
>>> f'{x:.2f}'
'0.16'
>>> f'{x:>12}'
'     0.16251'
>>> f'{x:.3E}'
'1.625E-01'
```

Here's a quick reference for some commonly used format specifiers:

option	meaning	example
`<`	align left, can be combined with width	`<12`
`>`	align right, can be combined with width	`>15`
`f`	fixed point notation, combined with precision	`.2f`
`%`	percentage (multiplies by 100 automatically)	`.1%`
`,`	use thousands separators, for example, 1,000	`,`
`E`	scientific notation, combined with precision	`.4E`

7.12 Exceptions

`ValueError`

In an earlier chapter, we saw how trying to use `math.sqrt()` with a negative number as an argument results in a `ValueError`.

In this chapter we've seen another case of `ValueError`—where inputs to numeric constructors, `int()` and `float()`, are invalid. These functions take strings, so the issue isn't the type of the argument. Rather, it's an issue with the *value* of the argument—some strings cannot be used to construct an object of numeric types.

For example,

```
>>> int('5')
5
>>> int('5.0')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '5.0'
>>> int('kumquat')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'kumquat'
```

The first call, with the argument `'5'` succeeds because the string `'5'` can be used to construct an object of type `int`. The other two calls fail with a `ValueError`.

`int('5.0')` fails because Python doesn't know what to do about the decimal point when trying to construct an `int`. Even though 5.0 has a

corresponding integer value, 5, Python rejects this input and raises a `ValueError`.

The last example, `int('kumquat')`, fails for the obvious reason that the string `'kumquat'` cannot be used to construct an integer.

What about the float constructor, `float()`? Most numeric strings are valid arguments to the `float` constructor. Examples:

```
>>> float('3.1415926')
3.1415926
>>> float('7')
7.0
>>> float('6.02E23')
6.02e+23
```

The first example should be obvious. The second example is OK because `'7'` can be used to construct a float. Notice the result of `float('7')` is `7.0`. The third example shows that the float constructor works when using scientific notation as well.

Python has special values for positive and negative infinity (these do come in handy from time to time), and the float constructor can handle the following strings:

```
>>> float('+inf')
inf
>>> float('-inf')
-inf
```

Here are some examples of conversions that will fail, resulting in `ValueError`.

```
>>> float('1,000')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '1,000'
>>> float('pi')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'pi'
>>> float('one')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'one'
>>> float('toothache')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'toothache'
```

What about the string constructor, `str()`? It turns out that this can never fail, since all objects in Python have a default string representation. There's no object type that can't be turned into a string! Even things like

functions have a default string representation (though this representation isn't particularly human-friendly). Example:

```
>>> def f():
...     return 0
...
>>> str(f)
'<function f at 0x10101ea70>'
```

7.13 Exercises

Exercise 01

The following code has a bug. Instead of printing what the user types, it prints `<built-in function input>`. What's wrong and how would you fix it?

```
input('Please enter your name: ')
print(input)
```

Exercise 02

Which of the following are valid arguments to the `int()` constructor? (Try these out in a Python shell.)

a. `'1'`
b. `'1.0'`
c. `'-3'`
d. `5`
e. `'pumpkin'`
f. `'1,000'`
g. `'+1 802 555 1212'`
h. `'192.168.1.1'`
i. `'2023-01-15'`
j. `'2023/01/15'`
k. `2023-01-15`
l. `2023/01/15`

Exercise 03

Which of the following are valid arguments to the `float()` constructor? (Try these out in a Python shell.)

a. `3.141592`
b. `'A'`
c. `'5.0 + 1.0'`
d. `'17'`
e. `'inf'` (Be sure to try this one out! What do you think it means?)
f. `2023/01/15`
g. `2023/1/15`

Exercise 04

Write a program which prompts the user for two floating-point numbers and displays the product of the two numbers entered. Example:

```
Enter a floating-point number: 3.14159
Enter another floating-point number: 17
53.40703
```

Notice here that the second input is an integer string. It's easy to treat an input like this as an integer, just as easily as we could write "17.0" instead of "17".

```
>>> float(17)
17.0
```

Exercise 05

Write a program which prompts the user for their name and then prints 'Hello' followed by the user's name. Example:

```
What is your name? Egbert
Hello Egbert
```

Chapter 8

Branching, comparisons, and conditions

In this chapter, we'll further our understanding of Booleans and Boolean expressions, learn about branching and flow control, comparisons and conditions, and some convenient string methods.

Learning objectives

- You will learn more about Booleans, how to construct a truth table, and how to combine Booleans using the connectives `and` and `or`.
- You will learn how to write `if`, `if/else`, `if/elif`, and `if/elif/else` statements in Python, which allow program execution to follow different branches based on certain conditions.
- You will learn how to test for identity with `is`, and the difference between equivalence and identity.
- You will learn how to represent decisions using and decision trees.
- You will learn a little about input validation.
- You will learn how to use convenient string methods such as `.upper()`, `.lower()`, and `.capitalize()`.

Terms and string methods introduced

- Boolean expression
- branching
- comparison operator
- conditional
- De Morgan's Laws
- decision tree
- equivalence
- identity
- `if`, `elif`, and `else`
- `is`
- lexicographic order
- string methods `.upper()`, `.lower()`, and `.capitalize()`
- truth value
- truthiness and falsiness

8.1 Boolean logic and Boolean expressions

Boolean expressions and Boolean logic are widely used in mathematics, computer science, computer programming, and philosophy. These take their name from the 19th century mathematician and logician George Boole. His motivation was to systematize and formalize the logic that philosophers and others used, which had come down to us from ancient Greece, primarily due to Aristotle.

The fundamental idea is really quite simple: we have truth values—*true* or *false*—and rules for simplifying compound expressions.

It's easiest to explain by example. We'll start with informal presentation, and then formalize things a little later.

Say we have this sentence "It is raining." Now, either it is, or it is not raining. If it is raining, we say that this sentence is *true*. If it is *not* raining, we say that this sentence is *false*.

We call a sentence like this a *proposition*.

Notice that there is no middle ground here. From our perspective, a proposition like this is either true or false. It can't be 67% true and 33% false, for example. We call this the *law of the excluded middle*.

Another way of stating this law is that either a proposition is true, or its negation is true. That is to say, either "It is raining" is true or its *negation*, "It is not raining" (or "It is not the case that it is raining") is true. One or the other.[1]

What does this mean for us as computer programmers? Sometimes we want our code to do one thing if a certain condition is true, and do something different if that condition is false (not true). Without this ability, our programs would be very inflexible.

But before we get to coding, let's learn a little more about *Boolean expressions*.

Boolean expressions

What is a Boolean expression? Well, true and false are both Boolean expressions. We can build more complex expressions using the Boolean connectives *not*, *and*, and *or*.

We often use *truth tables* to demonstrate. Here's the simplest possible truth table. We usually abbreviate true and false as T and F, respectively, but here we'll stick with the Python Boolean literals `True` and `False`.

Expression	Truth value
`True`	`True`
`False`	`False`

[1] There are some logicians who reject the law of the excluded middle. Consider this proposition called the *liar paradox* or *Epimenides paradox*: "This statement is false." Is this true or false? If it's true it's false, if it's false it's true! Some take this as an example of where the law of the excluded middle fails. Thankfully, we don't need to worry about this in this textbook, but if you're curious, see: Law of the excluded middle (Wikipedia). There's even an episode in the original *Star Trek* television series, in which Captain Kirk and Harry Mudd defeat a humanoid robot by confronting it with the liar's paradox. You can view it on YouTube: https://www.youtube.com/watch?v=QqCiw0wD44U.

True is true, and false is false (big surprise, I know).

Now let's go crazy and mix it up. We'll begin with the Boolean connective *not*. `not` simply negates the value or expression which follows.

Expression	Truth value
`True`	`True`
`False`	`False`
`not True`	`False`
`not False`	`True`

Now let's see what happens with the other connectives, *and* and *or*. Some languages have special symbols for these (for example, Java uses `&&` and `||` for *and* and *or* respectively). In Python, we simply use `and` and `or`. When we use these connectives, we refer to the expressions being connected as *clauses*.

Expression	Truth value
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

So when using the conjunctive *and*, the expression is true if and only if both clauses are true. In all other cases (above) the expression is false.

Here's *or*.

Expression	Truth value
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

You see, in the case of *or*, as long as one clause is true, the entire expression is true. It is only when both clauses are false that the entire expression is false.

We refer to clauses joined by *and* as a *conjunction*. We refer to clauses joined by *or* as a *disjunction*. Let's try this out in the Python shell:

```
>>> True
True
>>> False
False
>>> not True
False
>>> not False
True
```

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

Now, we don't usually use literals like this in our code. Usually, we want to test some condition to see if it evaluates to `True` or `False`. Then our program does one thing if the condition is true and a different thing if the condition is false. We'll see how this works in another section.

De Morgan's Laws

When working with Boolean expressions *De Morgan's Laws* provide us with handy rules for transforming Boolean expressions from one form to another. Here we'll use `a` and `b` to stand in for arbitrary Boolean expressions.

`not (a or b)`	is the same as	`(not a) and (not b)`
`not (a and b)`	is the same as	`(not a) or (not b)`

You can think of this as a kind of distributive law for negation. We distribute the `not` over the disjunction `(a or b)`, but when we do, we change the `or` to `and`. By the same token, we distribute `not` over the conjunction `(a and b)`, but when we do, we change the `and` to `or`.

You'll see these may come in handy when we get to input validation (among other applications).

Supplemental information

If you'd like to explore this further, here are some good resources:

- Stanford Encyclopedia of Philosophy's entry on George Boole: https://plato.stanford.edu/entries/boole
- Boolean Algebra (Wikipedia): https://en.wikipedia.org/wiki/Boolean_algebra
- De Morgan's Laws (Wikipedia): https://en.wikipedia.org/wiki/De_Morgan%27s_laws

8.2 Comparison operators

It is often the case that we wish to compare two objects or two values. We do this with *comparison operators*.

Comparison operators compare two objects (or the values of these objects) and return a Boolean `True` if the comparison holds, and `False` if it does not.

Python provides us with the following comparison operators (and more):

Operator	Example	Explanation
`==`	`a == b`	Does the value of `a` *equal* the value of `b`?
`>`	`a > b`	Is the value of `a` *greater than* the value of `b`?
`<`	`a < b`	Is the value of `a` *less than* the value of `b`?
`>=`	`a >= b`	Is the value of `a` *greater than or equal to* the value of `b`?
`<=`	`a <= b`	Is the value of `a` *less than or equal to* the value of `b`?
`!=`	`a != b`	Is the value of `a` *not equal to* the value of `b`?

It's important to understand that *these operators perform comparisons* and expressions which use them to *evaluate to a Boolean value* (`True` or `False`).

Let's demonstrate in the Python shell.

```
>>> a = 12
>>> b = 31
>>> a == b
False
>>> a > b
False
>>> a < b
True
>>> a >= b
False
>>> a <= b
True
>>> a != b
True
>>> not (a == b)
True
```

Now what happens in the case of strings? Let's try it and find out!

```
>>> a = 'duck'
>>> b = 'swan'
>>> a == b
False
>>> a > b
False
```

```
>>> a < b
True
>>> a >= b
False
>>> a <= b
True
>>> a != b
True
>>> not (a == b)
True
```

What's going on here? When we compare strings, we compare them *lexicographically*. A string is *less than* another string if its lexicographic order is lower than the other. A string is *greater than* another string if its lexicographic order is greater than the other.

What is *lexicographic* order?

Lexicographic order is like alphabetic order, but is somewhat more general. Consider our example 'duck' and 'swan'. This is an easy case, since both are four characters long, so alphabetizing them is straightforward.

But what about 'a' and 'aa'? Which comes first? Both start with 'a' so their first character is the same. If you look in a dictionary you'll find that 'a' appears before 'aa'.[2] Why? Because when comparing strings of different lengths, the comparison is made as if the shorter string were padded with an invisible character which comes before all other characters in the ordering. Hence, 'a' comes before 'aa' in a lexicographic ordering.

```
>>> 'a' < 'aa'
True
>> 'a' > 'aa'
False
```

The situation is a little more complex than this, because strings can have any character in them (not just letters, and hence the term "alphabetic order" loses its meaning). So what Python actually compares are the *code points* of Unicode characters. Unicode is the system that Python uses to encode character information, and Unicode includes many other alphabets (Arabic, Armenian, Cyrillic, Greek, Hangul, Hebrew, Hindi, Telugu, Thai, *etc.*), symbols from non-alphabetic languages such as Chinese or Japanese Kanji, and many special symbols (®, €, ±, ∞, *etc.*). Each character has a number associated with it called a *code point* (yes, this is a bit of a simplification). In comparing strings, Python compares these values.[3]

[2] Yes, "aa" is a word, sometimes spelled "a'a". It comes from the Hawai'ian, meaning rough and jagged cooled lava (as opposed to *pahoehoe*, which is very smooth).

[3] If you want to get really nosy about this, you can use the Python built-in function `ord()` to get the numeric value associated with each character. *E.g.,*

```
>>> ord('A')
65
```

Thus, `'duck' < 'swan'` evaluates to `True`, `'wing' < 'wings'` evaluates to `True`, and `'bingo' < 'bin'` evaluates to `False`.

```
>>> 'duck' < 'swan'
True
>>> 'wing' < 'wings'
True
>>> 'bingo' < 'bin'
False
```

Now, you may wonder what happens in alphabetic systems, like English and modern European languages, which have majuscule (upper-case) and miniscule (lower-case) letters (not all alphabetic systems have this distinction).

```
'a' > 'A'
True
'a' < 'A'
False
```

Upper-case letters have lower order than lower-case letters.

```
>>> 'ALPHA' < 'aLPHA'
True
```

So keep this in mind when comparing strings.

8.3 Branching

Up until this point, all the programs we've seen and written proceed in a linear fashion from beginning to end. This is fine for some programs, but it's rather inflexible. Sometimes we want our program to respond differently to different conditions.

Imagine we wanted to write a program that calculates someone's income tax. The US Internal Revenue Service recognizes five different filing statuses:

- single,
- married, filing jointly,
- married, filing separately,
- head of household,

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord('£')
163
```

See also: Joel Spolsky's *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)* last seen in the wild at https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/

- qualifying widow or widower with dependent child.[4]

So in writing our program we'd need to prompt the user for different questions, gather different data, perform different calculations, and use different tax tables depending on a user's filing status. Obviously, this cannot be done in a strictly linear fashion.

Instead, we'd want our program to be able to make decisions, and follow different *branches*, depending on the results of those decisions.

This example of an income tax program is by no means unusual. In fact, most real-world programs involve some kind of branching.

When our program includes branches, we execute different portions of our program depending on certain conditions. Which conditions might those be? It depends entirely on the program we wish to write.

Thus, most programming languages (Python included) allow for *control flow*—which includes branching and conditional execution of code.

How do we do this? In Python, we accomplish this with `if`, `elif` and `else` statements (or combinations thereof).

8.4 `if`, `elif`, and `else`

`if`, `elif`, and `else` work with Boolean expressions to determine which branch (or branches) our program will execute.

Since tax preparation is complicated, let's consider more modest examples.

Examples using `if`, `elif`, and `else`

A minimal program we could write using `if` and `else` might work like this:

- Prompt the user to guess a magic word.
- If the user guesses correctly, print "You WIN!"
- If the user does not guess correctly, print "You LOSE!"

Let's think about what we'd need for this program to work:

- A secret word.
- Prompt the user for their guess.
- Then *compare* the user's guess with the secret word.
- Print the appropriate message.

Here's how we can do this in Python using `if` and `else`.

```
"""
CS1210
Guess the secret word
"""

secret_word = "secret"
```

[4]Discussion of the fairness or consequences of such a classification is outside the scope of this text—a state of affairs that suits this author just fine.

```
user_guess = input("What do you think the secret word is? ")

if user_guess == secret_word:
    print("You WIN!")
else:
    print("You LOSE!")
```

Here's another example, but this time we're using a nested `if` statement and an `elif` (which is a portmanteau of "else" and "if"):

```
"""
CS 1210
What's for dinner?
"""

bank_balance = float(input('What is your bank balance $? '))

if bank_balance < 20.0:
    meal_points = input('Do you have any meal points? y/n: ')
    if meal_points == 'y':
        print('Eat at the dining hall.')
    else:
        print('Eat that leftover burrito.')
elif bank_balance < 50.0:
    print('Order pizza from Leonardo\'s')
else:
    print('Go out to Tiny Thai in Winooski with a friend!')
```

First we prompt the user for their bank balance. If this amount is less than $20.00 then we prompt the user *again* to find out if they have any meal points left. If they do, that is, `if meal_points == 'y'`, we print "Eat at the dining hall." If not, we print "Eat that leftover burrito."

Now, what happens if that very first condition is false? If that's false, we *know* we have more than $20.00, so our next comparison is:

```
elif bank_balance < 50.0:
```

Why not

```
elif bank_balance >= 20 and bank_balance < 50.0:
```

you might ask? Because we only reach the `elif` if the first condition is false. There's no need to check again.

So if the bank balance is greater than or equal to $20.00 and less than $50.00 we print "Order pizza from Leonardo's".

Now, what if the bank balance is greater than or equal to $50.00? We print "Go out to Tiny Thai in Winooski with a friend!".

We can have a single `if` statement, without `elif` or `else`. We can also, as we've just seen, write compound statements which combine `if` and `else`, `if` and `elif`, or all three, `if`, `elif`, and `else`. We refer to each block of code in such compound statements as *clauses* (distinct from clauses in a compound Boolean expression).

Some important things to keep in mind

1. If we have a compound `if/else` statement in our program either the body of the `if` clause is executed or the body of the `else` clause is executed—never both.

2. If we have a compound `if/elif/else` statement in our program, the body of only one of the branches is executed.

Supplemental resources

For more on control of flow, see: https://docs.python.org/3/tutorial/controlflow.html

8.5 Truthy and falsey

Python allows many shortcuts with Boolean expressions. Most everything in Python has a truth value. As noted earlier, we refer to the truth values of anything other than Booleans with the whimsical terms "truthy" and "falsey" (we also use the terms "truthiness" and "falsiness").

When used in Boolean expressions and conditions for loops or branching (`if/elif`), *truthy* values are treated (more-or-less) as `True`, and *falsey* values are treated (more-or-less) as `False`.

Truthy things	Falsey things
any non-zero valued `int` or `float`, `5`, `-17`, `3.1415`	`0`, `0.0`
any non-empty list, `['foo']`, `[0]`, `['a', 'b', 'c']`	the empty list, `[]`
any non-empty tuple, `(42.781, -73.901)`, `('vladimir')`	the empty tuple, `()`
any non-empty string, `"bluster"`, `"kimchee"`	the empty string, `""`

This allows us to use conditions such as these:

```
if x % 2:
    # it's odd
    print(f"{x} is odd")
```

```
if not s:
   print("The string, s, is empty!")
```

If you want to know if something is truthy or falsey in Python, you can pass the value or expression to the Boolean constructor, `bool()`.

```
>>> bool(1)
True
>>> bool(-1)
True
>>> bool(0)
False
>>> bool('xyz')
True
>>> bool('')
False
>>> bool(None)
False
```

8.6 Identity and equivalence

Identity and equivalence aren't the same thing. Here's a simple example:

```
>>> a = 12345678987654321
>>> b = 12345678987654321
>>> a == b
True
```

So far, this is exactly what we'd expect: `a` and `b` certainly have the same value.

Python provides the keyword `is` to test for identity—asking if two things are actually the same object going by different names.

```
>>> a is b
False
```

Python is telling us that despite `a` and `b` having the exact same value, they are, in fact two different objects. In other words, the value of `a` is equivalent to the value of `b`, but `a` and `b` refer to different objects.

Let's check.

```
>>> id(a)
4312275152
>>> id(b)
4312270928
```

Yup. They have different ids (if you try this on your computer you'll get different ids, but the id for `a` will differ from the id for `b`).

Let's check another way.

```
>>> a = 5
>>> b
12345678987654321
```

Yup. We've changed the value of `a` and the value of `b` remains unchanged. Clearly these are different objects.

Let's see an example of two identifiers referring to the same object.

```
>>> a = 12345678987654321
>>> b = a
>>> a is b
True
>>> b is a
True
```

In this instance, `a` and `b` are different names for the same object. Let's check their ids.

```
>>> id(a)
4312271472
>>> id(b)
4312271472
```

Yup. They're exactly the same.

When is this useful? Sometimes we want to know whether two names refer to the same object. We'll see some interesting examples when we learn about *mutability* in Chapter 10. For now, the most common use of `is` is to check for `None` in a conditional.

If you recall, `None` is a special value in Python which means "no value" (I know, it's a little odd having an object that has a value which means "no value", but there you have it). There are never multiple copies of `None`. `None` is a singular object.

```
>>> a = None
>>> b = 42
>>> b = None
>>> c = None
>>> d = None
>>> a == b == c == d   # This is true,...
True
>>> a is b is c is d   # but this is more important.
True
```

All these variables are just different names for the same object, the singular object `None`. Accordingly, the best practice when checking for `None` isn't to use the comparison operator (`==`) but rather the identity keyword `is`.

```
if x is None:
    print("x has no value")
```

8.7 Input validation

Earlier, we wrote a Python program to convert kilograms to pounds. We trusted the user to provide a valid integer or float for weight in kilograms, and we did nothing to ensure that the information provided was reasonable. That is, we did not *validate* the input.

Validation of input is a crucial part of any program that accepts user input. Users sometimes provide invalid input. Some reasons for this include:

- The prompt was insufficiently clear.
- The user did not read the prompt carefully.
- The user did not understand what kind of input is needed.
- The user was being mischievous—trying to break the program.
- The user made a typo or formatting error when entering data.

We would like our programs to respond gracefully to invalid input.

In this textbook, we'll see several different ways to validate input and to respond to invalid input. The first that we'll learn right now is simple *bounds checking*.

Bounds checking is an approach to input validation which ensures that a value is in some desired range. To return to our kilogram to pound conversion program, it does not make sense for the user to enter a negative value for the weight in kilograms. We might guard against this with bounds checking.

```
POUNDS_PER_KILOGRAM = 2.204623

kg = float(input('Enter weight in kilograms: '))
if kg >= 0:
    # convert kilograms to pounds and print result
else:
    print('Invalid input! '
          'Weight in kilograms must not be negative!')
```

So our program would perform the desired calculations if and only if the weight in kilograms entered by the user were non-negative (that is, greater than or equal to zero).

Here's another example. Let's say we're writing a program that plays the game *evens and odds*. This is a two-player game where one player calls "even" or "odd", and then the two players simultaneously reveal zero, one, two, three, four or five fingers. Then the sum is calculated and the caller wins if the sum agrees with their call.

In such a game, we'd want the user to enter an integer in the interval $[0, 5]$. Here's how we might validate this input:

```
fingers = int(input('Enter a number of fingers [0, 5]: '))
if fingers >= 0 and fingers <= 5:
    # Generate a random integer in the range [0, 5],
    # calculate sum, and report the winner.
else:
    print('Invalid input!')
```

Admittedly, these aren't satisfactory solutions. Usually, when a user enters invalid data the program gives the user another chance, or chances, until valid data are supplied. We'll see how to do this soon.

Nevertheless, simple bounds checking is a good start!

Comprehension check

1. Can you use De Morgan's Laws (see: Boolean expressions) to rewrite the bounds checking above?
2. If we were to do this, would we be checking to see if `fingers` is in the desired range or outside the desired range?
3. If your answer to 2 (above) was *outside the desired range*, how would you need to modify the program?

8.8 Some string methods

Python provides us with many tools for manipulating strings. We won't introduce them all here, but instead we'll demonstrate a few which we'll use in programming exercises, and then introduce more as we need them.

First, what is a *string method*? If you've ever programmed in Java or C# or other OOP language, you may be familiar with methods. If not, don't fret, because the concept isn't too difficult.

Strings are a type of *object* in Python. Consider what happens when we ask Python what type the string "Mephistopheles" is.

```
>>> type('Mephistopheles')
<class 'str'>
```

What Python is telling us is that "Mephistopheles" is an object of type `str`.

When the developers of Python defined the types `str`, `int`, `float`, `bool`, *etc.* they created *classes* corresponding to these different types of objects. We won't cover object-oriented programming in this book, but you can think of a class as a blueprint for creating objects of a given type. We *instantiate* an object by making an assignment, for example

```
n = 42
```

creates an *object* of type `int`, and

```
s = 'Cheese Shoppe'
```

creates an *object* of type `str`, and so on. The class definitions give Python a blueprint for instantiating objects of these different types.

One of the things classes allow us to do is to define *methods* that are part of the class definition and which are included with the objects along with their data. Methods are nothing more than functions defined for a class of objects which operate on the data of those objects.

Here's an example. Let's create a string object, `s`

```
>>> s = 'mephistopheles'
```

Now, just like we can access individual members of the `math` module with the member (`.`) operator (for example, `math.pi`, `math.sqrt(2)`, `math.sin(0.478)`, *etc.*) we can access string methods the same way!

For example, the `capitalize()` method can be called for any string object, and it will return a copy of the string with the first character capitalized (note: this does not modify the string, it just returns a copy of the string).

```
>>> s = 'mephistopheles'  # note: this is all lower case
>>> s.capitalize()
'Mephistopheles'
```

Here's another method: `upper()` (you can guess what this does).

```
>>> s = 'mephistopheles'  # note: this is all lower case
>>> s.upper()
'MEPHISTOPHELES'
```

Now what if we had a string in all upper case, but wanted it in lower case?

```
>>> s = 'PLEASE STOP YELLING'
>>> s.lower()
'please stop yelling'
```

As you might imagine, these can come in handy, and there are many more (take a peek at *Built-in Types* (https://docs.python.org/3/library/stdtypes.html) and scroll down to *String Methods* if you're curious).

It is important to keep in mind that these do not alter the string's value, they only *return* an altered copy of the string.

```
>>> s = 'PLEASE STOP YELLING'
>>> s.lower()
'please stop yelling'
>>> s
'PLEASE STOP YELLING'
```

If you want to use the result returned by these methods you may need to assign the result to a new object or overwrite the value of the current variable, thus:

```
>>> s = 'PLEASE STOP YELLING'
>>> s = s.lower()
>>> s
'please stop yelling'
```

This isn't always necessary, but keep this in mind.

Some applications

Let's say we wanted to write a program that prompted the user to see if they wish to continue. At some point in our code, we might have something like this:

```
response = input('Do you wish to continue? Enter "y" '
                 'to continue or any other key to abort: ')
if response == 'y':
    # This is where we'd continue whatever we were doing
else:
    # This is where we'd abort
```

What would happen if the user were to enter upper case 'Y'? Clearly the user intends to continue, but the comparison

```
response == 'y'
```

would return `False` and the program would abort. That might make for an unhappy user.

We could write

```
response = input('Do you wish to continue? Enter "y" '
                 'to continue or any other key to abort: ')
if response == 'y':
    # This is where we'd continue whatever we were doing
elif response == 'Y':
    # This is where we'd continue whatever we were doing
else:
    # This is where we'd abort
```

or

```
response = input('Do you wish to continue? Enter "y" '
                 'to continue or any other key to abort: ')
if response == 'y' or response == 'Y':
    # This is where we'd continue whatever we were doing
else:
    # This is where we'd abort
```

Instead, we could use `lower()` and simplify our code!

```
response = input('Do you wish to continue? Enter "y" to '
                 'continue or any other key to abort: ')
if response.lower() == 'y':
    # This is where we'd continue whatever we were doing
else:
    # This is where we'd abort
```

This code (above) behaves the same whether the user enters ‘y’ or ‘Y’, because we convert to lower case before performing the comparison. This is one example of an application for string methods.

Another might be dealing with users that have the CAPS LOCK key on.

```
name = input('Please enter your name: ')
# Now what if the user enters: 'EGBERT'?
# We can fix that:
name = name.capitalize()
# Now name is 'Egbert'
```

There are *lots* of uses.

8.9 Decision trees

We may represent a decision-making process diagrammatically with a *decision tree*. Decision trees are commonly used for species identification.[5]

Here's an example:

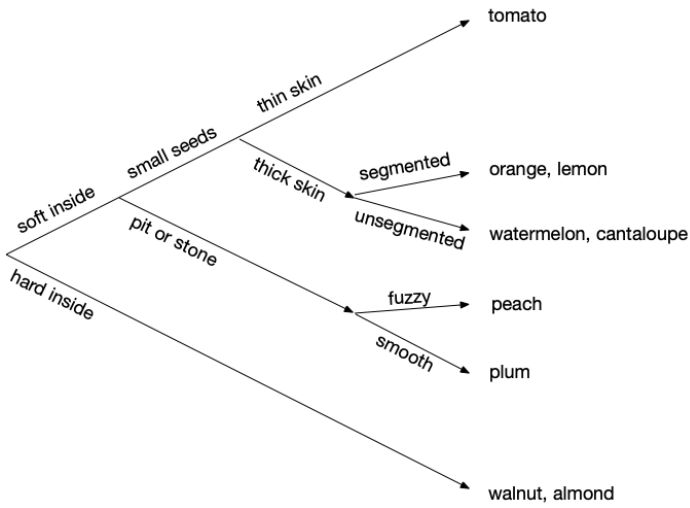


Figure 8.1: Decision tree for fruit

Here, we start on the left and move toward the right, making decisions along the way. Notice that at each branching point we have two branches.

So, for example, to reach "watermelon, cantaloupe", we make the decisions: soft inside, small seeds, thick skin, and unsegmented. To reach "peach", we'd have to have made the decisions: soft inside, pit or stone, and fuzzy.

How do we encode these decision points? One way is to treat them as yes or no questions. So if we ask "Is the fruit soft inside?" then we have a yes or no answer. If the answer is "no", then we know the fruit is not soft inside and thus must be hard inside (like a walnut or almond).

Here's a snippet of Python code, demonstrating a single question:

```
response = input('Is the fruit soft inside? y/n  ')
if response == 'y':
    # we know the fruit is soft inside
    # ...
else:
    # we know the fruit is hard inside
    # ...
```

[5]If you've had a course in biology, you may have heard of a *cladogram* for representing taxonomic relationships of organisms. A cladogram is a kind of decision tree. If you're curious, see: https://en.wikipedia.org/wiki/Cladogram and similar applications.

We can write a program that implements this decision tree by using multiple, nested `if` statements.

```
"""
CS 1210
Decision tree for fruit identification
"""

response = input('Is the fruit soft inside? y/n  ')
if response.lower() == 'y':
    # soft inside
    response = input('Does it have small seeds? y/n  ')
    if response.lower() == 'y':
        # small seeds
        response = input('Does it have a thin skin? y/n  ')
        if response.lower() == 'y':
            # thin skin
            print("Tomato")
        else:
            # thick skin
            response = input('Is it segmented? y/n  ')
            if response.lower() == 'y':
                # segmented
                print("Orange or lemon")
            else:
                # unsegmented
                print("Watermelon or cantaloupe")
    else:
        # pit or stone
        response = input('Is it fuzzy? y/n  ')
        if response.lower() == 'y':
            # segmented
            print("Peach")
        else:
            # unsegmented
            print("Plum")
else:
    # hard inside
    print("Walnut or almond")
```

Comprehension check

1. In the decision tree above, Figure 8.1, which decisions lead to plum? (There are three.)

2. Revisit Section 8.4 and draw decision trees for the code examples shown.

8.10 Exceptions

UnboundLocalError

While there are several ways one's code might raise an `UnboundLocalError` exception, one of the most common involves branching. As noted earlier, an *unbound local error* is a more specific kind of `NameError`—one which occurs within a function or method. This type of error is raised when we try to read a value from a variable before the variable has been assigned a value within the body (scope) of a function. Here's an example:

```
def f(x):
    if x > 0:
        y = 1
    elif x < 0:
        y = -1
    return y
```

This might seem OK, but what would happen if the argument passed to this function were zero? Neither of the branches would be executed and we'd wind up trying to return `y` before it had been assigned a value, and Python would complain with `UnboundLocalError: local variable 'y' referenced before assignment`.

We can fix this by repairing the `if`/`elif` to include an `else`.

```
def f(x):
    if x > 0:
        y = 1
    elif x < 0:
        y = -1
    else:
        y = 0
    return y
```

Or we could provide a default value for `y`:

```
def f(x):
    y = 0
    if x > 0:
        y = 1
    elif x < 0:
        y = -1
    return y
```

Or, we could dispense with `y` altogether.

```
def f(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    return 0
```

Any of the above will work.

8.11 Exercises

Exercise 01

Evaluate the result of the following, given that we have:

```
a = True
b = False
c = True
```

Do these on paper first, then check your answers in the Python shell.

1. `a or b and c`
2. `a and b or c`
3. `a and b and c`
4. `not a or not b or c`
5. `not (a and b)`

Exercise 02

Evaluate the result of the following, given that we have:

```
a = 1
b = 'pencil'
c = 'pen'
d = 'crayon'
```

Do these on paper first, then check your answers in the Python shell. Some of these may surprise you!

1. `a == b`
2. `b > c`
3. `b > d or a < 5`
4. `a != c`
5. `d == 'rabbit'`
6. `c < d or b > d`
7. `a and b < d`
8. `(a == b) and (b != c)`
9. `(a and b) and (b < c)`
10. `not (a and b and c and d)`

Ask yourself, what does it mean for `'crayon'` to be less than `'pencil'`? How would you interpret this? Ask yourself, what's going on when an expression like `0 or 'crayon'` is evaluated?

Exercise 03

Complete the following `if` statements so that they print the correct message. Notice that there are blank spaces in the code that you should complete. You may assume we have three variables, with string values assigned: `cheese`, `blankets`, `toast`, for example,

```
cheese = 'runny'
```

1. Cheese is smelly and blankets are warm!

```
if cheese == 'smelly' and                        :
    print('Cheese is smelly and blankets are warm!')
```

2. Blankets are warm and toast is not pickled. Hint: use `not` or `!=`

```
if blankets                                  :
    print('Blankets are warm but toast is not pickled.')
```

3. Toast is yummy and so is cheese!

```
if                                   :
    print('Toast is yummy and so is cheese!')
```

4. Either toast is yummy or toast is green (or maybe both).

```
if                                   :
    print('Either toast is yummy or toast is green '
          '(or maybe both).')
```

Exercise 04

What is printed at the console for each of the following?

1.

```
>>> 'HELLO'.capitalize()
```

2.

```
>>> s = 'HoverCraft'
>>> s.lower()
```

3.

```
>>> s = 'wATer'.lower()
>>> s
```

Exercise 05

If we have only two possible outcomes in a decision tree, and decisions are binary, then our tree has only one branching point. If we have four possible outcomes, then our tree must have three branching points.

a. If we have eight possible outcomes in a decision tree, and decisions are binary, how many branching points must we have?

b. What about 16?

c. Can you find a formula that calculates the number of branching points given the number of outcomes? (OK if you can't, so don't sweat it.)

Exercise 06

Consider this code.

```
def order_pizza(bank_balance, weekday):
    """OK to order if bank balance is $50 or more and it's
    not a Sunday (pizza not allowed on Sunday!) """
    if bank_balance < 50.0:
        place_order = False
    elif weekday == 'Sunday':
        place_order = False
    return place_order
```

This can fail with `UnboundLocalError`. At the very least, repair this `if/elif` to prevent such an exception. However, it's probably best to write this with a one-line body. Give it a try!

Exercise 07

Render the following in Python.

a. If it's raining, the sidewalk is wet.

b. If your name is "Egbert" you must be a porcupine.

c. If x is a multiple of seven, let $y = 5$.

Chapter 9

Structure, development, and testing

I don't know what I think until I write it down.
–Joan Didion

It's important to be methodical when programming, and in this chapter we'll see how best to structure your Python code. Following this structure takes much of the guesswork out of programming. Many questions about where certain elements of your program belong are already answered for you. What's presented here is based on common (indeed nearly universal) practice for professionally written code.

We'll also learn a little bit about how to proceed when writing code (that is, in small, incremental steps), how to test your code, how to use *assertions*, and what to do about the inevitable bugs.

Learning objectives

- You will learn about incremental development, and how to use comments as "scaffolding" for your code.
- You will learn how to organize and structure your code.
- You will understand how Python handles the main *entry point* of your program, and how Python distinguishes between modules that are imported and modules that are to be executed.
- You will be able to write code with functions that can be imported and used independently of any *driver code*.
- You will understand how to test your code, and when to use assertions in your code.

Terms and Python keywords introduced

- `assert` (Python keyword) and assertions
- `AssertionError`
- bug
- driver code
- dunder
- entry point and top-level code environment
- incremental development
- namespace
- rubberducking

9.1 `main` the Python way

So far, we've followed this general outline:

```
"""
A program which prompts the user for a radius of a circle,
r, and calculates and reports the circumference.
"""

import math

def circumference(r_):
    return 2 * math.pi * r_

r = float(input('Enter a non-negative real number: '))
if r >= 0:
    c = circumference(r)
    print(f'The circumference of a circle of radius '
          f'{r:,.3f} is {c:,.3f}.')
else:
    print(f'I asked for a non-negative number, and '
          f'{r} is negative!')
```

This is conventional and follows good coding style (PEP 8).

You may have seen something like this:

```
"""
A program which prompts the user for a radius of a circle,
r, and calculates and reports the circumference.
"""

import math

def circumference(r_):
    return 2 * math.pi * r_

def main():
    r = float(input('Enter a non-negative real number: '))
    if r >= 0:
        c = circumference(r)
        print(f'The circumference of a circle of radius '
              f'{r:,.3f} is {c:,.3f}.')
    else:
        print(f'I asked for a non-negative number, and '
              f'{r} is negative!')

main()
```

While this is not syntactically incorrect, it's not really the Python way either.

Some textbooks use this, and there are abundant examples on the internet, perhaps attempting to make Python code look more similar to languages like C or Java (in the case of Java, an executable program *must* implement `main()`). But again, *this is not the Python way.*

Here's how things work in Python. Python has what is called the *top-level code environment.* When a program is executed in this environment (which is what happens when you run your code within your IDE or from the command line), there's a special variable `__name__` which is automatically set to the value `'__main__'`.[1] `'__main__'` is the name of the environment in which top-level code is run.

So if we wish to distinguish portions of our code which are automatically run when executed (sometimes called *driver code*) from other portions of our code (like imports and the functions we define), we do it thus:

[1]Some other programming languages refer to the top-level as the *entry point.* `'__main__'` is the name of a Python program's entry point.

```
"""
A program which prompts the user for a radius of a circle,
r, and calculates and reports the circumference.
"""

import math

def circumference(r_):
    return 2 * math.pi * r_

if __name__ == '__main__':

    # This code will only be executed if this module
    # (program) is run. It will *not* be executed if
    # this module is imported.

    r = float(input('Enter a non-negative real number: '))
    if r >= 0:
        c = circumference(r)
        print(f'The circumference of a circle of radius '
              f'{r:,.3f} is {c:,.3f}.')
    else:
        print(f'I asked for a non-negative number, and '
              f'{r} is negative!')
```

Let's say we saved this file as `circle.py`. If we were to run this program from our IDE or from the command line with

```
$ python circle.py
```

Python would read the file, would see that we're executing it, and thus would set `__name__` equal to `'__main__'`. Then, after reading the definition of the function `circumference(r_)`, it would reach the `if` statement,

```
if __name__ == '__main__':
```

This condition evaluates to `True`, and the code nested within this `if` statement would be executed. So it would prompt the user for a radius, and then check for valid input and return an appropriate response.

Another simple demonstration

Consider this Python program

```
"""
tlce.py (top-level code environment)
Another program to demonstrate the significance
of __name__ and __main__.
"""

print(__name__)

if __name__ == '__main__':
    print("Hello World!")
```

Copy this code and save it as `tlce.py` (short for top-level code environment). Then, try running this program from within your IDE or from the command line. What will it print when you run it? It should print

```
__main__
Hello World!
```

So, you see, when we run a program in Python, Python sets the value of the variable `__name__` to the string `'__main__'`, and then, when the program performs the comparison `__name__ == '__main__'` this evaluates to `True`, and the code within the `if` is executed.

What happens if we import our module in another program?

Now write another program which imports this module (formally we refer to Python programs as *modules*).

In the same directory where you have `tlce.py`, create a new file

```
"""
A program which imports tlce (from the previous example).
"""

import tlce
```

Save this as `use_tlce.py` and then run it. What is printed? This program should print

```
tlce
```

So, if we import `tlce` then Python sets `__name__` equal to `'tlce'`, and the body of the `if` is never executed.

Why would we do this? One reason is that we can write functions in one module, and import the module without executing any of the module's code, but make the functions available to us. Sound familiar? It should. Consider what happens when we import the `math` module.

Nothing is executed, but now we have `math.pi`, `math.sqrt()`, `math.sin()`, *etc.* available to us.

A complete example

Earlier we created a program which, given some radius, r, provided by the user, calculated the circumference, diameter, surface area, and volume of a sphere of radius r. Here it is, with some minor modifications, notably the addition of the check on the value of `__name__`.

```
"""
Sphere calculator (sphere.py)

Prompts the user for some radius, r, and then prints
the circumference, diameter, surface area, and volume
of a sphere with this radius.
"""

import math

def circumference(r_):
    return 2 * math.pi * r_

def diameter(r_):
    return 2 * r_

def surface_area(r_):
    return 4 * math.pi * r_ ** 2

def volume(r_):
    return 4 / 3 * math.pi * r_ ** 3

if __name__ == '__main__':
    r = float(input("Enter a radius >= 0.0: "))
    if r < 0:
        print("Invalid input")
    else:
        print(f"The diameter is "
              f"{diameter(r):0,.3f} units.")
        print(f"The circumference is "
              f"{circumference(r):0,.3f} units.")
        print(f"The surface area is "
              f"{surface_area(r):0,.3f} units squared.")
        print(f"The volume is "
              f"{volume(r):0,.3f} units cubed.")
```

Now we have a program that prompts the user for some radius, r, and uses some convenient functions to calculate these other values for a sphere. But it's not a stretch to see that we might want to use these functions somewhere else!

Let's say we're manufacturing yoga balls—those inflatable balls that people use for certain exercises requiring balance. We'd want to know how much plastic we'd need to manufacture some number of balls. Say our yoga balls are 33 centimeters in radius when inflated, and that we want the thickness of the balls to be 0.1 centimeter.

In order to complete this calculation, we'll need to calculate volume. Why reinvent the wheel? We've already written a function to do this!

Let's import `sphere.py` and use the function provided by this module.

```
"""
Yoga ball material requirements
"""

import sphere
# sphere.py must be in the same directory for this to work

RADIUS_CM = 33
THICKNESS_CM = 0.1
VINYL_G_PER_CC = 0.95
G_PER_KG = 1000

if __name__ == '__main__':
    balls = int(input("How many balls do you want "
                       "to manufacture this month? "))
    outer = sphere.volume(RADIUS_CM)
    inner = sphere.volume(RADIUS_CM - THICKNESS_CM)
    material_per_ball = outer - inner
    total_material = balls * material_per_ball
    total_material_by_weight
            = total_material / VINYL_G_PER_CC / G_PER_KG

    print(f"To make {balls} balls, you will need "
          f"{total_material:,.1f} cc of vinyl.")
    print(f"Order at least "
          f"{total_material_by_weight:,.1f} "
          f"kg of vinyl to meet material requirements.")
```

See? We've imported `sphere` so we can use its functions. When we import `sphere`, `__name__` (for `sphere`) takes on the value `sphere` so the code under `if __name__ == '__main__'` isn't executed!

This allows us to have our cake (a program that calculates diameter, circumference, surface area, and volume of a sphere) and eat it too (by allowing imports and code reuse)! How cool is that?

What's up with the funny names?

These funny names `__name__` and `'__main__'` are called *dunders*. Dunder is short for *double underscore*. This is a naming convention that Python uses to set special variables, methods, and functions apart from the typical names programmers use for variables, methods, and functions they define.

9.2 Program structure

There is an order to things, and programs are no different. Your Python code should follow this general layout:

1. docstring
2. imports (if any)
3. constants (if any)
4. function definitions (if any)

... and then, nested under `if __name__ == '__main__':`, all the rest of your code. Here's an example:

```
"""
A docstring, delimited by triple double-quotes,
which includes your name and a brief description
of your program.
"""

import foo   # imports (if any)

MEGACYCLES_PER_FROMBULATION = 133   # constants (if any)

# Functions which you define...
def f(x_):
    return 2 * x_ + 1

def g(x_):
    return (x_ - 1) ** 2

# The rest of your code...
if __name__ == '__main__':
    x = float(input("Enter a real number: "))
    print(f"Answer: {f(g(x))
            / MEGACYCLES_PER_FROMBULATION} megacycles!")
```

9.3 Iterative and incremental development

Incremental development is a process whereby we build our program incrementally—often in small steps or by components. This is a structured, step-by-step approach to writing software. This approach has long been used to make the process of building complex programs more reliable. Even if you're not undertaking a large-scale software development project, this approach can be fruitful. Moreover, decomposing problems into small portions or components can help reduce the complexity of the task you're working on at any given time.

Here's an example. Let's say we want to write a program that prompts the user for mass and velocity and calculates the resulting kinetic energy. If you haven't had a course in physics before, don't sweat it—the formula is rather simple.

$$K_e = \frac{1}{2}mv^2$$

where K_e is kinetic energy in Joules, m is mass in kg, and v is velocity in m / s.

How would you go about this incrementally? The first step might be to sketch out what needs to happen with comments.[2]

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and save result
# Step 2: Prompt user for velocity in m / s and save result
# Step 3: Calculate kinetic energy in Joules using formula
# Step 4: Display pretty result
```

That's a start, but then you remember that Python's `input()` function returns a string, and thus you need to convert these strings to floats. You decide that before you start writing code you'll add this to your comments, so you don't forget.

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and convert input
#         to float and save result
# Step 2: Prompt user for velocity in m / s and convert
#         input to float and save result
# Step 3: Calculate kinetic energy in Joules using formula
# Step 4: Display pretty result
```

Now you decide you're ready to start coding, so you start with step one.

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and convert input
#         to float and save result
mass = float(input('Enter mass in kg: '))
print(mass)
# Step 2: Prompt user for velocity in m / s and convert
#         input to float save result
# Step 3: Calculate kinetic energy in Joules using formula
# Step 4: Display pretty result
```

[2]Here we've excluded `if __name__ == __main__:` to avoid clutter in presentation.

Notice the comments are left intact and there's a print statement added to verify mass is correctly stored in `mass`. Now you run your code—yes, it's incomplete, but you decide to run it to confirm that the first step is correctly implemented.

```
Enter mass in kg: 72.1
72.1
```

So that works as expected. Now you decide you can move on to step two.

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and convert
#         input to float and save result
mass = float(input('Enter mass in kg: '))
print(mass)
# Step 2: Prompt user for velocity in m / s and
#         convert input to float save result
velocity = float(input('Enter velocity in m / s: '))
print(velocity)
# Step 3: Calculate kinetic energy in Joules using formula
# Step 4: Display pretty result
```

Now when you run your code, this is the result:

```
Enter mass in kg: 97.13
97.13
Enter velocity in m / s: 14.5
14.5
```

Again, so far so good. Now it's time to perform the calculation of kinetic energy.

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and convert
#         input to float and save result
mass = float(input('Enter mass in kg: '))
print(mass)
# Step 2: Prompt user for velocity in m / s and
#         convert input to float save result
velocity = float(input('Enter velocity in m / s: '))
print(velocity)
# Step 3: Calculate kinetic energy in Joules using formula
kinetic_energy = 0.5 * mass * velocity ** 2
print(kinetic_energy)
```

```
# Step 4: Display pretty result
```

You run your code again, testing different values.

```
Enter mass in kg: 22.7
22.7
Enter velocity in m / s: 30.1
30.1
10283.213500000002
```

At this point, you decide that getting the input is working OK, so you remove the print statements following mass and velocity. Then you decide to focus on printing a pretty result. You know you want to use format specifiers, but you don't want to fuss with that quite yet, so you start with something simple (but not very pretty).

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and convert
#         input to float and save result
mass = float(input('Enter mass in kg: '))
# Step 2: Prompt user for velocity in m / s and
#         convert input to float save result
velocity = float(input('Enter velocity in m / s: '))
# Step 3: Calculate kinetic energy in Joules using formula
kinetic_energy = 0.5 * mass * velocity ** 2
# Step 4: Display pretty result
print(f'Mass = {mass} kg')
print(f'Velocity = {velocity} m / s')
print(f'Kinetic energy = {energy} Joules')
```

Now you run this and get an error.

```
Enter mass in kg: 17.92
Enter velocity in m / s: 25.0
Traceback (most recent call last):
  File "/blah/blah/kinetic_energy.py", line 10, in <module>
    print(f'Kinetic energy = {energy} Joules')
NameError: name 'energy' is not defined
```

You realize that you typed `energy` when you should have used `kinetic_energy`. That's not hard to fix, and since you know the other code is working OK you don't need to touch it.

Here's the fix:

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and convert input
#         to float and save result
mass = float(input('Enter mass in kg: '))
# Step 2: Prompt user for velocity in m / s and convert
#         input to float save result
velocity = float(input('Enter velocity in m / s: '))
# Step 3: Calculate kinetic energy in Joules using formula
kinetic_energy = 0.5 * mass * velocity ** 2
# Step 4: Display pretty result
print(f'Mass = {mass} kg')
print(f'velocity = {velocity} m / s')
print(f'kinetic energy = {kinetic_energy} Joules')
```

Now this runs without error.

```
Enter mass in kg: 22.901
Enter velocity in m / s: 13.33
Mass = 22.901 kg
Velocity = 13.33 m / s
Kinetic energy = 2034.6267494499998 Joules
```

The last step is to add format specifiers for pretty printing, but since everything else is working OK, the *only* thing you need to focus on are the format specifiers. Everything else is working!

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and convert input
#         to float and save result
mass = float(input('Enter mass in kg: '))
# Step 2: Prompt user for velocity in m / s and convert
#         input to float save result
velocity = float(input('Enter velocity in m / s: '))
# Step 3: Calculate kinetic energy in Joules using formula
kinetic_energy = 0.5 * mass * velocity ** 2
# Step 4: Display pretty result
print(f'Mass = {mass:.3f} kg')
print(f'velocity = {velocity:.3f} m / s')
print(f'kinetic energy = {kinetic_energy:.3f} Joules')
```

You test your code:

```
Enter mass in kg: 100
Enter velocity in m / s: 20
Mass = 100.000 kg
Velocity = 20.000 m / s
Kinetic energy = 20000.000 Joules
```

You decide that's OK, but you'd rather have comma separators in your output, so you modify the format specifiers.

```
"""
Kinetic Energy Calculator
"""

# Step 1: Prompt user for mass in kg and convert input
#         to float and save result
mass = float(input('Enter mass in kg: '))
# Step 2: Prompt user for velocity in m / s and convert
#         input to float save result
velocity = float(input('Enter velocity in m / s: '))
# Step 3: Calculate kinetic energy in Joules using formula
kinetic_energy = 0.5 * mass * velocity ** 2
# Step 4: Display pretty result
print(f'Mass = {mass:,.3f} kg')
print(f'velocity = {velocity:,.3f} m / s')
print(f'kinetic energy = {kinetic_energy:,.3f} Joules')
```

You test one more time and get a nice output.

```
Enter mass in kg: 72.1
Enter velocity in m / s: 19.5
Mass = 72.100 kg
Velocity = 19.500 m / s
Kinetic energy = 13,708.012 Joules
```

Looks great!

Now we can remove the comments we used as scaffolding, and we finish with:

```
"""
Kinetic Energy Calculator
"""

mass = float(input('Enter mass in kg: '))
velocity = float(input('Enter velocity in m / s: '))
kinetic_energy = 0.5 * mass * velocity ** 2
print(f'Mass = {mass:,.3f} kg')
print(f'velocity = {velocity:,.3f} m / s')
print(f'kinetic energy = {kinetic_energy:,.3f} Joules')
```

So now you've seen how to do incremental development.[3] Notice that we did not try to solve the entire problem all at once. We started with comments as placeholder / reminders, and then built up the program one step at a time, testing along the way. Using this approach can make the whole process easier by decomposing the problem into small, manageable, bite-sized (or should I say "byte-sized"?) chunks. That's incremental development.

9.4 Testing your code

It's important to test your code. In fact, one famous dictum of programming is:

> If it hasn't been tested, it's broken.

When writing code, try to anticipate odd or non-conforming input, and then test your program to see how it handles such input.

If your code has multiple branches, it's probably a good idea to test each branch. Obviously, with larger programs this could get unwieldy, but for small programs with few branches, it's not unreasonable to try each branch.

Some examples

Let's say we had written a program that is intended to take pressure in pounds per square inch (psi) and convert this to bars. A *bar* is a unit of pressure and 1 bar is equivalent to 14.503773773 psi.

Without looking at the code, let's test our program. Here are some values that represent reasonable inputs to the program.

input (in psi)	expected output (bars)	actual output (bars)
0	0	
14.503773773	1.0	
100.0	~ 6.894757293	

Here's our first test run.

```
Enter pressure in psi: 0
Traceback (most recent call last):
  ...
  File "/.../pressure.py", line 8, in psi_to_bars
    return PSI_PER_BAR / p
TypeError: unsupported operand type(s) for /: 'float' and 'str'
```

Oh dear! Already we have a problem. Looking at the last line of the error message we see

[3] If you're curious about how the pros do iterative and incremental development, see the Wikipedia article on iterative and incremental development: https://en.wikipedia.org/wiki/Iterative_and_incremental_development

```
TypeError: unsupported operand type(s) for /: 'float' and 'str'
```

What went wrong?

Obviously we're trying to do arithmetic—division—where one operand is a `float` and the other is a `str`. That's not allowed, hence the type error.

When we give this a little thought, we realize it's likely we didn't convert the user input to a `float` before attempting the calculation (remember, the `input()` function *always* returns a string).

We go back to our code and fix it so that the string we get from `input()` is converted to a `float` using the float constructor, `float()`. Having made this change, let's try the program again.

```
Enter pressure in psi: 0
Traceback (most recent call last):
  ...
  File "/.../pressure.py", line 8, in psi_to_bars
    return PSI_PER_BAR / p
ZeroDivisionError: float division by zero
```

Now we have a different error:

```
ZeroDivisionError: float division by zero
```

How could this have happened? Surely if pressure in psi is zero, then pressure in bars should also be zero (as in a perfect vacuum).

When we look at the code (you can see the offending line in the traceback above), we see that instead of taking the value in psi and dividing by the number of psi per bar, we've got our operands in the wrong order. Clearly we need to divide psi by psi per bar to get the correct result. Again you can see from the traceback, above, that there's a constant `PSI_PER_BAR`, so we'll just reverse the operands. This has the added benefit of having a non-zero constant in the denominator, so after this change, this operation can never result in a `ZeroDivisionError` ever again.

Now let's try it again.

```
Enter pressure in psi: 0
0.0 psi is equivalent to 0.0 bars.
```

That works! So far, so good.

Now let's try with a different value. We know, from the definition of *bar* that one bar is equivalent to 14.503773773 psi. Therefore, if we enter 14.503773773 for psi, the program should report that this is equivalent to 1.0 bar.

```
Enter pressure in psi: 14.503773773
14.503773773 psi is equivalent to 1.0 bars.
```

Brilliant.

Let's try a different value. How about 100? You can see in the table above that 100 psi is approximately equivalent to ~6.894757293 bars.

```
Enter pressure in psi: 100
100.0 psi is equivalent to 6.894757293178307 bars.
```

This looks correct, though we can see now that we're displaying more digits to the right of the decimal point than are useful.

Let's say we went back to our code and added format specifiers to that both psi and bars are displayed to four decimal places of precision.

```
Enter pressure in psi: 100
100.0000 psi is equivalent to 6.8948 bars.
```

This looks good.

Returning to our table, and filling in the actual values, now we have

input (in psi)	expected output (bars)	actual output (bars)
0	0.0	0.0000
14.503773773	1.0	1.0000
100.0	~ 6.894757293	6.8948

All our observed, actual outputs agree with our expected outputs.

What about negative values for pressure? Yes, there are cases where a negative pressure value makes sense. Take, for example, an isolation room for biomedical research. The air pressure in the isolation room should be lower than pressure in the outside hallways or adjoining rooms. In this way, when the door to an isolation room is opened, air will flow into the room, not out of it. This helps prevent contamination of uncontrolled outside environments. It's common to express the difference in pressure between the isolation room and the outside hallway as a negative value.

Does our program handle such values? Let's expand our table:

input (in psi)	expected output (bars)	actual output (bars)
0	0.0	0.0000
14.503773773	1.0	1.0000
100.0	~ 6.894757293	6.8948
-0.01	~ -0.000689476	??

Does our program handle this correctly?

```
Enter pressure in psi: -0.01
-0.0100 psi is equivalent to -0.0007 bars.
```

Again, this looks OK.

Now let's try to break our program to test its limits. Let's try some large values. The atmospheric pressure on the surface of Venus is 1334

psi. We'd expect a result in bars of approximately 91.9761 bars. The pressure at the bottom of the Mariana Trench in the Pacific Ocean is 15,750 psi, or roughly 1,086 bars.

input (in psi)	expected output (bars)	actual output (bars)
0	0.0	0.0000
14.503773773	1.0	1.0000
100.0	~ 6.894757293	6.8948
-0.01	~ -0.000689476	0.0007
1334	~ 91.9761	??
15,750	~ 1086	??

Let's test:

```
Enter pressure in psi: 1334
1334.0000 psi is equivalent to 91.9761 bars.
```

This one passes, but what about the next one (with the string containing the comma)? Will the conversion of the string `'15,750'` (notice the comma) be converted correctly to a float? Alas, this fails:

```
Traceback (most recent call last):
  File "/.../pressure.py", line 13, in <module>
    psi = float(input("Enter pressure in psi: "))
ValueError: could not convert string to float: '15,750'
```

Later, we'll learn how to create a modified copy of such a string with the commas removed, but for now let it suffice to say this can be fixed. Notice however, that if we hadn't checked this large value, which could reasonably be entered by a human user with the comma as shown, we might not have realized that this defect in our code existed! *Always test with as many ways the user might enter data as you can think of!*

With that fix in place, all is well.

```
Enter pressure in psi: 15,750
15750.0000 psi is equivalent to 1085.9243 bars.
```

By testing these larger values, we see that it might make sense to format the output to use commas as thousands separators for improved readability. Again, we might not have noticed this if we hadn't tested larger values. To fix this, we just change the format specifiers in our code.

```
Enter pressure in psi: 15,750
15,750.0000 psi is equivalent to 1,085.9243 bars.
```

Splendid.

This prompts another thought: what if the user entered psi in scientific notation like 1E3 for 1,000? It turns out that the float constructor handles inputs like this—but it never hurts to check!

Notice that by testing, we've been able to learn quite a bit about our code without actually reading the code! In fact, it's often the case that the job of writing tests for code falls to developers who aren't the ones writing the code that's being tested! One team of developers writes the code, a different team writes the tests for the code.

The important things we've learned here are:

- Work out in advance of testing (by using a calculator, hand calculation, or other method) what the *expected* output of your program should be on any given input. Then you can compare the expected value with the *actual* value and thus identify any discrepancies.
- Test your code with a wide range of values. In cases where inputs are numeric, test with extreme values.
- Don't forget how humans might enter input values. Different users might enter 1000 in different ways: 1000, 1000.0000, 1E3, 1,000, 1,000.0, *etc.* Equivalent values for inputs should always yield equivalent outputs!

Another example: grams to moles

If you've ever taken a chemistry course, you've converted grams to *moles*. A *mole* is a unit which measures quantity of a substance. One mole is equivalent to $6.02214076 \times 10^{23}$ *elementary entities*, where an elementary entity may be an atom, an ion, a molecule, *etc.* depending on context. For example, a reaction might yield so many grams of some substance, and by converting to moles, we know exactly how many entities this represents. In order to convert moles to grams, one needs the mass of the entities in question.

Here's an example. Our reaction has produced 75 grams of water, H_2O. Each water molecule contains two hydrogen atoms and one oxygen atom. The atomic mass of hydrogen is 1.008 grams per mole. The atomic mass of oxygen is 15.999 grams per mole. Accordingly, the molecular mass of one molecule of H_2O is

$$2 \times 1.008\ g/\text{mole} + 1 \times 15.999\ g/\text{mole} = 18.015\ g/\text{mole}.$$

Our program will require two inputs: grams, and grams per mole (for the substance in question). Our program should return the number of moles.

Let's build a table of inputs and outputs we can use to test our program.

grams	grams per mole	expected output (moles)	actual output (moles)
0	any	0	
75	18.015	~ 4.16319 E0	
245	16.043	~ 1.527240 E1	
3.544	314.469	~ 1.12698 E-2	
1,000	100.087	~ 9.99130 E0	

Let's test our program:

```
How many grams of stuff have you? 75
What is the atomic weight of your stuff? 18.015
You have 4.1632E+00 moles of stuff!
```

That checks out.

```
How many grams of stuff have you? 245
What is the atomic weight of your stuff? 16.043
You have 1.5271E+01 moles of stuff!
```

Keep checking...

```
How many grams of stuff have you? 3.544
What is the atomic weight of your stuff? 314.469
You have 1.1270E-02 moles of stuff!
```

Still good. Keep checking...

```
How many grams of stuff have you? 1,000
Traceback (most recent call last):
  File "/.../moles.py", line 9, in <module>
    grams = float(input("How many grams of stuff have you? "))
ValueError: could not convert string to float: '1,000'
```

Oops! This is the same problem we saw earlier: the float constructor doesn't handle numeric strings containing commas. Let's assume we've applied a similar fix and then test again.

```
How many grams of stuff have you? 1,000
What is the atomic weight of your stuff? 100.087
You have 9.9913E+00 moles of stuff!
```

Yay! Success!

Now, what happens if we were to test with negative values for either grams or atomic weight?

```
How many grams of stuff have you? -500
What is the atomic weight of your stuff? 42
You have -1.1905E+01 moles of stuff!
```

Nonsense! Ideally, our program should not accept negative values for grams, and should not accept negative values or zero for atomic weight.

In any event, you see now how useful testing a range of values can be. Don't let yourself be fooled into thinking your program is defect-free if you've not tested it with a sufficient variety of inputs.

9.5 The origin of the term "bug"

So where do we get the word "bug" anyhow? Alas, the origin of the term is lost. However, in 1947, the renowned computer pioneer Grace Murray Hopper was working on the Harvard Mark I computer, and a program was misbehaving.[4]

Figure 9.1: Grace Murray Hopper. Source: The Grace Murray Hopper Collection, Archives Center, National Museum of American History (image is in the public domain)

After reviewing the code and finding no error, she investigated further and found a moth in one of the computer's relays (remember this was back in the days when a computer filled an entire large room). The moth was removed, and taped into Hopper's lab notebook.

[4] Judging from Hopper's notebook (9 September 1947), the misbehaving program was a "multi-adder test". It appears they were running the machine through a sequence of tests—for example, tests for certain trigonometric functions took place earlier that day. At least one had failed and some relays (hardware components) were replaced. The multi-adder test was started at 3:25 PM (Hopper uses military time in the notebook: "1525"), and twenty minutes later, the moth was taped into the notebook. It's not clear how the problem became manifest, but someone went looking at the hardware and found the moth.

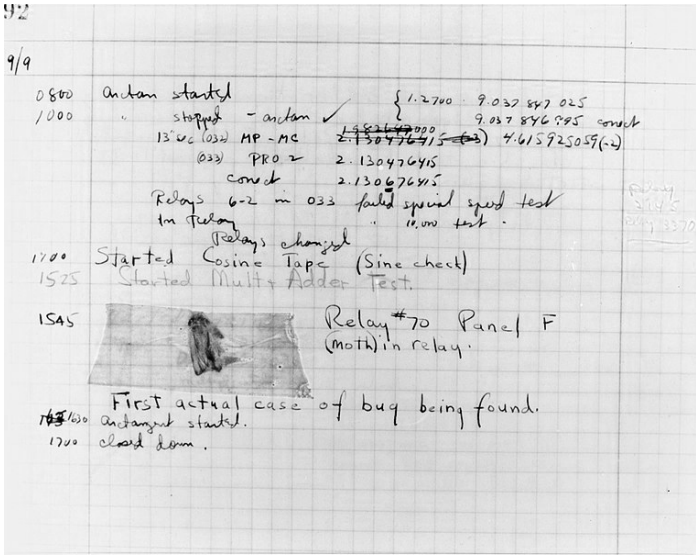


Figure 9.2: A page from Hopper's notebook containing the first "bug". Source: US Naval Historical Center Online Library (image is in public domain)

In interviews, Hopper said that after this discovery, whenever something was wrong she and her team would say "There must be a bug."

Not everyone likes the term "bug." For example, the famously grumpy Edsger Dijkstra thought that calling errors "bugs" was intellectually dishonest. He made this point in an essay with the remarkable title "On the cruelty of really teaching computing science."[5]

> We could, for instance, begin with cleaning up our language by no longer calling a bug a bug but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, *viz.* with the programmer who made the error. The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it disguises that the error is the programmer's own creation.

[5]Edsger Dijkstra, 1988, "On the cruelty of really teaching computing science". This essay is recommended. See the entry in the Edsger Dijkstra archive hosted by the University of Texas at Austin: https://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html

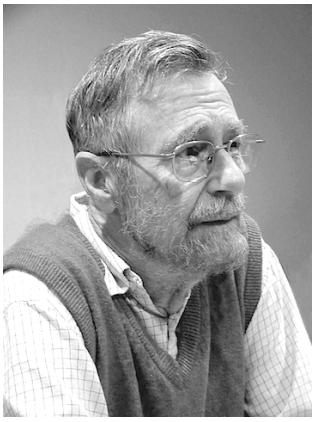


Figure 9.3: Edsger Dijkstra. Source: University of Texas at Austin, under a Creative Commons license

Despite Dijkstra's remonstrances, the term stuck. So now we have "bugs."

Bugs are, of course, inevitable. What's important is how we strive to avoid them and how we fix them when we find them.

9.6 Using assertions to test your code

Many languages, Python included, allow for *assertions* or *assert statements*. These are used to verify things you believe *should* be true about some condition or result. By making an assertion, you're saying "I believe x to be true", whatever x might be. Assertions are a powerful tool for verifying that a function or program actually does what you expect it to do.

Python provides a keyword, `assert` which can be used in *assert statements*. Here are some examples:

Let's say you have a function which takes a list of items for some purchase and applies sales tax. Whatever the subtotal might be, we know that the sales tax must be greater than or equal to zero. So we write an assertion:

```
sales_tax = calc_sales_tax(items)
assert sales_tax >= 0
```

If `sales_tax` is ever negative (which would be unexpected), this statement would raise an `AssertionError`, informing you that something you believed to be true, was not, in fact, true. This is roughly equivalent to

```
if sales_tax < 0:
    raise AssertionError
```

but is more concise and readable.

Notice that if the assertion holds, no exception is raised, and the execution of your code continues uninterrupted.

Here's another example:

```
def calc_hypotenuse(a, b):
    """Given two legs of a right triangle, return the
    length of the hypotenuse. """
    assert a >= 0
    assert b >= 0

    return math.sqrt(a ** 2 + b ** 2)
```

What's going on here? This isn't data validation. Rather, we're documenting conditions that must hold for the function to return a valid result, and we ensure that the program will fail if these conditions aren't met. We could have a *degenerate triangle*, where one or both legs have length zero, but it cannot be the case that either leg has negative length. This approach has the added benefit of reminding the programmer what conditions must hold in order to ensure correct behavior.

Judicious use of assertions can help you write correct, robust code.

Adding friendly messages

Python's `assert` allows you to provide a custom message in the event of an `AssertionError`. The syntax is simple,

```
assert 1 + 1 == 2, "Something is horribly wrong!"
```

Some caveats

It's important to understand that `assert` is a Python keyword and *not* the name of a built-in function. This is correct:

```
assert 0.0 <= x <= 1.0, "x must be in [0.0, 1.0]"
```

but this is not

```
assert(0.0 <= x <= 1.0, "x must be in [0.0, 1.0]")
```

Why? This will treat the tuple

```
(0.0 <= x <= 1.0, "x must be in [0.0, 1.0]")
```

as what is being asserted. But non-empty tuples are truthy, and so this will *never* result in an `AssertionError`, no matter what the value of `x`!

Let's test it

```
>>> x = -42
>>> assert(0.0 <= x <= 1.0, "x must be in [0.0, 1.0]")
<stdin>:1: SyntaxWarning: assertion is always true,
        perhaps remove parentheses?
>>>
```

However, this works as intended

```
>>> x = -42
>>> assert 0.0 <= x <= 1.0, "x must be in [0.0, 1.0]"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: x must be in [0.0, 1.0]
>>>
```

and if `x` is in the indicated interval, all is well.

```
>>> x = 0.42
>>> assert 0.0 <= x <= 1.0, "x must be in [0.0, 1.0]"
>>>
```

(Notice the `>>>` at the end of the snippet above, indicating that the assertion has passed and is thus silent.)

You should try adding assertions to your code. In fact, the NASA/JPL Laboratory for Reliable Software published a set of guidelines for producing reliable code, and one of these is "Use a minimum of two runtime assertions per function."[6]

9.7 Rubberducking

"Rubberducking"? What on earth is "rubberducking"? Don't laugh: rubberducking is one of the most powerful debugging tools in the known universe! Many programmers keep a little rubber duck handy on their desk, in case of debugging emergencies.

Here's how it works. If you get stuck, and cannot solve a particular problem or cannot fix a pesky bug you *talk to the duck*. Now, rubber ducks aren't terribly sophisticated, so you have to explain things to them in the simplest possible terms. Explain your problem to the duck using as little computer jargon as you can. Talk to your duck as if it were an intelligent five-year-old. You'd be amazed at how many problems can be solved this way!

Why does it work?

First, by talking to your duck, you step outside your code for a while. You're talking about your code without having to type at the keyboard, and without getting bogged down in the details of syntax. You're talking about what you think your code should be doing.

Second, your duck will never judge you. It will remain silent while you do your best to explain. Ducks are amazing listeners!

[6] G.J. Holzmann, 2006, "The Power of 10: Rules for Developing Safety-Critical Code", *IEEE Computer*, 39(6). doi:10.1109/MC.2006.212.

It's very often the case that while you're explaining your troubles to the duck, or describing what you think your code *should* be doing, that you reach a moment of realization. By talking through the problem you arrive at a solution or you recognize where you went wrong.

What if I don't have a rubber duck?

That's OK. Many other things can stand in for a duck if need be. Do you have a stuffed animal? a figurine of any kind? a photograph of a friend? a roommate with noise-cancelling headphones? Any of these can be substituted for a duck if need be.

The important thing is that you take your hands off the keyboard, and maybe even look away from your code, and describe your problem in simple terms.

Trust the process! It works!

9.8 Exceptions

`AssertionError`

As we've seen, if an assertion passes, code execution continues normally. However, if an assertion fails, an `AssertionError` is raised. This indicates that what has been asserted has evaluated to `False`.

If you write an assertion, and when you test your code an `AssertionError` is raised, then you should do two things:

1. Make sure that the assertion you've written is correct. That is, you are asserting some condition is true when it should, in fact, be true.
2. If you've verified that your assertion statement(s) are correct, and an `AssertionError` continues to be raised, then it's time to debug your code. Continue updating and testing until the issue is resolved.

9.9 Exercises

Exercise 01

Arrange the following code and add any missing elements so that it follows the stated guidelines for program structure (as per section 9.2):

```
x = float(input("Enter a value for x: "))

def square(x_):
    return x_ * x_

x_sqrd = square(x)
print(f"{x} squared is {x_sqrd}.")
```

Exercise 02

Write a complete program which prompts the user for two integers (one at a time) and then prints the sum of the integers. Be sure to follow the stated guidelines for program structure.

Exercise 03

Egbert has written a function which takes two arguments, both representing angles in degrees. The function returns the sum of the two degrees, modulo 360.

Here's an example of one test of this function:

```
>>> sum_angles(180, 270)
90.0
```

What other values might you use to test such a function? For each pair of values you choose, give the expected output of the function. (See section 9.3)

Exercise 04

Consider this module (program):

```
"""
A simple program
"""

def cube(x_):
    return x_ ** 3

# Test function to make sure it works
# as intended

assert cube(3) == 27
assert cube(0) == 0
assert cube(-1) == -1

# Allow for other test at user's discretion

x = float(input("Enter a number: "))
print(f"The cube of {x} is {cube(x)}.")
```

a. What happens if we import this module?

b. What undesirable behavior occurs on import, and how can we fix it?

Exercise 05

What's wrong with these assertions and how would you fix them?

a.

```
assert 1 + 1 = 5, "I must not understand addition!"
```

b.

```
n = int(input("Enter an integer: "))
assert (n + n == 2 * n + 1, "Arithmetic error!")
```

Hint: Try these out in the Python shell.

Exercise 06

Write a program with a function that takes three integers as arguments and returns their sum. Comment first, then write your code.

Chapter 10

Sequences

> Order is heaven's first law.
>
> –Alexander Pope

> A sequence works in a way a collection never can.
>
> –George Murray

In this chapter, we'll introduce two new types, `lists` and `tuples`. These are fundamental data structures that can be used to store, retrieve, and, in some contexts, manipulate data. We sometimes refer to these as "sequences" since they carry with them the concepts of order and sequence. Strings (`str`) are also sequences. _introduction.qmd

Learning objectives

- You will learn about lists, and about common list operations, including adding and removing elements, finding the number of elements in a list, checking to see if a list contains a certain value, *etc.*
- You will learn that lists are *mutable*. This means that you can modify a list after it's created. We can append items, remove items, change the values of individual items and more.
- You will learn about tuples. Tuples are unlike lists in that they are immutable—they cannot be changed.
- You will learn that strings are sequences.
- You will learn how to use indices to retrieve individual values from these structures.

In the next chapter, we'll learn how to iterate over the elements of a sequence.

Terms introduced

- element
- `global` keyword

- immutable
- testing for membership with `in`
- index
- lexicographic order
- list
- list methods `.append()`, `.pop()`, and `.sort()`
- mutable
- membership
- sequence methods `.count()` and `.index()`
- sequence unpacking
- slicing
- `sorted()`
- string methods `.split()`, `.join()`, and `.replace()`
- tuple
- zero indexing

10.1 Lists

The *list* is one of the most widely used data structures in Python. One could not enumerate all possible applications of lists.[1] Lists are ubiquitous, and you'll see they come in handy!

What is a list?

A **list** is a *mutable sequence of objects*. That sounds like a mouthful, but it's not that complicated. If something is **mutable**, that means that it can change (as opposed to *immutable*, which means it cannot). A *sequence* is an *ordered collection*—that is, each element in the collection has its own place in some ordering.

For example, we might represent customers queued up in a coffee shop with a list. The list can change—new people can get in the coffee shop queue, and the people at the front of the queue are served and they leave. So the queue at the coffee shop is *mutable*. It's also *ordered*—each customer has a place in the queue, and we could assign a number to each position. This is known as an *index*.

How to write a list in Python

The syntax for writing a list in Python is simple: we include the objects we want in our list within square brackets. Here are some examples of lists:

```
coffee_shop_queue = ['Brian', 'Egbert', 'Jason', 'Lisa',
                     'Jim', 'Jackie', 'Sami']
scores = [74, 82, 78, 99, 83, 91, 77, 98, 74, 87]
```

We separate elements in a list with commas.

[1] If you've programmed in another language before, you may have come to know similar data structures, for example, `ArrayList` in Java, mutable vectors in C++, *etc.* However, there are many differences, so keep that in mind.

Unlike many other languages, the elements of a Python list needn't all be of the same type. So this is a perfectly valid list:

```
mixed = ['cheese', 0.1, 5, True]
```

There are other ways of creating lists in Python, but this will suffice for now.

At the Python shell, we can display a list by giving its name.

```
>>> mixed = ['cheese', 0.1, 5, True]
>>> mixed
['cheese', 0.1, 5, True]
```

The empty list

Is it possible for a list to have no elements? Yup, and we call that the *empty list*.

```
>>> aint_nothing_here = []
>>> aint_nothing_here
[]
```

Accessing individual elements in a list

As noted above, lists are *ordered*. This allows us to access individual elements within a list using an *index*. An **index** is just a number that corresponds to an element's position within a list. The only twist is that in Python, and most programming languages, indices start with zero rather than one.[2] So the first element in a list has index 0, the second has index 1, and so on. Given a list of n elements, the indices into the list will be integers in the interval $[0, n-1]$.

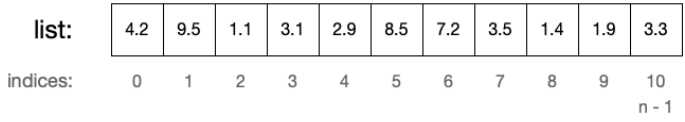


Figure 10.1: A list and its indices

In the figure (above) we depict a list of floats of size eleven—that is, there are eleven elements in the list. Indices are shown below the list, with each index value associated with a given element in the list. Notice that with a list of eleven elements, indices are integers in the interval $[0, 10]$.

[2]Some languages are *one-indexed*, meaning that their indices start at one, but these are in the minority. *One-indexed* languages include Cobol, Fortran, Julia, Matlab, R, and Lua.

Let's turn this into a concrete example:

```
>>> data = [4.2, 9.5, 1.1, 3.1, 2.9, 8.5, 7.2, 3.5, 1.4, 1.9, 3.3]
```

Now let's access individual elements of the list. For this, we give the name of the list followed immediately by the index enclosed in brackets:

```
>>> data[0]
4.2
```

The element in the list `data`, at index 0, has a value of 4.2. We can access other elements similarly.

```
>>> data[1]
9.5
>>> data[9]
1.9
```

`IndexError`

Let's say we have a list with n elements. What happens if we try to access a list using an index that doesn't exist, say index n or index $n + 1$?

```
>>> foo = [2, 4, 6]
>>> foo[3]   # there is no element at index 3!!!
Traceback (most recent call last):
  File "/.../code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

This `IndexError` message is telling us there *is no element with index 3*.

Changing the values of individual elements in a list

We can use the index to access individual elements in the list for modification as well (remember: lists are mutable).

Let's say there was an error in data collection, and we wanted to change the value at index 7 from 3.5 to 6.1. To do this, we put the list and index on the left side of an assignment.

```
>>> data
[4.2, 9.5, 1.1, 3.1, 2.9, 8.5, 7.2, 3.5, 1.4, 1.9, 3.3]
>>> data[7] = 6.1
>>> data
[4.2, 9.5, 1.1, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
```

Let's do another: We'll change the element at index 2 to 4.7.

```
>>> data[2] = 4.7
>>> data
[4.2, 9.5, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
```

Some convenient built-in functions that work with lists

Python provides many tools and built-in functions that work with lists (and tuples, which we'll see soon). Here are a few such built-in functions:

	description	constraint(s) if any	example
`sum()`	calculates sum of elements	values must be numeric *	`sum(data)`
`len()`	returns number of elements	applies to sequences	`len(data)`
`max()`	returns largest value	elements must be comparable	`max(data)`
`min()`	returns smallest value	elements must be comparable	`min(data)`

* In the context of `sum()`, `max()`, and `min()`, Boolean `True` is treated as 1 and Boolean `False` is treated as 0.

Using our example `data` (from above):

```
>>> sum(data)
52.8
>>> len(data)
11
>>> max(data)
9.5
>>> min(data)
1.4
```

It seems natural at this point to ask, can I calculate the average (mean) of the values in a list? If the list contains only numeric values, the answer is "yes," but Python doesn't supply a built-in for this. However, the solution is straightforward.

```
>>> sum(data) / len(data)
4.8
```

…and there's our mean!

Some convenient list methods

We've seen already that string objects have methods associated with them. For example, `.upper()`, `.lower()`, and `.capitalize()`. Recall that methods are just functions associated with objects of a given type, which operate on the object's data (value or values).

Lists also have handy methods which operate on a list's data. Here are a few:

	description	constraint(s)
`.sort()`	sorts list	elements must be comparable
`.append(x)`	appends `x` to list	must supply element to append
`.pop()`	pops the last element off list and returns its value	cannot pop from empty list
`.pop(i)`	removes the element at index i and returns its value	must be valid index (int)

There are many others, but let's start with these.

Appending an element to a list

To append an element to a list, we use the `.append()` method, where `x` is the element we wish to append.

```
>>> data
[4.2, 9.5, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
>>> data.append(5.9)
>>> data
[4.2, 9.5, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3, 5.9]
```

By using the `.append()` method, we've appended the value 5.9 to the end of the list.

"Popping" elements from a list

We can remove (pop) elements from a list using the `.pop()` method. If we call `.pop()` without an argument, Python will remove the last element in the list and return its value.

```
>>> data.pop()
5.9
>>> data
[4.2, 9.5, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
```

Notice that the value 5.9 is returned, and that the last element in the list (5.9) has been removed.

Sometimes we wish to pop an element from a list that doesn't happen to be the last element in the list. For this we can supply an index, `.pop(i)`, where `i` is the index of the element we wish to pop.

```
>>> data.pop(1)
9.5
>>> data
[4.2, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
```

For reasons which may be obvious, we cannot `.pop()` from an empty list, and we cannot `.pop(i)` if the index `i` does not exist.

Sorting a list in place

Now let's look at `.sort()`. *In place* means that the list is modified right where it is, and there's no list returned from `.sort()`. This means that calling `.sort()` *alters the list*!

```
>>> data
[4.2, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
>>> data.sort()
>>> data
[1.4, 1.9, 2.9, 3.1, 3.3, 4.2, 4.7, 6.1, 7.2, 8.5]
```

This is *unlike* the string methods like `.lower()` which return an altered *copy* of the string. Why is this? Strings are immutable; lists are mutable.

Because `.sort()` sorts a list *in place*, it returns `None`. So don't think you can work with the return value of `.sort()` because there isn't any! Example:

```
>>> m = [5, 7, 1, 3, 8, 2]
>>> n = m.sort()
>>> n
>>> type(n)
<class 'NoneType'>
```

Some things you might not expect

Lists behave differently from many other objects when performing assignment. Let's say you wanted to preserve your data "as-is" but also have a sorted version. You might think that this would do the trick.

```
>>> data = [4.2, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
>>> copy_of_data = data   # naively thinking you're making a copy
>>> data.sort()
>>> data
[1.4, 1.9, 2.9, 3.1, 3.3, 4.2, 4.7, 6.1, 7.2, 8.5]
```

But now look what happens when we inspect `copy_of_data`.

```
>>> copy_of_data
[1.4, 1.9, 2.9, 3.1, 3.3, 4.2, 4.7, 6.1, 7.2, 8.5]
```

Wait! What? How did that happen?

When we made the assignment `copy_of_data = data` we assumed (quite reasonably) that we were making a copy of our data. It turns out this is not so. What we wound up with was *two names for the same underlying data structure*, `data` and `copy_of_data`. This is the way things work with mutable objects (like lists).[3] Let's verify this using the Python keyword `is`.

```
>>> data = [4.2, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
>>> copy_of_data = data
>>> copy_of_data is data
True
```

There you have it: `data` and `copy_of_data` are one and the same.

So how do we get a copy of our list? One way is to use the `.copy()` method.[4] This will return a copy of the list, so we have two different list instances.[5]

```
>>> data = [4.2, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
>>> copy_of_data = data.copy()   # call the copy method
>>> data.sort()
>>> data
[1.4, 1.9, 2.9, 3.1, 3.3, 4.2, 4.7, 6.1, 7.2, 8.5]
>>> copy_of_data
[4.2, 4.7, 3.1, 2.9, 8.5, 7.2, 6.1, 1.4, 1.9, 3.3]
```

A neat trick to get the last element of a list

Let's say we have a list, and don't know how many elements are in it. Let's say we want the *last* element in the list. How might we go about it?

We could take a brute force approach. Say our list is called `x`.

```
>>> x[len(x) - 1]
```

Let's unpack that. Within the brackets we have the expression `len(x) - 1`. `len(x)` returns the number of elements in the list, and then we subtract 1 to adjust for zero-indexing (if we have n elements in a list,

[3] The reasons for this state of affairs is beyond the scope of this text. However, if you're curious, see: https://docs.python.org/3/library/copy.html.

[4] There are other approaches to creating a copy of a list, specifically using the list constructor or slicing with `[:]`, but we'll leave these for another time. However, slicing is slower than the other two. *Source*: I timed it.

[5] Actually it makes what's called a *shallow* copy. See: https://docs.python.org/3/library/copy.html.

the index of the last element is $n - 1$). So that works, but it's a little clunky. Fortunately, Python allows us to get the last element of a list with an index of -1.

```
>>> x[-1]
```

You may think of this as counting backward through the indices of the list.

A puzzle (optional)

Say we have some list `x` (as above), and we're intrigued by this idea of counting backward through a list, and we want to find an alternative way to access the first element of *any* list of any size with a negative-valued index. Is this possible? Can you write a solution that works for any list `x`?

10.2 Tuples

A tuple? What's a tuple? A **tuple** is an *immutable sequence* of objects.

Like lists they allow for indexed access of elements. Like lists they may contain any arbitrary type of Python object (int, float, bool, str, *etc.*). Unlike lists they are immutable, meaning that once created they cannot change. You'll see that this property can be desirable in certain contexts.

How do we write a tuple?

The crucial thing in writing a tuple is commas—we separate elements of a tuple with commas—but it's conventional to write them with parentheses as well.

Here are some examples:

```
>>> coordinates = 0.378, 0.911
>>> coordinates
(0.378, 0.911)
>>> coordinates = (1.452, 0.872)
>>> coordinates
(1.452, 0.872)
```

We can create a tuple with a single element with a comma, with or without parentheses.

```
>>> singleton = 5,
>>> singleton
(5,)
>>> singleton = ('Hovercraft',)
>>> singleton
('Hovercraft',)
```

Notice that it's the comma that's crucial.

```
>>> (5)
5
>>> ('Hovercraft')
'Hovercraft'
```

We can create an empty tuple, thus:

```
>>> empty = ()
>>> empty
()
```

In this case, no comma is needed.

Accessing elements in a tuple

As with lists, we can access the elements of a tuple with integer indices.

```
>>> t = ('cheese', 42, True, -1.0)
>>> t[0]
'cheese'
>>> t[1]
42
>>> t[2]
True
>>> t[3]
-1.0
```

Just like lists, we can access the last element by providing -1 as an index.

```
>>> t[-1]
-1.0
```

Finding the number of elements in a tuple

As with lists, we can get the number of elements in a tuple with `len()`.

```
>>> t = ('cheese', 42, True, -1.0)
>>> len(t)
4
```

Why would we use tuples instead of lists?

First, there are cases in which we want immutability. Lists are a dynamic data structure. Tuples on the other hand are well-suited to static data.

As one example, say we're doing some geospatial tracking or analysis. We might use tuples to hold the coordinates of some location—the latitude and longitude. A tuple is appropriate in this case.

```
>>> (44.4783021, -73.1985849)
```

Clearly a list is not appropriate: we'd never want to append elements or remove elements from latitude and longitude, and coordinates like this belong together—they form a pair.

Another case would be records retrieved from some kind of database.

```
>>> student_record = ('Porcupine', 'Egbert', 'eporcupi@uvm.edu',
....                  3.21, 'sophomore')
```

Another reason that tuples are preferred in many contexts is that creating a tuple is more efficient than creating a list. So, for example, if you're reading many records from a database into Python objects, using tuples will be faster. However, the difference is small, and could only become a factor if handling many records (think millions).

You say tuples are immutable. Prove it.

Just try modifying a tuple. Say we have the tuple `(1, 2, 3)`. We can read individual elements from the tuple just like we can for lists. But unlike lists we can't use the same approach to assign a new value to an element of the tuple.

```
>>> t = (1, 2, 3)
>>> t[0]
1
>>> t[0] = 51
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

There you have it: " 'tuple' object does not support item assignment."

What about this?

```
>>> t = (1, 2, 3)
>>> t = ('a', 'b', 'c')
```

"There!" you say, "I've changed the tuple!" No, you haven't. What's happened here is that you've created a new tuple, and given it the same name `t`. In doing so, you've destroyed the first variable and created a new. Let's verify this:

```
>>> t = (1, 2, 3)
>>> id(t)
4375862208
>>> t = ('a', 'b', 'c')
>>> id(t)
4375920896
```

Two different ids means two different objects. The object first named `t` is not the same object as the second object named `t`.

What about a tuple that contains a list?

Tuples can contain any type of Python object—even lists. This is valid:

```
>>> t = ([1, 2, 3],)
>>> t
([1, 2, 3],)
```

Now let's modify the list.

```
>>> t[0][0] = 5
>>> t
([5, 2, 3],)
```

Haven't we just modified the tuple? Actually, no. The tuple contains the list (which is mutable). So we can modify the list within the tuple, but we can't replace the list with another list.

```
>>> t = ([1, 2, 3],)
>>> new_list = [4, 5, 6]
>>> t[0] = new_list
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Again, the tuple is unchanged.

You may ask: What's up with the two indices? Say we have a list within a tuple. The list has an index within the tuple, and the elements of the list have their indices within the list. So the first index is used to retrieve the list from within the tuple, and the second is used to retrieve the element from the list.

```
>>> t = (['a', 'b', 'c'],)
>>> t[0]
['a', 'b', 'c']
>>> t[0][0]
'a'
>>> t[0][1]
'b'
```

```
>>> t[0][2]
'c'
```

10.3 Mutability and immutability

Mutability and immutability are properties of certain classes of object. For example, these are immutable—once created they cannot be changed:

- numeric types (int and float)
- Booleans
- strings
- tuples

However, lists are mutable. Later, we'll see another mutable data structure, the dictionary.

Immutable objects

You may ask what's been happening in cases like this:

```
>>> x = 75021
>>> x
75021
>>> x = 61995
>>> x
61995
```

Aren't we changing the value of `x`? While we might speak this way casually, what's really going on here is that we're creating a new int, `x`.

Here's how we can demonstrate this—using Python's built-in function `id()`.[6]

```
>>> x = 75021
>>> id(x)
4386586928
>>> x = 61995
>>> id(x)
4386586960
```

See? The IDs have changed.

The IDs you'll see if you try this on your computer will no doubt be different. But you get the idea: different IDs mean we have two different objects!

Same goes for strings, another immutable type.

[6] While using `id()` is fine for tinkering around in the Python shell, this is the only place it should be used. Never include `id()` in any programs you write. The Python documentation states that `id()` returns "the 'identity' of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value." So please keep this in mind.

```
>>> s = 'Pharoah Sanders'  # who passed away the day I wrote this
>>> id(s)
4412581232
>>> s = 'Sal Nistico'
>>> id(s)
4412574640
```

Same goes for tuples, another immutable type.

```
>>> t = ('a', 'b', 'c')
>>> id(t)
4412469504
>>> t[0] = 'z'   # Try to change an element of t...
Traceback (most recent call last):
  File "/Library/.../code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> id(t)   # still the same object
4412469504
>>> t = ('z', 'y', 'x')
>>> id(t)
4412558784
```

Mutable objects

Now let's see what happens in the case of a list. Lists are mutable.

```
>>> parts = ['rim', 'hub', 'spokes']
>>> id(parts)
4412569472
>>> parts.append('inner tube')
>>> parts
['rim', 'hub', 'spokes', 'inner tube']
>>> id(parts)
4412569472
>>> parts.pop(0)
'rim'
>>> parts
['hub', 'spokes', 'inner tube']
>>> id(parts)
4412569472
```

See? We make changes to the list and the ID remains unchanged. It's the same object throughout!

Variables, names, and mutability

Assignment in Python is all about *names*, and it's important to understand that when we make assignments we are *not* copying values from

one variable to another. This becomes most clear when we examine the behavior with respect to mutable objects (for example, lists):

```
>>> lst_a = [1, 2, 3, 4, 5]
>>> lst_b = lst_a
```

Now let's change `lst_a`.

```
>>> lst_a.append(6)
>>> lst_b
[1, 2, 3, 4, 5, 6]
```

See? `lst_b` isn't a copy of `lst_a`, it's a different name for the same object!

If a mutable value has more than one name, if we affect some change in the value via one name, all the other names still refer to the mutated value.

Now, what do you think about this example:

```
>>> lst_a = [1, 2, 3, 4, 5]
>>> lst_b = [1, 2, 3, 4, 5]
>>> lst_a.append(6)
```

Are `lst_a` and `lst_b` different names for the same object? Or do they refer to different objects?

```
>>> lst_a
[1, 2, 3, 4, 5, 6]
>>> lst_b
[1, 2, 3, 4, 5]
```

`lst_a` and `lst_b` are names for different objects! Now, does this mean that assignment works differently for mutable and immutable objects? Not at all.

Then why, you may ask, when we assign 1 to `x` and 1 to `y` do both names refer to the same value, whereas when we assign `[1, 2, 3, 4, 5]` to `lst_a` and `[1, 2, 3, 4, 5]` to `lst_b` we have two different lists?

Let's say you and a friend wrote down lists of the three greatest baseball teams of all time. Furthermore, let's say your lists were identical…

Trigger warning: opinions about MLB teams follow!

```
>>> my_list = ['Cubs', 'Tigers', 'Dodgers']
>>> janes_list = ['Cubs', 'Tigers', 'Dodgers']
```

Now, my list is my list, and Jane's list is Jane's list. These are two different lists.

Let's say that the Dodgers fell out of favor with Jane, and she replaced them with the Cardinals (abhorrent, yes, I know).

```
>>> janes_list.pop()
'Dodgers'
>>> janes_list.append('Cardinals')
>>> janes_list
['Cubs', 'Tigers', 'Cardinals']
>>> my_list
['Cubs', 'Tigers', 'Dodgers']
```

That makes sense, right? Even though the lists started with identical elements, they're still two different lists and mutating one does not mutate the other.

But be aware that we can give two different names to the same mutable object (as shown above).

```
>>> lst_a = [1, 2, 3, 4, 5]
>>> lst_b = lst_a
>>> lst_a.append(6)
>>> lst_a
[1, 2, 3, 4, 5, 6]
>>> lst_b
[1, 2, 3, 4, 5, 6]
```

This latter case is relevant when we pass a list to a function. We may think we're making a copy of the list, when in fact, we're only giving it another name. This can result in unexpected behavior—we think we're modifying a copy of a list, when we're actually modifying the list under another name!

10.4 Subscripts are indices

Here we make explicit the connection between subscript notation in mathematics and indices in Python.

In mathematics: Say we have a collection of objects X. We can refer to individual elements of the collection by associating each element of the collection with some index from the natural numbers. Thus,

$$\begin{gathered} x_0 \in X \\ x_1 \in X \\ \dots \\ x_n \in X \end{gathered}$$

Different texts may use different starting indices. For example, a linear algebra text probably starts indices at one. A text on set theory is likely to use indices starting at zero.

In Python, sequences—lists, tuples, and strings—are indexed in this fashion. All Python indices start at zero, and we refer to Python as being *zero indexed*.

Indexing works the same for lists, tuples, and even strings. Remember that these are sequences—ordered collections—so each element has an index, and we may access elements within the sequence by its index.

```
my_list = ['P', 'O', 'R', 'C', 'U', 'P', 'I', 'N', 'E']
```

0	1	2	3	4	5	6	7	8
P	O	R	C	U	P	I	N	E

We start indices at zero, and for a list of length n, the indices range from zero to $n - 1$.

It's exactly the same for tuples.

```
my_tuple = ('P', 'O', 'R', 'C', 'U', 'P', 'I', 'N', 'E')
```

0	1	2	3	4	5	6	7	8
P	O	R	C	U	P	I	N	E

The picture looks the same, doesn't it? That's because it is! It's even the same for strings.

```
my_string = 'PORCUPINE'
```

0	1	2	3	4	5	6	7	8
P	O	R	C	U	P	I	N	E

While we don't explicitly separate the characters of a string with commas, they are a sequence nonetheless, and we can read characters by index.

10.5 Concatenating lists and tuples

Sometimes we have two or more lists or tuples, and we want to combine them. We've already seen how we can concatenate strings using the + operator. This works for lists and tuples too!

```
>>> plain_colors = ['red', 'green', 'blue', 'yellow']
>>> fancy_colors = ['ultramarine', 'ochre', 'indigo', 'viridian']
>>> all_colors = plain_colors + fancy_colors
>>> all_colors
['red', 'green', 'blue', 'yellow', 'ultramarine', 'ochre',
 'indigo', 'viridian']
```

or

```
>>> plain_colors = ('red', 'green', 'blue', 'yellow')
>>> fancy_colors = ('ultramarine', 'ochre', 'indigo', 'viridian')
>>> all_colors = plain_colors + fancy_colors
>>> all_colors
('red', 'green', 'blue', 'yellow', 'ultramarine', 'ochre',
 'indigo', 'viridian')
```

This works just like coupling railroad cars. Coupling two trains with multiple cars preserves the ordering of the cars.

Answering the inevitable question: Can we concatenate a list with a tuple using the + operator? No, we cannot.

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [1, 2, 3] + (4, 5, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "tuple") to list
```

10.6 Copying lists

We've seen elsewhere that the following simply gives another name to a list.

```
>>> lst_1 = ['gamma', 'epsilon', 'delta', 'alpha', 'beta']
>>> lst_2 = lst_1
>>> lst_1.sort()
>>> lst_2
['alpha', 'beta', 'delta', 'epsilon', 'gamma']
```

However, there are times when we really mean to make a copy. The `.copy()` method returns a *shallow copy* of a list.

```
>>> lst_1 = ['gamma', 'epsilon', 'delta', 'alpha', 'beta']
>>> lst_2 = lst_1.copy()
>>> lst_1.sort()
>>> lst_2
['gamma', 'epsilon', 'delta', 'alpha', 'beta']
```

We can copy a list using a slice.

```
>>> lst_1 = ['gamma', 'epsilon', 'delta', 'alpha', 'beta']
>>> lst_2 = lst_1[:]  # slice
>>> lst_1.sort()
>>> lst_2
['gamma', 'epsilon', 'delta', 'alpha', 'beta']
```

There's another way we can copy a list: using the *list constructor*. The list constructor takes some iterable and iterates it, producing a new list composed of the elements yielded by iteration. Since lists are iterable, we can use this to create a copy of a list.

```
>>> lst_1 = ['gamma', 'epsilon', 'delta', 'alpha', 'beta']
>>> lst_2 = list(lst_1)  # using the list constructor
>>> lst_1.sort()
>>> lst_2
['gamma', 'epsilon', 'delta', 'alpha', 'beta']
```

So now we have *three* ways to make a copy of a list:

- By using the `.copy()` method
- By slicing (`lst_2 = lst_1[:]`)
- By using the list constructor (`lst_2 = list(lst_1)`)

Fun fact: Under the hood, `.copy()` simply calls the list constructor to make a new list.

10.7 Finding an element within a sequence

It should come as no surprise that if we have a sequence of objects, we often wish to see if an element is in the sequence (list, tuple, or string). Sometimes we also want to find the *index* of the element within a sequence. Python makes this relatively straightforward.

Checking to see if an element is in a sequence

Say we have the following list:

```
>>> fruits = ['kumquat', 'papaya', 'kiwi', 'lemon', 'lychee']
```

We can check to see if an element exists using the Python keyword `in`.

```
>>> 'kiwi' in fruits
True
>>> 'apple' in fruits
False
```

We can use the evaluation of such expressions in conditions:

```
>>> if 'apple' in fruits:
...    print("Let's bake a pie!")
... else:
...    print("Oops. No apples.")
...
Oops. No apples.
```

or

```
>>> if 'kiwi' in fruits:
...    print("Let's bake kiwi tarts!")
... else:
...    print("Oops. No kiwis.")
...
Let's bake kiwi tarts!
```

This works the same with numbers or with mixed-type lists.

```
>>> some_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> 5 in some_primes
True
>>> 4 in some_primes
False
```

or

```
>>> mixed = (42, True, 'tobacconist', 3.1415926)
>>> 42 in mixed
True
>>> -5 in mixed
False
```

We can also check to see if some substring is within a string.

```
>>> "quick" in "The quick brown fox..."
True
```

So, we can see that the Python keyword `in` can come in very handy in a variety of ways.

Getting the index of an element in a sequence

Sometimes we want to know the index of an element in a sequence.

For this we use `.index()` method. This method takes some value as an argument and returns the index of the first occurrence of that element in the sequence (if found).

```
>>> fruits = ['kumquat', 'papaya', 'kiwi', 'lemon', 'lychee']
>>> fruits.index('lychee')
4
```

However, this one can bite. If the element is *not* in the list, Python will raise a ValueError exception.

```
>>> fruits.index('frog')
Traceback (most recent call last):
  File "/Library/Frameworks/.../code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
ValueError: 'frog' is not in list
```

This is rather inconvenient, since if this were to occur when running your program, it would crash your program! Yikes! So what can be done? Later on in this textbook we'll learn about *exception handling*, but for now, here's a different solution: just check to see if the element is in the list (or other sequence) first by using an `if` statement, and *then* get the index if it is indeed in the list.

```
>>> if 'frog' in fruits:
...     print(f"The index of frog in fruits is "
...           f"{fruits.index('frog')}")
... else:
...     print("'frog' is not among the elements in the list!")
```

This way you can avoid `ValueError`.

10.8 Counting elements within a sequence

Sometimes we have a sequence—string, list, or tuple—and we'd like to know how many times a particular element occurs in the sequence. Python supplies a `.count()` method for each of these types. This method takes an argument and returns the number of times the value of the argument appears in the underlying sequence.

Examples:

```
s = "How many 'a's are in this string which includes " \
    "the phrase 'I'm bananas about bananas'?"
n = s.count('a')
print(n)   # prints 11

# Now let's check for the letter 'z'
n = s.count('z')
print(n)   # prints 0, because there are no 'z's in s
```

You can count non-overlapping substrings in a string as well.

```
dna_fragment = "GACTGTAGACTTAGGGCTTAGAGTACTAGTAGCTGACTGACACG"
n = dna_fragment.count("GTA")
print(n)   # prints 3 (check for yourself)
```

Another example:

```
"""Source of quote: The Metamorphosis by Franz Kafka,
translated by David Wyllie, via Project Gutenberg:
https://www.gutenberg.org/cache/epub/5200/pg5200.txt"""

metamorphosis = ("One morning, when Gregor Samsa woke from "
                 "troubled dreams, he found himself transformed "
                 "in his bed into a horrible vermin. He lay "
                 "on his armour-like back, and if he lifted "
                 "his head a little he could see his brown "
                 "belly, slightly domed and divided by arches "
                 "into stiff sections. The bedding was "
                 "hardly able to cover it and seemed ready "
                 "to slide off any moment. His many legs, "
                 "pitifully thin compared with the size of "
                 "the rest of him, waved about helplessly as "
                 "he looked.")
n = metamorphosis.count("Gregor")
print(n)     # prints 1
n = metamorphosis.count("bed")
print(n)     # prints 2
n = metamorphosis.count("he")
print(n)     # prints 11
# Doesn't count "He" (case-sensitive) but does count "he"
# in "when", "head", "arches", "the", and "helplessly".
```

It works with lists too.

```
lst = [9, 0, 3, 1, 3, 2, 5, 8, 9, 7, 1, 0, 2, 2, 4, 0, 5, 1]
n = lst.count(1)
print(n)   # prints 3
n = lst.count(9)
print(n)   # prints 2
n = lst.count(8)
print(n)   # prints 1
n = lst.count(6)
print(n)   # prints 0
```

It works with tuples also.

```
t = ('Bob', 'Charlene', 'Bob', 'Alice', 'Alice', 'Dolores')
n = t.count('Bob')
print(n)    # prints 2
n = t.count('Charlene')
print(n)    # prints 1
n = t.count("Egbert")
print(n)    # print 0
```

Keep in mind what counts as an element in a sequence.

```
t = (('a', 97), ('b', 98), ('c', 99), ('d', 100), ('e', 101))

n = t.count(('c', 99))
print(n)    # prints 1
n = t.count('c')
print(n)    # prints 0
n = t.count(99)
print(n)    # prints 0
```

The last two calls to `.count()` return zero because neither `'c'` nor `99` are elements of `t`. The elements of `t` are tuples, *e.g.*, `('c', 99)`.

Comprehension check

1. Given the string: `s = "amanaplanacanalpanama"`, what is the evaluation of `s.count('a')`?
2. Given the same string, what is the evaluation of `count('ma')`?
3. Does this work? `count(s, 'p')`? Why or why not?
4. Given this tuple, `t = ('cheddar', 'gouda', 'munster')`, what is the evaluation of `t[0].count('d')`?

10.9 Sequence unpacking

Python provides us with an elegant syntax for unpacking the individual elements of a sequence as separate variables. We call this *unpacking*.

```
>>> x, y = (3.945, 7.002)
>>> x
3.945
>>> y
7.002
```

Here, each element in the tuple on the right-hand side is assigned to a matching variable on the left-hand side.

```
>>> x = (2,)
>>> x
2
>>> x, y, z = ('a', 'b', 'c')
>>> x
'a'
>>> y
'b'
>>> z
'c'
```

This works with tuples of any size!

```
a, b, c, d, e = ('Hello', 5, [1, 2, 3], 'Chocolate', 2022)
```

However, the number of variables on the left-hand side must match the number of elements in the tuple on the right-hand side. If they don't match, we get an error, either `ValueError: too many values to unpack` or `ValueError: not enough values to unpack`.

Tuple unpacking is particularly useful by:

- allowing for more concise and readable code by assigning multiple values to variables on a single line,
- allowing for multiple values to be returned by a function,
- making it easier to swap variable values (more on this shortly).

Can we unpack lists too?

Yup. We can unpack lists the same way.

```
x, y = [1, 2]
```

But this isn't used as much as tuple unpacking. Can you think why this may be so?

The reason is that lists are dynamic, and we may not know at runtime how many elements we have to unpack. This scenario occurs less often

with tuples, since they are immutable, and once created, we know how many elements they have.

What if we want to unpack but we don't care about certain elements in the sequence?

Let's say we want the convenience of sequence unpacking, but on the right-hand side we have a tuple or a function which returns a tuple, and we don't care about some element in the tuple. In cases like this, we often use the variable name _ to signify "I don't really care about this value".

Examples:

```
>>> _, lon = (44.318393, -72.885126)  # don't care about latitude
```

or

```
>>> lat, _ = (44.318393, -72.885126)  # don't care about longitude
```

This makes it clear visually that we're only concerned with a specific value, and is preferred over names like `temp`, `foo`, `junk` or `whatever`.

Occasionally, you may see code where two elements of an unpacked sequence are ignored. In these cases, it's not unusual to see both _ and __ used as variable names to signify "I don't care."

Examples:

```
>>> _, lon, __ = (44.318393, -72.885126, 1244.498)
```

or

```
>>> lat, _, __ = (44.318393 -72.8851266, 1244.498)
```

or

```
>>> _, __, elevation = (44.318393, -72.8851266, 1244.498)
```

It is possible also to reuse _. For example, this works just fine:

```
>>> _, _, elevation = (44.318393, -72.8851266, 1244.498)
```

In this instance, Python unpacks the first element of the tuple to _, then it unpacks the second element of the tuple to _, and then it unpacks the third element to the variable `elevation`.

If you were to inspect the value of _ after executing the line above, you'd see it holds the value −72.8851266.

Swapping variables with tuple unpacking

In many languages, swapping variables requires a temporary variable. Let's say we wanted to swap the values of variables `a` and `b`. In most languages we'd need to do something like this:

```
int a = 1
int b = 2

// now swap
int temp = a
a = b
b = temp
```

This is unnecessary in Python.

```
a = 1
b = 2

# now swap
b, a = a, b
```

That's a fun trick, eh?

10.10 Strings are sequences

We've already seen another sequence type: strings. Strings are nothing more than immutable sequences of characters (more accurately sequences of Unicode code points). Since a string is a sequence, we can use index notation to read individual characters from a string. For example:

```
>>> word = "omphaloskepsis"  # which means "navel gazing"
>>> word[0]
'o'
>>> word[-1]
's'
>>> word[2]
'p'
```

We can use `in` to check whether a substring is within a string. A substring is one or more contiguous characters within a string.

```
>>> word = "omphaloskepsis"
>>> "k" in word
True
>>> "halo" in word
True
>>> "chicken" in word
False
```

We can use `min()` and `max()` with strings. When we do, Python will compare characters (Unicode code points) within the string. In the case of `min()`, Python will return the character with the lowest-valued code point. In the case of `max()`, Python will return the character with the highest-valued code point.

We can also use `len()` with strings. This returns the length of the string.

```
>>> word = "omphaloskepsis"
>>> max(word)
's'
>>> min(word)
'a'
>>> len(word)
14
```

Recall that Unicode includes thousands of characters, so these work with more than just letters in the English alphabet.

Comprehension check

1. What is returned by `max('headroom')`?
2. What is returned by `min('frequency')`?
3. What is returned by `len('toast')`?

10.11 Some more string methods

We've already seen `.upper()`, `.lower()`, and `.capitalize()` but there are many more string methods. Here are a few that you're likely to find quite useful.

`.split()`

It's often the case that we want to divide a string into smaller parts like words or sentences. Of course, we can split a string into individual symbols using the list constructor:

```
>>> s = "My wombat has indigestion."
>>> list(s)
['M', 'y', ' ', 'w', 'o', 'm', 'b', 'a', 't', ' ', 'h', 'a',
 's', ' ', 'i', 'n', 'd', 'i', 'g', 'e', 's', 't', 'i', 'o',
 'n', '.']
```

But what if we wanted to split this into *words* rather than symbols? `.split()` to the rescue! `.split()` will, by default, split a string wherever it encounters whitespace. Example:

```
>>> s = "My wombat has indigestion."
>>> s.split()
['My', 'wombat', 'has', 'indigestion.']
```

Notice what's going on here. The `.split()` method has returned a list of portions of the original string that had been separated by whitespace. The original string is unchanged. This is useful if you want to work with the words in a sentence or other string.

You may ask, what happened to the whitespace? Python discards whatever you use to split the string—in this case, whitespace.

If we have a multi-line string we can split it into words just the same.

```
>>> s = """Wombats are largely herbivorous.
... They eat grasses, shrubs, and other plants.
... They'll also eat roots and bark.
... They've been known to eat certain fungi.
... If they eat Gladys' cooking they'll get indigestion."""
>>> s.split()
['Wombats', 'are', 'largely', 'herbivorous.', 'They', 'eat',
'grasses,', 'shrubs,', 'and', 'other', 'plants.', "They'll",
'also', 'eat', 'roots', 'and', 'bark.', "They've", 'been',
'known', 'to', 'eat', 'certain', 'fungi.', 'If', 'they',
'eat', "Gladys'", 'cooking', "they'll", 'get',
'indigestion.']
```

We can also split a multi-line string into individual lines, by providing the newline character as an argument to `.split()`.

```
>>> s.split("\n")
['Wombats are largely herbivorous.',
 'They eat grasses, shrubs, and other plants.',
 "They'll also eat roots and bark.",
 "They've been known to eat certain fungi.",
 "If they eat Gladys' cooking they'll get indigestion."]
```

By default, `.split()` splits on any whitespace: `' '`, `'\n'`, `'\t'`, *etc.* If we want to be more specific or split on somthing entirely different, we can provide an argument to `.split()`. In the example above, `s.split("\n")` splits `s` on the newline symbol, and that gets us individual sentences.

You can split on just about anything:

```
>>> s = "tussock grass|kangaroo grass|bonfire moss"
>>> s.split("|")
['tussock grass', 'kangaroo grass', 'bonfire moss']
```

`.join()`

Sometimes we wish to assemble a string from individual parts—in essence the opposite of split. For this, we have `.join()`. Join acts on some string,

called the *glue*, and uses that glue to connect parts provided as a list or tuple.

For example, imagine we had a list of tasty things for a wombat to eat (borrowed from the previous example):

```
>>> tasty_things = ['tussock grass', 'kangaroo grass',
... 'bonfire moss']
>>> " and ".join(tasty_things)
'tussock grass and kangaroo grass and bonfire moss'
```

Here, the glue is the string `" and "` and `.join()` takes the elements of the list `tasty_things` and joins them together with the glue. The glue can be any string at all.

Sometimes we want to concatenate a list of strings without anything between them at all. For this, we use the empty string as the glue!

```
>>> letters = ['w', 'o', 'm', 'b', 'a', 't']
>>> "".join(letters)
'wombat'
```

`.replace()`

If you've ever used search and replace to replace some string in a word processing document, this one will seem familiar. Given some string, we can replace all occurrences of a given substring with another using `.replace()`.

```
>>> s = "My wallaby has indigestion."
>>> s.replace('wallaby', 'wombat')
'My wombat has indigestion.'
```

Notice that `s` remains unchanged because strings are immutable. `.replace()` returns a copy of the string, with the requested replacements. So if you want to use this to update `s` (or whatever your string is named), you'll need to perform an assignment.

```
>>> s = "My wallaby has indigestion."
>>> s = s.replace('wallaby', 'wombat')
>>> s
'My wombat has indigestion.'
```

Comprehension check

1. Given the string `'New South Wales'`, how would you split this to produce the list `['New', 'South', 'Wales']`?

2. Given the string `'New South Wales, Queensland, South Australia, Tasmania, Victoria, Western Australia'`, how would you split this to produce the list `['New South Wales', 'Queensland', 'South Australia', 'Tasmania', 'Victoria', 'Western Australia']`?

3. Given the list `['New', 'South', 'Wales']`, how would you join the elements of the list to produce the string `'New South Wales'`?
4. Given the string `'New South Whales'`, how would you construct the string `'New South Wales'`?
5. Given the string `'No Sloth Whales'`, how would you construct the string `'New South Wales'`?
6. What's the result of the following:

   a. `"A wallaby is like a wee tiny kangaroo.".split()`?
   b. `" ".join("A wallaby is like a wee tiny kangaroo.".split())`?

Check your answers in the Python shell.

10.12 The Python built-in `sorted()`

We've seen that lists are mutable, and that a list can be sorted in-place with the list method `.sort()`. Example:

```
>>> hand = ['8♥', '5♦', '2♣', '9♠', '4♥']
>>> hand.sort()  # notice there's no assignment!
>>> hand
['2♣', '4♥', '5♦', '8♥', '9♠']
```

However, because strings and tuples are immutable, they have no `.sort()` method.

```
>>> ("Dolores", "Bob", "Egbert", "Alice", "Charlene").sort()
Traceback (most recent call last):
  File "<python-input-12>", line 1, in <module>
    ("Dolores", "Bob", "Egbert", "Alice", "Charlene").sort()
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'tuple' object has no attribute 'sort'
>>> 'fhtasunbweycjqzplivokgxmdr'.sort()
Traceback (most recent call last):
  File "<python-input-25>", line 1, in <module>
    'fhtasunbweycjqzplivokgxmdr'.sort()
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'str' object has no attribute 'sort'
```

But sometimes we *want* to sort a string or tuple. Does this mean we're out of luck? No. Python provides a built-in for this: `sorted()`.

`sorted()` takes an iterable as an argument and returns a sorted list of the elements in the iterable. Examples:

```
>>> sorted('fhtasunbweycjqzplivokgxmdr')
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
'y', 'z']
```

```
>>> sorted(("Dolores", "Bob", "Egbert", "Alice", "Charlene"))
['Alice', 'Bob', 'Charlene', 'Dolores', 'Egbert']
```

Now, you may say "That's fine, but what if I want a sorted string or tuple and not a list?" Easy. Just construct the desired object from the sorted list. For a string we can use `join()`:

```
>>> "".join(sorted('fhtasunbweycjqzplivokgxmdr'))
'abcdefghijklmnopqrstuvwxyz'
```

For a tuple, we just use the tuple constructor:

```
>>> tuple(sorted(("Dolores", "Bob", "Egbert",
... "Alice", "Charlene")))
('Alice', 'Bob', 'Charlene', 'Dolores', 'Egbert')
```

Comprehension check

1. Why don't objects of type `str` or `tuple` have a `.sort()` method?
2. What's the result of `tuple(sorted('egbert'))`?
3. What's the result of `"".join(sorted('egbert'))`?
4. Why won't this work: `"".join(['c', 'b', 'a'].sort())`?
5. How would you fix the example in #4 above?

10.13 Sequences: a quick reference guide

Mutability and immutability

Type	Mutable	Indexed read	Indexed write
`list`	yes	yes	yes
`tuple`	no	yes	no
`str`	no	yes	no

Built-ins

Type	`len()`	`sum()`	`min()` and `max()`
`list`	yes	yes (if numeric)	yes, with some restrictions
`tuple`	yes	yes (if numeric)	yes, with some restrictions
`str`	yes	no	yes

Methods

Type	`.sort()`, `.append()`, and `.pop()`	`.index()`
`list`	yes	yes
`tuple`	no	yes
`str`	no	yes

- If an object is *mutable*, then the object can be modified.
- Indexed read: `m[i]` where `m` is a list or tuple, and `i` is a valid index into the list or tuple.
- Indexed write: `m[i]` on left-hand side of assignment.
- Python built-in `len()` works the same for lists and tuples.
- Python built-ins `sum()`, `min()`, and `max()` behave the same for lists and tuples.
- Python built-in `sorted()` returns a sorted list of elements in a given sequence.
- For `sum()` to work `m` must contain only numeric types (`int`, `float`) or Booleans. So, for example, `sum([1, 1.0, True])` yields three. We cannot sum over strings.
- `min()` and `max()` work so long as the elements of the list or tuple are *comparable*—meaning that `>`, `>=`, `<`, `<=`, `==` can be applied to any pair of list elements. We cannot compare numerics and strings, but we can compare numerics with numerics and strings with strings.
- We can test whether a value is in a list or tuple with `in`. For example `'cheese' in m` returns a Boolean.
- `m.sort()`, `m.append()`, and `m.pop()` work for lists only. Tuples are immutable. Note that these change the list *in place*.
- We cannot apply `m.sort()` if the list contains elements which are not comparable. By the same token, `sorted()` will will not work if elements in the sequence are not comparable.
- We must supply an argument to `m.append()` (we have to append *something*).
- `m.pop()` without argument pops the last element from a list.
- `m.pop(i)` where `i` is a valid index into `m` pops the element at index `i` from the list.
- We cannot pop from an empty list (IndexError).
- `m.index(x)` will return the index of the first occurrence of `x` in `m`. Note: This will raise `ValueError` if `x` is not in `m`.
- `.replace()` returns a *copy* of a string, replacing occurrences of one substring with another.
- `.join()` takes some glue—a string—and uses this to connect elements in some sequence.
- `.split()` returns a list of elements resulting from splitting the string. Whitespace is the default delimiter, but any string can be supplied as an alternative.

10.14 Slicing

Python supports a powerful means for extracting data from a sequence (string, list or tuple) called *slicing*.

Basic slicing

We can take a *slice* through some sequence by specifying a range of indices.

```
>>> un_security_council = ['China', 'France', 'Russia', 'UK',
...                        'USA', 'Albania', 'Brazil', 'Gabon',
...                        'Ghana', 'UAE', 'India', 'Ireland',
...                        'Kenya', 'Mexico', 'Norway']
```

Let's say we just wanted the permanent members of the UN Security Council (these are the first five in the list). Instead of providing a single index within brackets, we provide a range of indices, in the form `<sequence>[<start>:<end>]`.

```
>>> un_security_council[0:5]
['China', 'France', 'Russia', 'UK', 'USA']
```

"Hey! Wait a minute!" you say, "We provided a range of *six* indices! Why doesn't this include 'Albania' too?"

Reasonable question. Python treats the ending index as its stopping point, so it slices from index 0 to index 5 *but not including the element at index 5!* This is the Python way, as you'll see with other examples soon. It does take a little getting used to, but when you see this kind of indexing at work elsewhere, you'll understand the rationale.

What if we wanted the non-permanent members whose term ends in 2023? That's Albania, Brazil, Gabon, Ghana, and UAE.

To get that slice we'd use

```
>>> un_security_council[5:10]
['Albania', 'Brazil', 'Gabon', 'Ghana', 'UAE']
```

Again, Python doesn't return the item at index 10; it just goes up to index 10 and stops.

Some shortcuts

Python allows a few shortcuts. For example, we can leave out the starting index, and Python reads from the start of the list (or tuple).

```
>>> un_security_council[:10]
['China', 'France', 'Russia', 'UK', 'USA',
 'Albania', 'Brazil', 'Gabon', 'Ghana', 'UAE']
```

By the same token, if we leave out the ending index, then Python will read to the end of the list (or tuple).

```
>>> un_security_council[10:]
['India', 'Ireland', 'Kenya', 'Mexico', 'Norway']
```

Now you should be able to guess what happens if we leave out both start and end indices.

```
>>> un_security_council[:]
['China', 'France', 'Russia', 'UK', 'USA',
 'Albania', 'Brazil', 'Gabon', 'Ghana', 'UAE',
 'India', 'Ireland', 'Kenya', 'Mexico', 'Norway']
```

We get a copy of the entire list (or tuple)!

Guess what these do:

- `un_security_council[-1:]`
- `un_security_council[:-1]`
- `un_security_council[5:0]`
- `un_security_council[5:-1]`

Specifying the stride

Imagine you're on a stepping-stone path through a garden. You might be able to step one stone at a time. You might be able to step two stones at a time—skipping over every other stone. If you have long legs, or the stones are very close together, you might be able to step three stones at a time! We call this *step size* or *stride*.

In Python, when specifying slices we can specify the stride as a third parameter. This comes in handy if we only want values at odd indices or at even indices.

The syntax is `<sequence>[<start>:<stop>:<stride>]`.

Here are some examples:

```
>>> un_security_council[::2]  # only even indices
['China', 'Russia', 'USA', 'Brazil', 'Ghana',
 'India', 'Kenya', 'Norway']
>>> un_security_council[1::2]  # only odd indices
['France', 'UK', 'Albania', 'Gabon', 'UAE',
 'Ireland', 'Mexico']
```

What happens if the stride is greater than the number of elements in the sequence?

```
>>> un_security_council[::1000]
['China']
```

Can we step backward? Sure!

```
>>> un_security_council[-1::-1]
['Norway', 'Mexico', 'Kenya', 'Ireland', 'India',
 'UAE', 'Ghana', 'Gabon', 'Brazil', 'Albania',
 'USA', 'UK', 'Russia', 'France', 'China']
```

Now you know one way to get the reverse of a sequence. Can you think of some use cases for changing the stride?

10.15 Passing mutables to functions

You'll recall that as an argument is passed to a function it is *assigned* to the corresponding formal parameter. This, combined with mutability, can sometimes cause confusion.[7]

Mutable and immutable arguments to a function

Here's an example where some confusion often arises.

```
>>> def foo(lst):
...     lst.append('Waffles')
...
>>> breakfasts = ['Oatmeal', 'Eggs', 'Pancakes']
>>> foo(breakfasts)
>>> breakfasts
['Oatmeal', 'Eggs', 'Pancakes', 'Waffles']
```

Some interpret this behavior incorrectly, assuming that Python must handle immutable and mutable arguments differently! This is *not* correct. Python passes mutable and immutable arguments the same way. The argument is assigned to the formal parameter.

The result of this example is only a little different than if we'd done this:

```
>>> breakfasts = ['Oatmeal', 'Eggs', 'Pancakes']
>>> lst = breakfasts
>>> lst.append('Waffles')
>>> breakfasts
['Oatmeal', 'Eggs', 'Pancakes', 'Waffles']
```

All we've done is give two different *names* to the same list, by assignment. In the example with the function `foo` (above) the only difference is that the name `lst` exists only within the scope of the function. Otherwise, these two examples behave the same.

Notice that there is no "reference" being passed, just an assignment taking place.

[7] If you search on the internet you may find sources that say that immutable object are *passed by value* and *mutable* objects are *passed by reference* in Python. This is not correct! Python always passes by assignment—no exceptions. This is different from many other languages (for example, C, C++, Java). If you haven't heard these terms before, just ignore them.

Names have scope, values do not

This example draws out another point. To quote Python guru Ned Batchelder:

> Names have scope but no type. Values have type but no scope.

What do we mean by that? Well, in the case where we pass the list `breakfast` to the function `foo`, we create a new name for `breakfast`, `lst`. This name, `lst`, exists only within the scope of the function, but the value persists. Since we've given this list another name, `breakfast`, which exists outside the scope of the function we can still access this list once the function call has returned, even though the name `lst` no longer exists.

Here's another demonstration which may help make this more clear.

```
>>> def foo(lst):
...     lst.append('Waffles')
...     print(lst)
...
>>> foo(['Oatmeal', 'Eggs', 'Pancakes'])
['Oatmeal', 'Eggs', 'Pancakes', 'Waffles']
```

However, now we can no longer use the name `lst` since it exists only within the scope of the function.

```
>>> lst
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lst' is not defined. Did you mean: 'list'?
```

Where did the list go?

When an object no longer has a name that refers to it, Python will destroy the object in a process called *garbage collection*. We won't cover garbage collection in detail. Let it suffice to understand that once an object no longer has a name that refers to it, it will be subject to *garbage collection*, and thus inaccessible.

So in the previous example, where we passed a list literal to the function, the only time the list had a name was during the execution of the function. Again, the formal parameter is `lst`, and the argument (in this last example) is the literal `['Oatmeal', 'Eggs', 'Pancakes']`. The assignment that took place when the function was called was `lst = ['Oatmeal', 'Eggs', 'Pancakes']`. Then we appended `'Waffles'`, printed the list, and returned.

Poof! `lst` is gone.

10.16 The `global` keyword

Python provides a keyword, `global`, which is often misunderstood and misused. Some folks seem to believe that use of this keyword is necessary

if we wish to use the value of a variable defined in the outer scope within a function. Not quite!

Consider the common case of having some constant which might be used in many computations. We'd like to have such a constant defined once, and never changed, but available wherever we might need it.

Unlike many other languages, Python does not have a `const` keyword that enforces that a value remain constant. In Python, that's down to you, the programmer.

Here's an example:

```
import math

g_0 = 9.806  # std. accel. due to gravity

def calc_period(l_):
    return 2 * math.pi * math.sqrt(l_ / g_0)


length = float(input("Enter length of pendulum (m): "))
period = calc_period(length)
print(f"The period of your pendulum is {period:.2f} s.")
```

This works just fine. If the user were to enter `10` for the length of the pendulum, it would print

```
The period of your pendulum is 6.35 s.
```

as you'd expect. Notice we did *not* use the `global` keyword, even though we're using `g_0` within our function. Why does this work? It works because if Python encounters an identifier within the body of a function which is not either the name of a formal parameter or a variable created within the body of the function, it will look in the nearest enclosing scope for a matching identifier. This kind of variable is called a *free variable*. So in this case, within the body of the function `g_0` is a free variable. It is defined in the outer scope, so Python finds it there, and uses it in the calculation.

On occasion, I've seen code that looks like this:

```
import math

g_0 = 9.806  # std. accel. due to gravity

def calc_period(l_):
    global g_0  # Notice this line here
    return 2 * math.pi * math.sqrt(l_ / g_0)


length = float(input("Enter length of pendulum (m): "))
period = calc_period(length)
print(f"The period of your pendulum is {period:.2f} s.")
```

It seems that in cases like this the programmer believes that the `global` keyword is telling Python to use `g_0` from the outer scope. But we saw in the previous example that this is not necessary!

So what does the `global` keyword do?

Here's an excerpt from the official Python documentation:

"The `global` statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without global, although free variables may refer to globals without being declared global."[8]

So the `global` keyword allows the programmer to *assign a new value to a variable defined in the outer scope* within *the body of the function*—and this quite often is *not* what is intended at all.

Consider the following example.

```
import math

g_0 = 9.806  # std. accel. due to gravity

def calc_period(l_):
    global g_0  # Notice this line here
    p = 2 * math.pi * math.sqrt(l_ / g_0)
    g_0 = 400  # This is an error
    return p


length = float(input("Enter length of pendulum (m): "))
period = calc_period(length)
print(f"The period of your pendulum is {period:.2f} s.")

# Now perform exactly the same computation, with the same input.
period = calc_period(length)
print(f"The period of your pendulum is {period:.2f} s.")
```

If the user enters `10` at the prompt, this program will print

```
The period of your pendulum is 6.35 s.
The period of your pendulum is 0.99 s.
```

Why? Because the inappropriate use of `global` allowed the programmer to modify `g_0` within the function! So when the function was called a second time, `g_0` had a different value.

Here, the use of `global` permitted a change to what should have been a constant! Yikes!

Let's see what would happen if we removed the `global` statement.

[8] https://docs.python.org/3/reference/executionmodel.html

```
import math

g_0 = 9.806  # std. accel. due to gravity

def calc_period(l_):
    p = 2 * math.pi * math.sqrt(l_ / g_0)
    g_0 = 400  # This creates a new, local g_0,
    # without changing g_0 in the outer scope!
    return p


length = float(input("Enter length of pendulum (m): "))
period = calc_period(length)
print(f"The period of your pendulum is {period:.2f} s.")

# Now perform exactly the same computation,
# with the same input.
period = calc_period(length)
print(f"The period of your pendulum is {period:.2f} s.")
```

This prints

```
The period of your pendulum is 6.35 s.
The period of your pendulum is 6.35 s.
```

This is because without the `global` statement, the assignment `g_0 = 400` created a new, local `g_0` which exists *only within the body of the function*.

So inappropriate use of `global` statements make it possible to modify variables we do not wish to modify!

When is it appropriate to use `global`?

Sometimes, it may make sense to use a global variable to maintain state. Yes, there are those who would say this is *never* a good idea, but it does come up, and Python does have a `global` statement, so let's look at an example anyway.

As one example, let's say we wanted to keep track of how many times a function is called during the execution of a program (maybe counting attempts at making a network connection, or number of times a magic spell is used in a fantasy game). We could use a global variable for this.

```
times_foo_has_been_called = 0

def foo(x, y, z):
    global times_foo_has_been_called
    # do some foo work here, whatever that might be
    times_foo_has_been_called += 1
    return  # ... return some useful value, perhaps
```

That's one example.

The important thing to remember is that Python will look outside the current code block for a free variable *without the use of* `global`, and that `global` should be used when we *want* to modify a global variable. These are two entirely different cases! Of course, if the variable in the outer scope is of a mutable type, that's a different discussion.

Also, make sure you understand mutability, immutability, and pass by assignment which is the only way Python passes arguments to functions.

10.17 Exceptions

`IndexError`

When dealing with sequences, you may encounter `IndexError`. This exception is raised when an integer is supplied as an index, but there is no element at that index.

```
>>> lst = ['j', 'a', 's', 'p', 'e', 'r']
>>> lst[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Notice that the error message explicitly states "list index out of range". In the example above, we have a list of six elements, so valid indices range up to five. There is no element at index six, so if we attempt to access `lst[6]`, an `IndexError` is raised.

`TypeError`

Again, when dealing with sequences, you may encounter `TypeError` in a new context. This occurs if you try to use something other than an integer (or slice) as an index.

```
>>> lst[1.0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
```

```
>>> lst['cheese']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not str
```

```
>>> lst[None]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not NoneType
```

Notice that the error message explicitly states that "list indices must be integers or slices" and in each case, it identifies the offending type that was actually supplied.

10.18 Exercises

Where appropriate, guess first, then check your guess in the Python shell.

Exercise 01

Given the following list,

```
>>> mammals = ['cheetah', 'aardvark', 'bat', 'dikdik', 'ermine']
```

write the indices that correspond to:

a. aardvark
b. bat
c. cheetah
d. dikdik
e. ermine

Give an example of an index, `n`, that would result in an IndexError if we were to use it in the expression `mammals[n]`.

Exercise 02

Given the following tuple

```
>>> elements = (None, 'hydrogen', 'helium', 'lithium',
...             'beryllium', 'boron', 'carbon',
...             'nitrogen', 'oxygen')
```

write the indices that correspond to

a. beryllium
b. boron
c. carbon
d. helium
e. hydrogen
f. lithium
g. nitrogen
h. oxygen

(Extra: If you've had a course in chemistry, why do you think the first element in the tuple is `None`?)

Exercise 03

Given the polynomial $4x^3 + 2x^2 + 5x - 4$, write the coefficients as a tuple and name the result `coefficients`. Use an assignment.

a. What is the value of `len(coefficients)`?

b. What is the value of `coefficients[2]`?

c. Did you write the coefficients in ascending or descending order? Why did you make that choice?

Exercise 04

Given the lists in exercises 1 and 2

```
>>> mammals = ['cheetah', 'aardvark', 'bat', 'dikdik', 'ermine']
>>> elements = (None, 'hydrogen', 'helium', 'lithium',
...             'beryllium', 'boron', 'carbon',
...             'nitrogen', 'oxygen')
```

what is the evaluation of the following expressions?

a. `len(mammals) > len(elements)`
b. `elements[5] < mammals[2]`
c. `elements[-1]`

Exercise 05

Write a function which, given any arbitrary list, returns `True` if the list has an even number of elements and `False` if the list has an odd number of elements. Name your function `is_even_length()`.

Exercise 06

Given the following list

```
moons = ['Mimas', 'Enceladus', 'Tethys', 'Dione']
```

write one line of code which calls the function `is_even_length()` (see Exercise 05) with `moons` as an argument and assigns the result to an object named `n`. What is `n`'s type?

Exercise 07

Given the following list, which contains data about some moons of Saturn and their diameters (in km),

```
moons = [('Mimas', 396.4), ('Enceladus', 504.2),
         ('Tethys', 1062.2), ('Dione', 1122.8)]
```

a. If we were to perform the assignment `m = moons[0]`, what would `m`'s type be?

b. How would we get the name of the moon from `m`?

c. How would we get the diameter of the moon from `m`?

d. Write a single line of code which calculates the average diameter of these moons, and assigns the result to an object named `avg_diameter`. (No, you do not need a loop.)

e. Write one line of code which adds Iapetus to the list, using the same structure. The diameter of Iapetus is 1468.6 (km).

f. In one line of code, write an expression which returns the diameter of Enceladus, and assigns the result to an object named `diameter`.

g. What is the evaluation of the expression `moons[0][0] < moons[1][0]`?

Exercise 08

Given the following

```
>>> countries = ['Ethiopia', 'Benin', 'Ghana', 'Angola',
...              'Cameroon', 'Kenya']
>>> countries.append('Nigeria')
>>> countries.sort()
>>> countries.pop()
>>> country = countries[2]
```

answer the following questions:

a. What is the value of `len(countries)`?

b. What is the resulting value of `country`?

c. What is the evaluation of `len(countries[3])`

Exercise 09

There are three ways to make a copy of a list:

```
lst_2 = lst_1.copy()   # using the copy() method

lst_2 = lst_1[:]       # slice encompasses entire list

lst_2 = list(lst_1)    # using the list() constructor
```

Write a function that takes a list as an argument, makes a copy of the list, modifies the list, and *returns* the modified copy. How you modify the list is up to you, but you should use at least two different list methods. Demonstrate that when you call this function with a list variable as an argument, that the function returns a modified copy, and that the original list is unchanged.

Chapter 11

Loops, iteration, and iterables

In this chapter, we introduce *loops*, *iteration*, and *iterables*. With loops, we can automate repetitive tasks or calculations. Why are they called loops? Well, so far we've seen code which (apart from function calls) progresses in a linear fashion from beginning to end (even if there are branches, we still proceed in a linear fashion). Loops change the shape of the execution of our code in a very interesting and powerful way. They loop!

Looping allows portions of our code to execute repeatedly—either for a fixed number of times or while some condition holds—before moving on to execute the rest of our code.

Python provides us with two types of loop: `for` loops and `while` loops. `for` loops work by iterating over some iterable object, be it a list, a tuple, a range, or even a string. `while` loops continue as long as some condition is true.

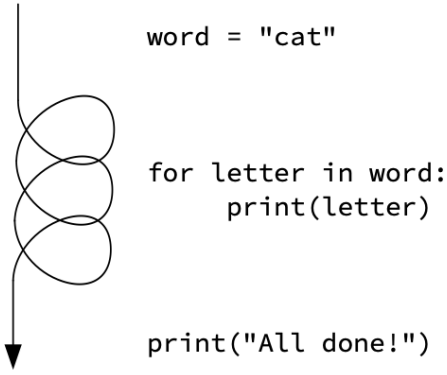


Figure 11.1

With variables, functions, branching, and loops, we have all the tools we need to create powerful programs. All the rest, as they say, is gravy.

Learning objectives

- You will learn how to use `for` loops and `while` loops.
- You will learn how to iterate over sequences.
- You will learn how to define and use conditions which govern a `while` loop.
- You will learn how to choose which type of loop is best for a particular problem.

Terms, Python keywords, built-in functions, and types introduced

- accumulator
- alternating sum
- arithmetic sequence
- `break`
- `enumerate()` (built-in) and `enumerate` (type)
- `.extend()` list method
- Fibonacci sequence
- `for`
- iterable
- iterate
- loop
- nested loop
- `range()` (built-in) and `range` (type)
- stride
- summation
- `while`

11.1 Loops: an introduction

It is very often the case that we want to perform a repetitive task, or perform some calculation that involves a repetition of a step or steps. This is where computers really shine. Humans don't much enjoy repetitive tasks. Computers couldn't care less. They're capable of performing repetitive tasks with relative ease.

We perform repetitive tasks or calculations, or operations on elements in some data structure using *loops* or *iteration*.

We have two basic types of loops in Python: `while` loops, and `for` loops.

If you've written code in another language, you may find that Python handles `while` loops in a similar fashion, but Python handles `for` loops rather differently.

A `while` loop performs some repetitive task or calculation *as long as some condition is true*.

A `for` loop *iterates* over an *iterable*. What is an *iterable*? An iterable is a composite object (made of parts, called "members" or "elements") which is capable of returning its members one at a time, in a specific sequence. Recall: lists, tuples, and strings are all iterable.

Take, for example, this list:

```
m = [4, 2, 0, 1, 7, 9, 8, 3]
```

If we ask Python to iterate over this list, the list object itself "knows" how to return a single member at a time: 4, then 2, then 0, then 1, *etc.* We can use this to govern how many iterations we wish to perform and also to provide data that we can use in our tasks or calculations.

In the following sections, we'll give a thorough treatment of both kinds of loop: `while` and `for`.

11.2 `while` loops

Sometimes, we wish to perform some task or calculation repetitively, but we only want to do this under certain conditions. For this we have the `while` loop. A `while` loop continues to execute, *as long as some condition is true or has a truthy value.*

Imagine you're simulating a game of blackjack, and it's the dealer's turn. The dealer turns over their down card, and if their total is less than 17 they must continue to draw until they reach 17 or they go over 21.[1] We don't know how many cards they'll draw, but they must continue to draw until the condition is met. A `while` loop will repeat as long as some condition is true, so it's perfect for a case like this.

Here's a little snippet of Python code, showing how we might use a `while` loop.

```
# Let's say the dealer has a five showing, and
# then turns over a four. That gives the dealer
# nine points. They *must* draw.
dealer = 9

prompt = "What's the value of the next card drawn? "

while dealer < 17:
    next_card = int(input(prompt))
    dealer = dealer + next_card

if dealer <= 21:
    print(f"Dealer has {dealer} points!")
else:
    print(f"Oh, dear. "
          f"Dealer has gone over with {dealer} points!")
```

Here the dealer starts with a total of nine. Then, in the `while` loop, we keep prompting for the number of points to be added to the dealer's hand. Points are added to the value of `dealer`. This loop will continue to execute as long as the dealer's score is less than 17. We see this in the `while` condition:

[1] If you're unfamiliar with the rules of blackjack, see https://en.wikipedia.org/wiki/Blackjack

```
while dealer < 17:
    ...
```

Naturally, this construction raises some questions.

- What can be used as a `while` condition?
- When is the `while` condition checked?
- When does the `while` loop terminate?
- What happens after the loop terminates?

A `while` condition can be any expression which evaluates to a Boolean or truthy value

In the example above, we have a simple Boolean expression as our `while` condition. However, a `while` condition can be any Boolean expression, simple or compound, or any value or expression that's truthy or falsey—and in Python, that's just about anything! Examples:

```
lst = ['a', 'b', 'c']

while lst:
    print(lst.pop(0))
```

Non-empty sequences are truthy. Empty sequences are falsey. So as long as the list `lst` contains any elements, this `while` loop will continue. This will print

```
a
b
c
```

and then terminate (because after popping the last element, the list is empty, and thus falsey).

```
while x < 100 and x % 2 == 0:
    ...
```

This loop will execute as long as `x` is less than 100 and `x` is even.

The condition of a `while` loop is checked *before* each iteration

The condition of a `while` loop is checked *before* each iteration of the loop. In this case, the condition is `dealer < 17`. At the start, the dealer has nine points, so `dealer < 17` evaluates to `True`. Since this condition is true, the *body* of the loop is executed. (The body consists of the indented lines under `while dealer < 17`.)

Once the body of the `while` loop has executed, the condition is checked *again*. If the condition remains true, then the body of the loop will be executed *again*. That's why we call it a loop!

It's important to understand that the condition is not checked while the body of the loop is executing.

The condition is always checked *before* executing the body of the loop. This might sound paradoxical. Didn't we just say that *after* executing the body the condition is checked again? Yes. That's true, and it's in the nature of a loop to be a little… circular. However, what we're checking in the case of a `while` loop is whether or not we should execute the body. If the condition is true, then we execute the body, then we loop back to the beginning and check the condition again.

Termination of a `while` loop

At some point (if we've designed our program correctly), the `while` condition becomes false. For example, if the dealer were to draw an eight, then adding eight points would bring the dealer's score to 17. At that point, the condition `dealer < 17` would evaluate to `False` (because 17 is not less than 17), and the loop terminates.

After the loop

Once a while terminates, code execution continues with the code which follows the loop.

It's important to understand that the `while` *condition is not evaluated again after the loop has terminated.*

Review of our blackjack loop

When we first reach this line of code, the value of dealer is 9, so the condition is true. We enter the loop and the body of the loop is executed.

```
dealer = 9

prompt = "What's the value of the next card drawn? "

while dealer < 17:
    next_card = int(input(prompt))
    dealer = dealer + next_card


if dealer <= 21:
    print(f"Dealer has {dealer} points!")
else:
    print(f"Oh, dear. "
          f"Dealer has gone over with {dealer} points!")
```

Figure 11.2

Having entered the loop, the body is executed. The condition is not evaluated while the body is being executed.

```
dealer = 9

prompt = "What's the value of the next card drawn? "

while dealer < 17:
    next_card = int(input(prompt))
    dealer = dealer + next_card


if dealer <= 21:
    print(f"Dealer has {dealer} points!")
else:
    print(f"Oh, dear. "
          f"Dealer has gone over with {dealer} points!")
```

Figure 11.3

Loop!

Go back to the start of the loop, and check the condition again.

If the condition is still true, we execute the body again.

```
dealer = 9

prompt = "What's the value of the next card drawn? "

while dealer < 17:
    next_card = int(input(prompt))
    dealer = dealer + next_card


if dealer <= 21:
    print(f"Dealer has {dealer} points!")
else:
    print(f"Oh, dear. "
          f"Dealer has gone over with {dealer} points!")
```

Figure 11.4

Since we've looped, we execute the body again.

```
dealer = 9

prompt = "What's the value of the next card drawn? "

while dealer < 17:
    next_card = int(input(prompt))
    dealer = dealer + next_card


if dealer <= 21:
    print(f"Dealer has {dealer} points!")
else:
    print(f"Oh, dear. "
          f"Dealer has gone over with {dealer} points!")
```

Figure 11.5

Loop!

At some point, the condition should become false, at which point the loop terminates…

```
dealer = 9

prompt = "What's the value of the next card drawn? "

while dealer < 17:
    next_card = int(input(prompt))
    dealer = dealer + next_card


if dealer <= 21:
    print(f"Dealer has {dealer} points!")
else:
    print(f"Oh, dear. "
          f"Dealer has gone over with {dealer} points!")
```

Figure 11.6

…at which point we exit the loop and continue with the program code after the loop.

Note that we don't re-evaluate the condition after exiting the loop.

```
dealer = 9

prompt = "What's the value of the next card drawn? "

while dealer < 17:
    next_card = int(input(prompt))
    dealer = dealer + next_card


if dealer <= 21:
    print(f"Dealer has {dealer} points!")
else:
    print(f"Oh, dear. "
          f"Dealer has gone over with {dealer} points!")
```

Figure 11.7

Another example: coffee shop queue with limited coffee

Here's another example of using a `while` loop.

Let's say we have a queue of customers at a coffee shop. They all want coffee (of course). The coffee shop offers small (8 oz), medium (12 oz) and large (20 oz) coffees. However, the coffee shop has run out of beans and all they have is what's left in the urn. The baristas have to serve the customers in order, and can only take orders as long as there's at least 20 oz in the urn.

We can write a function which calculates how many people are served in the queue and reports the result. To do this we'll use a `while` loop. Our function will take three arguments: the number of ounces of coffee in the urn, a list representing the queue of orders, and the minimum amount of coffee that must remain in the urn before the baristas must stop taking orders. The queue will be a list of values—8, 12, or 20—depending on which size each customer requests. For example,

```
queue = [8, 12, 20, 20, 12, 12, 20, 8, 12, ...]
```

Let's call the amount of coffee in the urn `reserve`, the minimum `minimum`, and our queue of customers `customers`. Our `while` condition is `reserve >= minimum`.

```
def serve_coffee(reserve, customers, minimum):

    customers_served = 0

    while reserve >= minimum:
        reserve = reserve - customers[customers_served]
        customers_served += 1

    print(f"We have served {customers_served} customers, "
          f"and we have only {reserve} ounces remaining.")
```

At each iteration, we check to see if we still have enough coffee to continue taking orders. Then, within the body of the loop, we take the customers in order, and—one customer at a time—we deduct the amount of coffee they've ordered. Once the reserve drops below the minimum, we stop taking orders and report the results.

What happens if the `while` condition is never met?

Let's say we called the `serve_coffee()` function (above), with the arguments, `6`, `lst`, and `8`, where `lst` is some arbitrary list of orders:

```
serve_coffee(6, lst, 8)
```

In this case, when we check the `while` condition the first time, the condition fails, because six is not greater than or equal to eight. Thus, the body of the loop would never execute, and the function would report:

```
We have served 0 customers, and we have only 6 ounces remaining.
```

So it's possible that the body of any given `while` loop might never be executed. If, at the start, the `while` condition is false, Python will skip past the loop entirely!

11.3 Input validation with `while` loops

A common use for a `while` loop is *input validation*.

Let's say we want the user to provide a number from 1 to 10, inclusive. We present the user with a prompt:

```
Pick a number from 1 to 10:
```

So far, we've only seen how to complain to the user:

```
Pick a number from 1 to 10: 999
You have done a very bad thing.
I will terminate now and speak to you no further!
```

That's not very user-friendly! Usually what we do in cases like this is we continue to prompt the user until they supply a suitable value.

```
Pick a number from 1 to 10: 999
Invalid input.
Pick a number from 1 to 10: -1
Invalid input.
Pick a number from 1 to 10: 7
You have entered 7, which is a very lucky number!
```

But here's the problem: We don't know how many tries it will take for the user to provide a valid input! Will they do so on the first try? On the second try? On the fourth try? On the twelfth try? We just don't know! Thus, a `while` loop is the perfect tool.

How would we implement such a loop in Python? What would serve as a condition?

Plot twist: In this case, we'd choose a condition that's *always* true, and then only *break* out of the loop when we have a number in the desired range. This is a common idiom in Python (and many other programming languages).

```
while True:
    n = int(input("Pick a number from 1 to 10: "))
    if 1 <= n <= 10:
        break
    print("Invalid input.")

if n == 7:
    print("You have entered 7, "
          "which is a very lucky number!")
else:
    print(f"You have entered {n}. Good for you!")
```

Notice what we've done here: the `while` condition is the Boolean literal `True`. This can *never* be false! So we have to have a way of exiting the loop. That's where `break` comes in. `break` is a Python keyword which means "break out of the nearest enclosing loop." The `if` clause includes a condition which is only true if the user's choice is in the desired range. Therefore, this loop will execute indefinitely, until the user enters a number between one and 10.

As far as user experience goes, this is much more friendly than just terminating the program immediately if the user doesn't follow instructions. Rather than complaining and exiting, our program can ask again when it receives invalid input.

A note of caution

While the example above demonstrates a valid use of `break`, `break` should be used sparingly. If there's a good way to write a `while` loop without using `break` then you should do so! This often involves careful consideration of `while` conditions—a worthwhile investment of your time.

It's also considered bad form to include more than one `break` statement within a loop. Again, please use `break` sparingly.

Other applications of `while` loops

We'll see many other uses for the `while` loop, including performing numeric calculations and reading data from a file.

Comprehension check

1. What is printed?

```
>>> c = 5
>>> while c >= 0:
...     print(c)
...     c -= 1
```

2. How many times is "Hello" printed?

```
>>> while False:
...    print("Hello")
...
```

3. What's the problem with this `while` loop?

```
>>> while True:
...     print("The age of the universe is...")
...
```

4. How many times will this loop execute?

```
>>> while True:
...     break
...
```

5. How many times will this loop execute?

```
>>> n = 10
>>> while n > 0:
...     n = n // 2
...
```

6. Here's an example showing how to pop elements from a list within a loop.

```
>>> while some_list:
...     element = some_list.pop()
...     # Now do something useful with that element
...
```

Ask yourself:

- Why does this work?
- When does the while loop terminate?
- What does this have to do with truthiness or falsiness?
- Is an empty list falsey?

Challenge!

How about this loop? Try this out with a hand-held calculator.

```
EPSILON = 0.01

x = 2.0

guess = x
while True:
    guess = sum((guess, x / guess)) / 2

    if abs(guess ** 2 - x) < EPSILON:
        break

print(guess)
```

(`abs()` is a built-in Python function which calculates the absolute value of a number.) What does this loop do?

11.4 An ancient algorithm with a `while` loop

There's a rather beautiful algorithm for finding the greatest common divisor of two positive integers.

You may recall from your elementary algebra course that the *greatest common divisor* of two positive integers, a and b, is the greatest integer which divides both a and b without a remainder.

For example, the greatest common divisor of 120 and 105 is 15. It's clear that 15 is a divisor of both 120 and 105:

$$120/15 = 8$$
$$105/15 = 7.$$

How do we know that 15 is the *greatest* common divisor? One way is to factor both numbers and find all the common factors.

$$120 = 2 \times 2 \times 2 \times 3 \times 5$$

$$105 = 3 \times 5 \times 7$$

We've found the common factors of 120 and 105, which are 3 and 5, and their product is 15. Therefore, 15 is the greatest common divisor of 120 and 105. This works, and it may well be what you learned in elementary algebra, however, it becomes difficult with larger numbers and isn't particularly efficient.

Euclid's algorithm

Euclid was an ancient Greek mathematician who flourished around 300 BCE. Here's an algorithm that bears Euclid's name. It was presented in Euclid's *Elements*, but it's likely that it originated many years before Euclid.[2]

Euclid's GCD algorithm
input : Positive integers, a and b
output: Calculates the GCD of a and b
while *b does not equal 0* **do**
  Find the remainder when we divide a by b;
  Let a equal b;
  Let b equal the remainder;
end
a is the GCD

Let's work out an example. Say we have $a = 342$ and $b = 186$.

First, we find the remainder of 342/186. 186 goes into 342 once, leaving a remainder of 156. Now, let $a = 186$, and let $b = 156$. Does b equal 0? No, so we continue.

Find the remainder of 186/156. 156 goes into 186 once, leaving a remainder of 30. Now, let $a = 156$, and let $b = 30$. Does b equal 0? No, so we continue.

Find the remainder of 156/30. 30 goes into 156 five times, leaving a remainder of 6. Now, let $a = 30$, and let $b = 6$. Does b equal 0? No, so we continue.

Find the remainder of 30/6. 6 goes into 30 five times, leaving a remainder of 0. Now, let $a = 6$, and let $b = 0$. Does b equal 0? Yes, so we are done.

The GCD is the value of a, so the GCD is 6.

Pretty cool, huh?

Why does it work?

If we have a and b both positive integers, with $a > b$, then we can write

$$a = bq + r$$

where q is the quotient of dividing a by b (Euclidean division) and r is the remainder. For example, in the first step of our example (above) we have

$$342 = 1 \times 186 + 156.$$

It follows that the GCD of a and b equals the GCD of b and r.[3] That is,

[2] Some historians believe that Eudoxus of Cnidus was aware of this algorithm (*c.* 375 BCE), and it's quite possible it was known before that time.

[3] If we have positive integers a, b with $a > b$, then by the division algorithm, we know there exist integers q, r, such that $a = bq + r$, with $b > r \geq 0$. Let d be a common divisor of a and b. Since d divides a and d divides b, then there exist integers n, m, such that $a = dm$ and $b = dn$. By substitution, we have $dm = dqn + r$. Rearranging terms we have $dm - dqn = r$. By factoring, we have $d(m - qn) = r$.

$$\gcd(a, b) = \gcd(b, r).$$

Thus, by successive divisions, we continue to reduce the problem to smaller and smaller terms. At some point in the execution of the algorithm, b becomes 0, and we can divide no further. At this point, what remains as the value for a is the GCD, because the greatest common divisor of a and zero is a!

This algorithm saves us from having to factor both terms.

Consider a larger problem instance with $a = 30759$ and $b = 9126$. Factoring these would be a nuisance, but the Euclidean algorithm takes only eight iterations to find the answer, 3.

Show me the code!

Here's an implementation in Python.

```
"""
Implementation of Euclid's algorithm.
"""

a = int(input("Enter a positive integer, a: "))
b = int(input("Enter a positive integer, b: "))

while b != 0:
    remainder = a % b
    a = b
    b = remainder

print(f"The GCD is {a}.")
```

That's one elegant little algorithm (and one of my personal favorites). It also demonstrates the use of a `while` loop, which is needed, since we don't know *a priori* how many iterations it will take to reach a solution.

11.5 `for` loops

We've seen that `while` loops are useful when we know we wish to perform a calculation or task, but we don't know in advance how many iterations we may need. Thus, `while` loops provide a condition, and we loop until that condition (whatever it may be) no longer holds true.

Python has another type of loop which is useful when:

- we know exactly how many iterations we require, or
- we have some sequence (for example, list, tuple, or string) and we wish to perform calculations, tasks, or operations with respect to the elements of the sequence (or some subset thereof).

Therefore, d divides r. Thus the set of common divisors of a and b is the same as the set of common divisors of b and r. Thus the greatest element of each of these sets must be the same. Therefore, we have $\gcd(a, b) = \gcd(b, r)$, as desired.

This new kind of loop is the `for` loop. `for` loops are so named because they iterate *for* each element in some iterable. Python `for` loops *iterate* over some *iterable*. Always.

What's an iterable? Something we can iterate over, of course! And what might that be? The sequence types we've seen so far (list, tuple, string) are sequences, and these are iterable. We can also produce other iterable objects (which we shall see soon).

Here's an example. We can iterate over a list, `[1, 2, 3]`, by taking the elements, one at a time, in the order they appear in sequence.

```
>>> numbers = [1, 2, 3]
>>> for n in numbers:
...     print(n)
...
1
2
3
```

See? In our `for` loop, Python iterated over the elements (a.k.a. "members") of the list provided. It started with 1, then 2, then 3. At that point the list was exhausted, so the loop terminated.

If it helps, you can read `for n in numbers:` as "for each number, n, in the iterable called 'numbers'."

This works for tuples as well.

```
>>> letters = ('a', 'b', 'c')
>>> for letter in letters:
...     print(letter)
...
a
b
c
```

Notice the syntax: `for <some variable> in <some iterable>:`. As we iterate over some iterable, we get each member of the iterable in turn, one at a time. Accordingly, we need to assign these members (one at a time) to some variable.

In the first example, above the variable has the identifier `n`.

```
>>> numbers = [1, 2, 3]
>>> for n in numbers:
...     print(n)
...
```

As we iterate over `numbers` (a list), we get one element from the list at a time (in the order they appear in the list). So at the first iteration, `n` is assigned the value `1`. At the second iteration, `n` is assigned the value `2`. At the third iteration, `n` is assigned the value `3`. After the third iteration, there are no more elements left in the sequence and the loop terminates.

Thus, the syntax of a `for` loop *requires* us to give a variable name for the variable which will hold the individual elements of the sequence. For example, we cannot do this:

```
>>> for [1, 2, 3]:
...     print("Hello!")
```

If we were to try this, we'd get a `SyntaxError`. The syntax that must be used is:

```
for <some variable> in <some iterable>:
    # body of the loop, indented
```

where `<some variable>` is replaced with a valid variable name, and `<some iterable>` is the name of some iterable, be it a list, tuple, string, or other iterable.

Iterating over a range of numbers

Sometimes we want to iterate over a *range* of numbers or we wish to iterate some fixed number of times, and Python provides us with a means to do this: the `range` type. This is a new type that we've not seen before. `range` objects are iterable, and we can use them in `for` loops.

We can create a new `range` object using Python's built-in function `range()`. This function, also called the `range` constructor, is used to create `range` objects representing *arithmetic sequences*.[4]

Before we create a loop using a `range` object, let's experiment a little. The simplest syntax for creating a `range` object is to pass a positive integer as an argument to the `range` constructor. What we get back is a `range` object, which is like a list of numbers. If we provide a positive integer, `n`, as a single argument, we get a `range` object with `n` elements.

```
>>> r = range(4)
```

Now we have a `range` object, named `r`. Let's get nosy.

```
>>> len(r)
4
```

OK. So `r` has four elements. That checks out.

[4] An arithmetic sequence, is a sequence of numbers such that the difference between any number in the sequence and its predecessor is constant. 1, 2, 3, 4, …is an arithmetic sequence because the difference between each of the terms is 1. Similarly, 2, 4, 6, 8, …is an arithmetic sequence because the difference between each term is 2. Python range objects are restricted to arithmetic sequences of integers.

```
>>> r[0]
0
>>> r[1]
1
>>> r[2]
2
>>> r[3]
3
>>> r[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: range object index out of range
```

We see that the values held by this `range` object, are 0, 1, 2, and 3, in that order.

Now let's use a `range` object in a `for` loop. Here's the simplest possible example:

```
>>> for n in range(4):
...     print(n)
```

What do you think this will print?

- The numbers 1 through 4?
- The numbers 0 through 4? (since Python is zero-indexed)
- The numbers 0 through 3? (since Python slices go up to, but do not include, the stop index)

Here's the answer:

```
>>> for n in range(4):
...     print(n)
...
0
1
2
3
```

Zero through three. `range(n)` with a single integer argument will generate an arithmetic sequence from zero up to, *but not including*, the value of the argument.

Notice, though, that if we use `range(n)` our loop will execute `n` times.

What if we wanted to iterate integers in the interval $[5, 10]$? How would we do that?

```
>>> for n in range(5, 11):
...     print(n)
...
5
6
7
8
9
10
```

The syntax here, when we use two arguments, is `range(<start>, <stop>)`, where `<start>` and `<stop>` are integers or variables with integer values. The `range` will include integers starting at the start value up to *but not including* the stop value.

What if, for some reason, we wanted only even or odd values? Or what if we wanted to count by threes, or fives, or tens? Can we use a different step size or stride? Yes, of course. These are all valid arithmetic sequences. Let's count by threes.

```
>>> for n in range(3, 19, 3):
...     print(n)
...
3
6
9
12
15
18
```

This three argument syntax is `range(<start>, <stop>, <stride>)`. The last argument, called the *stride* or *step size* corresponds to the difference between terms in the arithmetic sequence (the default stride is one).

Can we go backward? Yup. We just use a negative stride, and adjust the start and stop values accordingly.

```
>>> for n in range(18, 2, -3):
...     print(n)
...
18
15
12
9
6
3
```

This yields a `range` which goes from 18, down to *but not including* two, counting backward by threes.

So you see, `range()` is pretty flexible.

What if I just want to do something many times and I don't care about the members in the sequence?

No big deal. While we do require a variable to hold each member of the sequence or other iterable we're iterating over, we aren't *required* to use it in the body of the loop. There is a convention, not required by the language, but commonly used, to use an underscore as the name for a variable that we aren't going to use or don't really care about.

```
>>> for _ in range(5):
...     print("I don't like Brussles sprouts!")
...
I don't like Brussles sprouts!
I don't like Brussles sprouts!
I don't like Brussles sprouts!
I don't like Brussles sprouts!
I don't like Brussles sprouts!
```

(Now you know how I feel about Brussels sprouts.)

Comprehension check

1. What is the evaluation of `sum(range(5))`?
2. What is the evaluation of `max(range(10))`?
3. What is the evaluation of `len(range(0, 10, 2))`

11.6 Iterables

As we have seen, iterables are Python's way of controlling a `for` loop.

You can think of an iterable as a sequence or composite object (composed of many parts) which can return one element at a time, until the sequence is exhausted. We usually refer to the elements of an iterable as *members* of the iterable.

It's much like dealing playing cards from a deck, and doing something (performing a task or calculation) once for each card that's dealt.

A deck of playing cards is an iterable. It has 52 members (the individual cards). The cards have some order (they may be shuffled or not, but the cards in a deck are ordered nonetheless). We can deal cards *one at a time*. This is *iterating* through the deck. Once we've dealt the 52nd card, the deck is exhausted, and iteration stops.

Now, there are two ways we could use the cards.

First, we can *use the information that's encoded in each card*. For example, we could say the name of the card, or we could add up the pips on each card, and so on.

Alternatively, if we wanted to do something 52 times (like push-ups) we could do one push-up for every card that was dealt. In this case, the information encoded in each card and the order of the individual cards would be irrelevant. Nevertheless, if we did one push-up for every card that was dealt, we'd know when we reached the end of the deck that we'd done 52 push-ups.

So it is in Python. When iterating some iterable, we can use the data or value of each member (say calculating the sum of numbers in a list), or we can just use iteration as a way of keeping count. Both are OK in Python.

Using the data provided by an iterable

Here are two examples of using the data of members of an iterable.

First, assume we have a list of integers and we want to know how many of those numbers are even and how many are odd. Say we have such a list in a variable named `lst`.

```
evens = 0

for n in lst:
    if n % 2 == 0:  # it's even
        evens += 1

print(f"There are {evens} even numbers in this list, "
      f"and there are {len(lst) - evens} odd numbers.")
```

As another example, say we have a list of all known periodic comets, and we want to produce a list of those comets with an orbital period of less than 100 years. We would iterate through the list of comets, check to see each comet's orbital period, and if that value were less than 100 years, we'd append that comet to another list. In the following example, the list `COMETS` contains tuples in which the first element of the tuple is the name of the comet, and the second element is its orbital period in years.[5]

```
"""
Produce a list of Halley's type periodic comets
with orbital period less than 100 years.
"""
COMETS = [('Mellish', 145), ('Sheppard-Trujillo', 66),
          ('Levy', 51), ('Halley', 75), ('Borisov', 152),
          ('Tsuchinshan', 101), ('Holvorcem', 38)]
          # This list is abridged. You get the idea.

short_period_comets = []

for comet in COMETS:
    if comet[1] < 100:
        short_period_comets.append(comet)
        # Yes, there's a better way to do this,
        # but this suffices for illustration.
```

Here we're using the data encoded in each member of the iterable, `COMETS`.

[5]The *orbital period* of a comet is the time it takes for the comet to make one orbit around the sun.

Using an iterable solely to keep count

```
for _ in range(1_000_000):
    print("I will not waste chalk")
```

Here we're not using the data encoded in the members of the iterable. Instead, we're just using it to keep count. Accordingly, we've given the variable which holds the individual members returned by the iterable the name `_`. `_` is commonly used as a name for a variable that we aren't going to use in any calculation. It's the programmer's way of saying, "Yeah, whatever, doesn't matter what value it has and I don't care."

So these are two different ways to treat an iterable. In one case, we care about the value of each member of the iterable; in the other, we don't. However, *both* approaches are used to govern a `for` loop.

11.7 Iterating over strings

We've seen how we can iterate over sequences such as lists, tuples, and ranges. Python allows us to iterate over strings the same way we do for other sequences!

When we iterate over a string, the iterator returns one character at a time. Here's an example:

```
>>> word = "cat"
>>> for letter in word:
...     print(letter)
...
c
a
t
```

11.8 The list method `.extend()`

Now that we've seen sequences and iterables, we can introduce a useful list method, `.extend()`. It's often the case that we wish to append more than one element to an existing list.

First, let's review the limitation of append, and look at some ways to make a bigger list from smaller elements.

Recall that the append method requires an argument—the thing that gets appended to the list.

```
>>> shopping_list = ['lettuce', 'hummus', 'lentils']
>>> shopping_list.append('lemon')
>>> shopping_list
['lettuce', 'hummus', 'lentils', 'lemon']
```

But what if we want to add several elements? We could use `.append()` in a loop, but we can't do this:

```
>>> shopping_list = ['lettuce', 'hummus', 'lentils']
>>> shopping_list.append(['lemon', 'tahini', 'sumac'])
```

I mean, this is OK. It appends to `shopping_list`, but what does it append? Not the elements of the list `['lemon', 'tahini', 'sumac']`, but the list itself! So the result is:

```
>>> shopping_list
['lettuce', 'hummus', 'lentils', ['lemon', 'tahini', 'sumac']]
```

That's a list within a list. Not at all what we wanted.

We could concatenate the two lists, but that requires an assignment.

```
>>> shopping_list = ['lettuce', 'hummus', 'lentils']
shopping_list = shopping_list + ['lemon', 'tahini', 'sumac']
>>> shopping_list
['lettuce', 'hummus', 'lentils', 'lemon', 'tahini', 'sumac']
```

That's slightly inelegant.

We could use a slice:

```
>>> shopping_list = ['lettuce', 'hummus', 'lentils']
>>> shopping_list[len(shopping_list):] = ['lemon', 'tahini', 'sumac']
>>> shopping_list
['lettuce', 'hummus', 'lentils', 'lemon', 'tahini', 'sumac']
```

That's opaque at best—indeed, it's almost obfuscatory!

Here's how we'd do the same with `.extend()`.

```
>>> shopping_list = ['lettuce', 'hummus', 'lentils']
>>> shopping_list.extend(['lemon', 'tahini', 'sumac'])
>>> shopping_list
['lettuce', 'hummus', 'lentils', 'lemon', 'tahini', 'sumac']
```

Much easier to read; much prettier.

`.extend()` takes an *iterable* as an argument and appends each of the elements of the iterable to the underlying list. (Do you think there might be a loop lurking in the background?)

However, we can't dispense with `.append()`, because `.extend()` only takes an iterable as an argument. For example, this fails:

```
>>> numbers = [1, 2, 3]
>>> numbers.extend(4)
Traceback (most recent call last):
  File "<python-input-21>", line 1, in <module>
    numbers.extend(4)
    ~~~~~~~~~~~~~~~^^^
TypeError: 'int' object is not iterable
```

This fails because integers aren't iterable.

Comprehension check

1. Consider this interaction at the shell:

```
>>> shopping_list = ['lettuce', 'hummus', 'lentils']
>>> shopping_list.extend('lemon')
```

This succeeds, but what is the resulting list?

2. There's a way to append a single element to a list with `.extend()`. Can you produce an example?

3. What do you think happens here?

```
>>> numbers = [1, 2, 3]
>>> numbers.extend(range(4, 7))
```

11.9 Calculating a sum in a loop

While we have the Python built-in function `sum()` which sums the elements of a sequence (provided the elements of the sequence are all of numeric type), it's instructive to see how we can do this in a loop (in fact, summing in a loop is *exactly* what the `sum()` function does).

Here's a simple example.

```
t = (27, 3, 19, 43, 11, 9, 31, 36, 75, 2)

sum_ = 0

for n in t:
    sum_ += n

print(sum_)             # prints 256
assert sum_ == sum(t)   # verify answer
```

We begin with a tuple of numeric values, `t`. Since the elements of `t` are all numeric, we can calculate their sum. First, we create a variable to hold the result of the sum. We call this, `sum_`.[6] Then, we iterate over all the elements in `t`, and at each iteration of the loop, we add the value of each element to the variable `sum_`. Once the loop has terminated, `sum_` holds the sum of all the elements of `t`. Then we print, and compare with the result returned by `sum(t)` to verify this is indeed the correct result.

In calculations like this we call the variable, `sum_`, an *accumulator* (because it accumulates the values of the elements in the iteration).

That's how we calculate a sum in a loop!

[6] Why do we use the trailing underscore? To avoid overwriting the Python built-in function `sum()` with a new definition.

11.10 Loops and summations

It is often the case that we have some formula which includes a *summation*, and we wish to implement the summation in Python.

For example, the formula for calculating the arithmetic mean of a list of numbers requires that we first sum all the numbers:

$$\mu = \frac{1}{N} \sum_{i=0}^{N-1} x_i$$

If you've not seen the symbol $\sum$ before, it's just shorthand for "add them all up."

What's the connection between summations and loops? The summation *is a loop*!

In the formula above, there's some list of values, x indexed by i. The summation says: "Take all the elements, x_i, and sum them." The summation portion is just

$$\sum_{i=0}^{N-1} x_i$$

which is the same as

$$x_0 + x_1 + ... + x_{N-2} + x_{N-1}$$

Here's the loop in Python (assuming we have some list called `x`):

```
s = 0
for e in x:
    s = s + e
```

after which, we'd divide by the number of elements in the list:

```
mean = s / len(x)
```

Yes, we could calculate the sum with `sum()` but what do you think `sum()` does behind the scenes? Exactly this!

Here's another. Let's say we wanted to calculate the sum of the squares of a list of numbers (which is common enough). Here's the summation notation (again using zero indexing):

$$\sum_{i=0}^{N-1} x_i^2$$

Here's the loop in Python:

```
s = 0
for e in x:
    s = s + e ** 2
```

See? The connection between summations and loops is straightforward.

11.11 Products

The same applies to products. Just as we can sum by adding all the elements in some list or tuple of numerics, we can also take their product by multiplying. For this, instead of the symbol $\sum$, we use the symbol Π (that's an upper-case Π to distinguish it from the constant π).

$$\prod_{i=0}^{N-1} x_i$$

This is the same as

$$x_0 \times x_1 \times ... \times x_{N-2} \times x_{N-1}$$

The corresponding loop in Python:

```
p = 1
for e in x:
    p = p * e
```

Why do we initialize the accumulator to 1? Because that's the multiplicative identity. If we set this equal to zero the product would be zero, because anything multiplied by zero is zero. Anything multiplied by one is itself. Thus, if calculating a repeated product, we initialize the accumulator to one.

11.12 `enumerate()`

We've seen how to iterate over the elements of an iterable in a `for` loop.

```
for e in lst:
    print(e)
```

Sometimes, however, it's helpful to have both the element and the *index* of the element at each iteration. One common application requiring an element and the index of the element is in calculating an *alternating sum*. Alternating sums appear in analysis, number theory, combinatorics, many with real-world applications.

An alternating sum is simply a summation where the signs of terms alternate. Rather than

$$x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + ...$$

where the signs are all positive, an alternating sum would look like this:

$$x_0 - x_1 + x_2 - x_3 + x_4 - x_5 + ...$$

Notice that we alternate addition and subtraction.

There are many ways we could implement this. Here's one rather clunky example (which assumes we have a list of numerics named `lst`):

```
alternating_sum = 0

for i in range(len(lst)):
    if i % 2:  # i is odd, then we subtract
        alternating_sum -= lst[i]
    else:
        alternating_sum += lst[i]
```

This works. Strictly speaking from a mathematical standpoint it is correct, but `for i in range(len(lst))` and then using `i` as an index into `lst` is considered an "anti-pattern" in Python. (Anti-patterns are patterns that we should avoid.)

So what's a programmer to do?

`enumerate()` to the rescue! Python provides us with a handy built-in function called `enumerate()`. This iterates over all elements in some sequence and yields a tuple of the index and the element at each iteration.

Here's the same loop implemented using `enumerate()`:

```
alternating_sum = 0

for i, e in enumerate(lst):
    if i % 2:
        alternating_sum -= e
    else:
        alternating_sum += e
```

We do away with having to call `len(list)` and we do away with indexed reads from `lst`.

Here's another example, where we wish to perform indexed writes into a list in a loop. Say we wanted to increment every element in a list of numeric values by a constant. Here's how we can do this with `enumerate()`.

```
incr = 5
lst = [1, 2, 3, 4]

for i, e in enumerate(lst):
    lst[i] = e + incr
```

That's it! After this code has run, `lst` has the value `[6, 7, 8, 9]`.

In many cases, use of `enumerate()` leads to cleaner and more readable code. But how does it work? What, exactly, does `enumerate()` do?

If we pass some iterable—say a list, tuple, or string—as an argument to `enumerate()` we get a new iterable object back, one of type `enumerate` (this is a new type we haven't seen before). When we iterate over an enumerate object, it yields *tuples*. The first element of the tuple is the index of the element in the original iterable. The second element of the tuple is the element itself.

That's a lot to digest at first, so here's an example:

```
lst = ['a', 'b', 'c', 'd']
for i, element in enumerate(lst):
    print(f"The element at index {i} is '{element}'.")
```

This prints:

```
The element at index 0 is 'a'.
The element at index 1 is 'b'.
The element at index 2 is 'c'.
The element at index 3 is 'd'.
```

The syntax that we use above is *tuple unpacking* (which we saw in an earlier chapter). When using `enumerate()` this comes in really handy. We use one variable to hold the index and one to hold the element. `enumerate()` yields a tuple, and we unpack it on the fly to these two variables.

Let's dig a little deeper using the Python shell.

```
>>> lst = ['a', 'b', 'c']
>>> en = enumerate(lst)
>>> type(en)       # verify type is `enumerate`
<class 'enumerate'>
>>> for t in en:   # iterate `en` without tuple unpacking
...    print(t)
...
(0, 'a')
(1, 'b')
(2, 'c')
```

So you see, what's yielded at each iteration is a tuple of index and element. Pretty cool, huh?

Now that we've learned a little about `enumerate()` let's revisit the alternating sum example:

```
alternating_sum = 0

for i, e in enumerate(lst):
    if i % 2:
        alternating_sum -= e
    else:
        alternating_sum += e
```

Recall that we'd assumed `lst` is a previously defined list of numeric elements. When we pass `lst` as an argument to `enumerate()` we get an `enumerate` object. When we iterate over this object, we get tuples at each iteration. Here we unpack them to variables `i` and `e`. `i` is assigned the index, and `e` is assigned the element.

If you need both the element and its index, use `enumerate()`.

11.13 Tracing a loop

Oftentimes, we wish to understand the behavior of a loop that perhaps we did not write. One way to suss out a loop is to use a table to *trace* the execution of the loop. When we do this, patterns often emerge, and—in the case of a `while` loop—we understand better the termination criteria for the loop.

Here's an example. Say you were asked to determine the value of the variable `s` after this loop has terminated:

```
s = 0
for n in range(1, 10):
    if n % 2:
        # n is odd; 1 is truthy
        s = s + 1 / n
    else:
        # n must be even; 0 is falsey
        s = s - 1 / n
```

Let's make a table, and fill it out. The first row in the table will represent our starting point, subsequent rows will capture what goes on in the loop. In this table, we need to keep track of two things, `n` and `s`.

n	s
	0

Before we enter the loop, `s` has the value `0`.

Now consider what values we'll be iterating over. `range(1, 10)` will yield the values 1, 2, 3, 4, 5, 6, 7, 8 and 9. So let's add these to our table (without calculating values for `s` yet).

n	s
	0
1	?
2	?
3	?
4	?
5	?
6	?
7	?
8	?
9	?

Since there are no `break` or `return` statements, we know we'll iterate over all these values of `n`.

Now let's figure out what happens to `s` within the loop. At the first iteration, `n` will be 1, which is odd, so the `if` branch will execute. This will add `1 / n` to `s`, so at the end of the first iteration, `s` will equal 1 (`1 / 1`). So we write that down in our table:

n	s
	0
1	1
2	?
3	?
4	?
5	?
6	?
7	?
8	?
9	?

Now for the next iteration. At the next iteration, `n` takes on the value 2. Which branch executes? Well, 2 is even, so the `else` branch will execute and 1/2 will be subtracted from `s`. Let's not perform decimal expansion, so we can write:

n	s
	0
1	1
2	$1 - 1/2$
3	?
4	?
5	?
6	?
7	?
8	?
9	?

Now for the next iteration. `n` takes on the value 3, 3 is odd, and so the `if` branch executes and we add 1/3 to `s`. Again, let's not perform decimal expansion (not doing so will help us see the pattern that will emerge).

n	s
	0
1	1
2	$1 - 1/2$
3	$1 - 1/2 + 1/3$
4	?
5	?
6	?
7	?
8	?
9	?

Now for the next iteration. `n` takes on the value 4, 4 is even, and so the `else` branch executes and we subtract 1/4 to `s`.

n	s
	0
1	1
2	$1 - 1/2$
3	$1 - 1/2 + 1/3$
4	$1 - 1/2 + 1/3 - 1/4$
5	?
6	?
7	?
8	?
9	?

Do you see where this is heading yet? No? Let's do a couple more iterations.

At the next iteration. `n` takes on the value 5, 5 is odd, and so the `if` branch executes and we add 1/5 to `s`. Then `n` takes on the value 6, 6 is even, and so the `else` branch executes and we subtract 1/6 to `s`.

n	s
	0
1	1
2	$1 - 1/2$
3	$1 - 1/2 + 1/3$
4	$1 - 1/2 + 1/3 - 1/4$
5	$1 - 1/2 + 1/3 - 1/4 + 1/5$
6	$1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6$
7	?
8	?
9	?

At this point, it's likely you see the pattern (if not, just work out two or three more iterations). This loop is calculating

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} + \frac{1}{9}$$

See? At each iteration, we're checking to see if `n` is even or odd. If `n` is odd, we add `1 / n`; if `n` is even, we subtract `1 / n`.

We can write this more succinctly using summation notation. This loop calculates

$$s = \sum_{n=1}^{n=9} (-1)^{n-1} \frac{1}{n}.$$

You may ask yourself: What's up with the $(-1)^{n-1}$ term? That's handling the alternating sign!

- What's $(-1)^0$? 1.
- What's $(-1)^1$? -1.
- What's $(-1)^2$? 1.

- What's $(-1)^3$? -1.
- What's $(-1)^4$? 1.

Another example: factorial

In mathematics, the *factorial* of a natural number, n is the product of all the natural numbers up to and including n. It is written with an exclamation point, for example,

$$6\,! = 1 \times 2 \times 3 \times 4 \times 5 \times 6.$$

Let's trace a Python loop which calculates factorial.

```
n = 6

f = 1
for i in range(2, n + 1):
    f = f * i
```

What does this loop do? At each iteration, it multiplies f by i and makes this the new f.

i	f
	1
2	2
3	6
4	24
5	120
6	720

This calculates factorial, $n!$ for some n. (Yes, there are easier ways.) Remember:

$$n\,! = \prod_{i=1}^{i=n} i.$$

Another example: discrete population growth

```
birth_rate = 0.05
death_rate = 0.03
pop = 1000  # population
n = 4

for _ in range(n):
    pop = int(pop * (1 + birth_rate - death_rate))
```

_	p
	1000
1	1020
2	1040
3	1060
4	1081

Here we don't use the value of the loop index in our calculations, but we include it in our table just to keep track of which iteration we're on. In this example, we multiply the old `pop` by `(1 + birth_rate - death_rate)` and make this the new `pop` at each iteration. This one's a nuisance to work out by hand, but with a calculator it's straightforward.

What is this calculating? This is calculating the size of a population which starts with 1000 individuals, and which has a birth rate of 5% and a death rate of 3%. This calculates the population after four time intervals (for example, years).

Being able to trace through a loop (or any portion of a program) is a useful skill for a programmer.

11.14 Nested loops

It's not uncommon that we include one loop within another. This is called *nesting*, and such loops are called *nested loops*.

Let's say we wanted to print out pairings of contestants in a round robin chess tournament (a "round robin" tournament is one in which each player plays each other player). Because in chess white has a slight advantage over black, it's fair that each player should play each other player twice: once as black and once as white.

We'll represent each game with the names of the players, in pairs, where the first player listed plays white and the second plays black.

So in a tiny tournament with players Anne, Bojan, Carlos, and Doris, we'd have the pairings:

```
Anne (W) vs Bojan (B)
Anne (W) vs Carlos (B)
Anne (W) vs Doris (B)
Bojan (W) vs Anne (B)
Bojan (W) vs Carlos (B)
Bojan (W) vs Doris (B)
Carlos (W) vs Anne (B)
Carlos (W) vs Bojan (B)
Carlos (W) vs Doris (B)
Doris (W) vs Anne (B)
Doris (W) vs Bojan (B)
Doris (W) vs Carlos (B)
```

(we exclude self-pairings for obvious reasons).

Given the list of players, `['Anne', 'Bojan', 'Carlos', 'Doris']`, how could we write code to generate all these pairings? One way is with a nested loop.

```
players = ['Anne', 'Bojan', 'Carlos', 'Doris']

for white in players:
   for black in players:
       if white != black:  # exclude self-pairings
           print(f"{white} (W) vs {black} (B)")
```

This code, when executed, prints exactly the list of pairings shown above.

How does this work? The outer loop—`for white in players:`—iterates over all players, one at a time: Anne, Bojan, Carlos, and Doris. For each iteration of the outer loop, there's an iteration of the inner loop, again: Anne, Bojan, Carlos, and Doris. If the element assigned to `white` in the outer loop does not equal the element assigned to `black` in the inner loop, we print the pairing. In this way, all possible pairings are generated.

Here's another example—performing multiplication using a nested loop. (What follows is inefficient, and perhaps a little silly, but hopefully it illustrates the point.)

Let's say we wanted to multiply 5×7 without using the $\star$ operator. We could do this with a nested loop!

```
answer = 0

for _ in range(5):
    for __ in range(7):
        answer += 1

print(answer)   # prints 35
```

How many times does the outer loop execute? Five. How many times does the inner loop execute *for each iteration of the outer loop*? Seven. How many times do we increment `answer`? $5 \times 7 = 35$.

Using nested loops to iterate a two-dimensional list

Yikes! What's a two-dimensional list? A two-dimensional list is just a list containing other lists!

Let's say we have the outcome of a game of tic-tac-toe encoded in a two-dimensional list:

```
game = [
    ['X', ' ', 'O'],
    ['O', 'X', 'O'],
    ['X', ' ', 'X']
]
```

To print this information in tabular form we can use a nested loop.

```
for row in game:
    for col in row:
        print(col, end='')
    print()
```

This prints

```
X O
OXO
X X
```

11.15 Stacks and queues

Now that we've seen lists and loops, it makes sense to present two fundamental data structures: the *stack* and the *queue*. You'll see that implementing these in Python is almost trivial—we use a list for both, and the only difference is *how* we use the list.

Stacks

A *stack* is what's called a last-in, first-out data structure (abbreviated LIFO and pronounced "life-o"). It's a linear data structure where we add elements to a list at one end, and remove them from *the same end*.

The canonical example for a stack is cafeteria trays. Oftentimes these are placed on a spring-loaded bed, and cafeteria customers take the tray off the top and the next tray in the stack is exposed. The first tray to be put on the stack is the last one to be removed. You've likely seen chairs that can be stacked. The last one on is the first one off. Have you ever packed a suitcase? What's gone in last is the first to come out. Have you ever hit the 'back' button in your web browser? Web browsers use a stack to keep track of the pages you've visited. Have you ever used `ctrl-z` to undo an edit to a document? Do you have a stack of dishes or bowls in your cupboard? Guess what? These are all everyday examples of stacks.

We refer to appending an element to a stack as *pushing*. We refer to removing an element to a stack as *popping* (this should sound familiar).

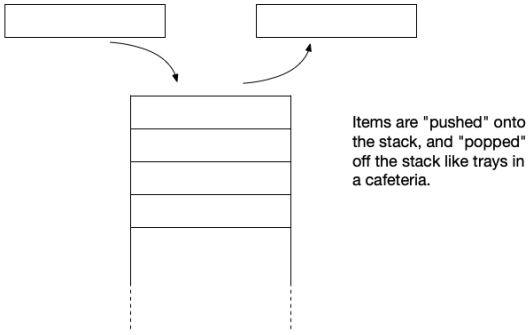


Figure 11.8: A stack

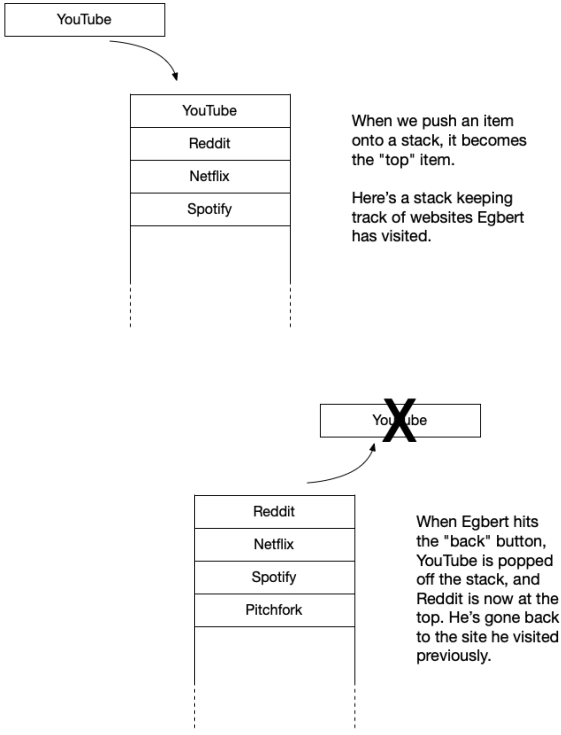


Figure 11.9: Push and pop

Stacks are very widely used in computer science and stacks are at the heart of many important algorithms. (In fact, function calls are managed using a stack!)

The default behavior for a list in Python is to function as a stack. Yes, that's right, we get stacks for free! If we append an item to a list, it's appended at one end. When we pop an item off a list, by default, it pops *from the same end*. So the last element in a Python list represents the *top* of the stack.

Here's a quick example:

```
>>> stack = []
>>> stack.append("Pitchfork")
>>> stack.append("Spotify")
>>> stack.append("Netflix")
>>> stack.append("Reddit")
>>> stack.append("YouTube")
>>> stack[-1]  # see what's on top
YouTube
>>> stack.pop()
YouTube
>>> stack[-1]  # see what's on top
Reddit
>>> stack.pop()
Reddit
>>> stack[-1]  # see what's on top
Netflix
```

So you see, implementing a stack in Python is a breeze.

Queues

A queue is a first-in, first-out linear data structure (FIFO, pronounced "fife-o"). The only difference between a stack and a queue is that with a stack we push and pop items from the same end, and with a queue we add elements at one end and remove them from the other. That's the only difference.

What are some real world examples? The checkout line at a grocery store—the first one in line is the first to be checked out. Did you ever wait at a printer because someone had sent a job before you did? That's another queue. Cars through a toll booth, wait lists for customer service chats, and so on—real world examples abound.

The terminology is a little different. We refer to appending an element to a queue as *enqueueing*. We refer to removing an element to a queue as *dequeueing*. But these are just fancy names for appending and popping.

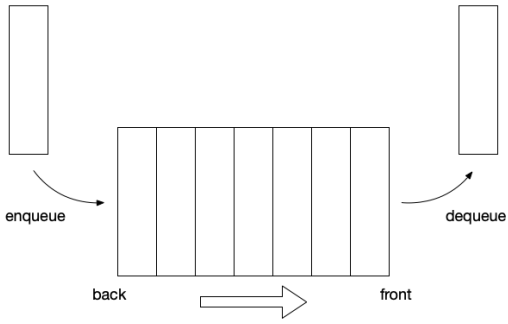


Figure 11.10: A queue

Like stacks, queues are very widely used in computer science and are at the heart of many important algorithms.

There's one little twist needed to turn a list into a queue. With a queue, we enqueue from one end and dequeue from the other. Like a stack, we can use append to enqueue. The little twist is that instead of `.pop()` which would pop from the same end, we use `.pop(0)` to pop from the other end of the list, and *voilà*, we have a queue.

Here's a quick example:

```
queue = []
>>> queue.append("Fred")    # Fred is first in line
>>> queue.append("Mary")    # Mary is next in line
>>> queue.append("Claire")  # Claire ie behind Mary
>>> queue.pop(0)            # Fred has been served
'Fred'
>>> queue[0]                # now see who's in front
'Mary'
>>> queue.append("Gladys")  # Gladys gets in line
>>> queue.pop(0)            # Mary's been served
'Mary'
```

So you see, a list can be used as a stack or a queue. Usually, stacks and queues are used within a loop. We'll see a little more about this in a later chapter.

11.16 A deeper dive into iteration in Python

What follows is a bit more detail about iterables and iteration in Python. You can skip this section entirely if you wish. This is presented here for the sole purpose of demonstrating what goes on "behind the scenes" when we iterate over some object in Python. With that said, let's start with the case of a list (it works the same with a tuple or any other iterable).

```
>>> m = ['Greninja', 'Lucario', 'Mimikyu', 'Charizard']
```

When we ask Python to iterate over some iterable, it calls the function `iter()` which returns an *iterator* for the iterable (in this case a list).

```
>>> iterator = iter(m)
>>> type(iterator)
<class 'list_iterator'>
```

The way iterators work is that Python keeps asking "give me the next member", until the iterator is exhausted. This is done (behind the scenes) by calls to the iterator's `__next__()` method.

```
>>> iterator.__next__()
'Greninja'
>>> iterator.__next__()
'Lucario'
>>> iterator.__next__()
'Mimikyu'
>>> iterator.__next__()
'Charizard'
```

Now what happens if we call `__next__()` one more time?
We get a `StopIteration` error

```
>>> iterator.__next__()
Traceback (most recent call last):
  File "/Library/.../code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
StopIteration
```

Again, behind the scenes, when iterating through an iterable, Python will get an iterator for that object, and then call `__next__()` until a `StopIteration` error occurs, and then it stops iterating.

What about `range()`?

As we've seen earlier, `range()` returns an object of the `range` type.

```
>>> r = range(5)
>>> type(r)
<class 'range'>
```

But ranges are iterable, so they work with the same function `iter(r)`, and the resulting iterator will have `__next__()` as a method.

```
>>> iterator = iter(r)
>>> iterator.__next__()
0
>>> iterator.__next__()
1
>>> iterator.__next__()
2
>>> iterator.__next__()
3
>>> iterator.__next__()
4
>>> iterator.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

When Python iterates over a sequence (list, tuple, string) or a `range` object, it first creates an iterator, then iterates the iterator.

It's important to note that `enumerate` objects are themselves iterators. *This is why an enumerate object can be iterated only once.* Python does not create a new iterator for these objects automatically, as it does when iterating sequences or `range` objects. (We'll see something similar later on when we get to `csv.reader` objects, but that's for another time.)

11.17 Exercises

Exercise 01

Write a `while` loop that adds all numbers from one to ten. Do not use `for`. What is the sum? Check your work. (See: section 11.2)

Exercise 02

Without a loop but using `range()`, calculate the sum of all numbers from one to ten. Check your work. (See: section 11.5)

Exercise 03

a. Write a `for` loop that prints the numbers 0, 2, 4, 6, 8, each on a separate line.

b. Write a `for` loop that prints some sentence of your choosing five times.

Exercise 04

Write a `for` loop that calculates the sum of the squares of some arbitrary list of numerics, named `data` (make up your own list, but be sure that it has at least four elements).

For example, if the list of numerics were `[2, 9, 5, -1]`, the result would be 111, because

$$
\begin{aligned}
2 \times 2 &= 4 \\
9 \times 9 &= 81 \\
5 \times 5 &= 25 \\
-1 \times -1 &= 1
\end{aligned}
$$

and $4 + 81 + 25 + 1 = 111$.

(See: section 11.6)

Exercise 05

Write a `for` loop that iterates the list

```
lst = [23, 7, 42, 17, 9, 38, 28, 31, 49, 22, 5, 26, 15]
```

and prints the *parity sum* of the list. That is, if the number is even, add it to the total; if the number is odd, subtract it from the total.

What is the sum? Check your work. Double-check your work.

(See: section 11.11)

Exercise 06

Write a `for` loop which iterates over a string and capitalizes every other letter. For example, with the string "Rumplestiltskin", the result should be "RuMpLeStIlTsKiN". With the string "HELLO WORLD!", the result should be "HeLlO WoRlD!" With the string "123456789", the result should be "123456789".

What happens if we capitalize a space or punctuation or number?

Exercise 07

Create some list of your own choosing. Your list should contain at least five elements. Once you've created your list, write a loop that uses `enumerate()` to iterate over your list, yielding both index and element at each iteration. Print the results indicating the element and its index.

For example, given the list

```
albums = ['Rid Of Me', 'Spiderland', 'This Stupid World',
          'Icky Thump', 'Painless', 'New Long Leg"]
```

your program would print

```
"Rid Of Me" is at index 0.
"Spiderland" is at index 1.
"This Stupid World" is at index 2.
"Icky Thump" is at index 3.
"Painless" is at index 4.
"New Long Leg" is at index 5.
```

Exercise 08 (challenge!)

The *Fibonacci sequence* is a sequence of integers, starting with 0 and 1, such that after these first two, each successive number in the sequence is the sum of the previous two numbers. So, for example, the next number is 1 because $0 + 1 = 1$, the number after that is 2 because $1 + 1 = 2$, the number after that is 3 because $1 + 2 = 3$, the number after that is 5 because $2 + 3 = 5$, and so on.

Write a program that uses a loop (not recursion) to calculate the first n terms of the Fibonacci sequence. Start with this list:

```
fibos = [0, 1]
```

You may use one call to `input()`, one `if` statement, and one `while` loop. You may not use any other loops. You may not use recursion. Examples:

```
Enter n for the first n terms in the Fibonacci sequence: 7
[0, 1, 1, 2, 3, 5, 8]
```

```
Enter n for the first n terms in the Fibonacci sequence: 10
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
Enter n for the first n terms in the Fibonacci sequence: 50
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657,
 46368, 75025, 121393, 196418, 317811, 514229, 832040,
 1346269, 2178309, 3524578, 5702887, 9227465, 14930352,
 24157817, 39088169, 63245986, 102334155, 165580141,
 267914296, 433494437, 701408733, 1134903170, 1836311903,
 2971215073, 4807526976, 7778742049]
```

Exercise 09

Write a function that takes a list as an argument and modifies the list in a loop. How you modify the list is up to you, but you should use at least two different list methods. Include code that calls the function, passing in a list variable, and then demonstrates that the list has changed, once the function has returned.

Exercise 10

Write a function which takes two integers, n and k as arguments, and produces a list of all odd multiples of k between 1 and n. *E.g.*, given input $n = 100$ and $k = 7$, the function should return

```
[7, 21, 35, 49, 63, 77, 91]
```

Exercise 11 (challenge!)

The modulo operator *partitions* the integers into *equivalence classes* based on their *residues* (remainders) with respect to some modulus. For example, with a modulus of three, the integers are partitioned into three equivalence classes: those for which $n \mod 3 \equiv 0$, $n \mod 3 \equiv 1$, and $n \mod 3 \equiv 2$.

Write and test a function which takes as arguments an arbitrary list of integers and some modulus, n, and returns a tuple containing the count of elements in the list in each equivalence class, where the index of the tuple elements corresponds to the residue. So, for example, if the input list were `[1, 5, 8, 2, 11, 15, 9]` and the modulus were 3, then the function should return the tuple `(2, 1, 4)`, because there are two elements with residue 0 (15 and 9), one element with residue 1, (1), and four elements with residue 2 (5, 8, 2, 11).

Notice also that if the modulus is n, the value returned will be an n-tuple.

Exercise 12

Use a Python list as a stack. Start with an empty list:

```
stack = []
```

1. push 'teal'
2. push 'magenta'
3. push 'yellow'
4. push 'viridian'
5. pop
6. pop
7. push 'amber'
8. pop
9. pop
10. push 'vermilion'

Print your stack. Your stack should look like this

```
['teal', 'vermilion']
```

If it does not, start over and try again.

See: Section 11.14 Stacks and queues

Exercise 13

At the Python shell, create a list and use it as a queue. At the start, the queue should be empty.

```
>>> queue = []
```

Now perform the following operations:

1. enqueue ‘red’
2. enqueue ‘blue’
3. dequeue
4. enqueue ‘green’
5. dequeue
6. enqueue ‘ochre’
7. enqueue ‘cyan’
8. enqueue ‘umber’
9. dequeue
10. enqueue ‘mauve’

Now, print your queue. Your queue should look like this:

```
['ochre', 'cyan', 'umber', 'mauve']
```

If it doesn’t, start over and try again.

Chapter 12

Randomness, games, and simulations

Here we will learn about some of the abundant uses of randomness. Randomness is useful in games (shuffle a deck, roll a die), but it's also useful for modeling and simulating a staggering variety of real world processes.

Learning objectives

- You will understand why it is useful to be able to generate pseudo-random numbers or make pseudo-random choices in a computer program.
- You will learn how to use some of the most commonly-used methods from Python's `random` module, including
  - `random.random()` to generate a pseudo-random a floating point number in the interval $[0.0, 1.0)$,
  - `random.randint()` to generate a pseudo-random integer in a specified interval,
  - `random.choice()` to make a pseudo-random selection of an item from an iterable object,
  - `random.shuffle()` to shuffle a list,
  - `random.sample()` to sample k elements from a population without replacement, and
  - `random.gauss()` to sample from a normal (Gaussian) distribution given mean and standard deviation.
- You will understand the role of a *seed* in the generation of pseudo-random numbers, and understand how setting a seed makes predictable the behavior of a program incorporating pseudo-randomness.

Terms introduced

- bell curve
- deterministic
- gambler's ruin
- Mersenne twister
- Monte Carlo simulation
- normal (Gaussian) distribution
- pseudo-random
- `random` module
- random walk
- sample
- seed
- shuffle

For *mean* and *standard deviation*, see Chapter 14: Data Analysis and Presentation.

12.1 The `random` module

Consider all the games you've ever played that involve throwing dice or shuffling a deck of cards. Games like this are fun in part because of the element of chance. We don't know how many dots will come up when we throw the dice. We don't know what the next card to be dealt will be. If we knew all these things in advance, such games would be boring!

Outside of games, there's a tremendous variety of applications which require randomness.

Simulations of all kinds make use of this—for example, modeling biological or ecological phenomena, statistical mechanics and physics, physical chemistry, modeling social or economic behavior of humans, operations research, and climate modeling. Randomness is also used in cryptography, artificial intelligence, and many other domains. For example, the Monte Carlo method (named after the famous casino in Monaco) is a widely used technique which repeatedly samples data from a random distribution, and has been used in science and industry since the 1940s.

Python's `random` module gives us many methods for generating "random" numbers or making "random" choices. These come in handy when we want to implement a game of chance (or game with some chance element) or simulation.

But think: how would you write code that simulates the throw of a die or picks a "random" number between, say, one and ten? Really. Stop for a minute and give it some thought. This is where the `random` module comes in. We can use it to simulate such events.

I put "random" in quotation marks (above) because true randomness cannot be calculated. What the Python random module does is generate *pseudo-random* numbers and make *pseudo-random* choices. What's the difference? To the casual observer, there is no difference. However, deep down there are *deterministic* (non-random) processes behind the generation of these pseudo-random numbers and making pseudo-random choices.

That sounds rather complicated, but using the `random` module isn't.

If we wish to use the `random` module, we first import it (just like we've been doing with the `math` module).

```
import random
```

Now we can use methods within the `random` module.

`random.choice()`

The `random.choice()` method takes an iterable and returns a pseudo-random choice from among the elements of the iterable. This is useful when selecting from a fixed set of possibilities. For example:

```
>>> import random
>>> random.choice(['heads', 'tails'])
'tails'
```

Each time we call `choice` this way, it will make a pseudo-random choice between 'heads' and 'tails', thus simulating a coin toss.

This works with any iterable.

```
>>> random.choice((1, 2, 3, 4, 5))
2
>>> random.choice(['A', 'K', 'Q', 'J', '10', '9', '8', '7', '6',
...                '5', '4', '3', '2'])
'7'
>>> random.choice(['rock', 'paper', 'scissors'])
'rock'
>>> random.choice(range(10))
4
```

It even works with a string as an iterable!

```
>>> random.choice("random")
'm'
```

Comprehension check

1. How could we use `random.choice()` to simulate the throw of a six-sided die?

2. How could we use `random.choice()` to simulate the throw of a twelve-sided die?

Using `random.choice()` for a random walk

The *random walk* is a process whereby we take steps along the number line in a positive or negative direction, at random.

Starting at 0, and taking five steps, choosing -1 or +1 at random, a walk might proceed like this: 0, -1, 0, 1, 2, 1. At each step, we move one

to the left (negative) or one to the right (positive). In a walk like this there are 2^n possible outcomes, where n is the number of steps taken.

Here's a loop which implements such a walk:

```
>>> position = 0
>>> for _ in range(5):
...     position = position + random.choice([-1, 1])
...
```

`random.random()`

This method returns the next pseudo-random floating point number in the interval $[0.0, 1.0)$. Note that the interval given here is in mathematical notation and is not Python syntax. Example:

```
x = random.random()
```

Here `x` is assigned a pseudo-random value greater than or equal to zero, and strictly less than 1.

What use is this? We can use this to simulate events with a certain probability, p. Recall that probabilities are in the interval $[0.0, 1.0]$, where 0.0 represents impossibility, and 1.0 represents certainty. Anything between these two extremes is interesting.

Comprehension check

1. How would we generate a pseudo-random number in the interval $[0.0, 10.0)$?

Using `random.random()` to simulate the toss of a biased coin

Let's say we want to simulate the toss of a slightly biased coin—one that's rigged to come up heads 53% of the time. Here's how we'd go about it.

```
if random.random() < 0.53:
    print("Heads!")
else:
    print("Tails!")
```

This approach is commonly used in simulations in physical or biological modeling, economics, and games.

What if you want to choose a pseudo-random floating point number in the interval $[-100.0, 100.0)$? No big deal. Remember `random.random()` gives us a pseudo-random number in the interval $[0.0, 1.0)$, so to get a value in the desired interval we simply subtract 0.5 (so the distribution is centered at zero) and multiply by 200 (to "stretch" the result).

```
x = (random.random() - 0.5) * 200
```

Comprehension check

1. How would we simulate an event which occurs with a probability of 1/4?
2. How would we generate a pseudo-random floating point number in the interval $[-2.0, 2.0)$?

`random.randint()`

As noted, we can use `random.choice()` to choose objects from some iterable. If we wanted to pick a number from one to ten, we could use

```
n = random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

This is correct, but it can get cumbersome. What if we wanted to choose a pseudo-random number between 1 and 1000? In cases like this, it's better to use `random.randint()`. This method takes two arguments representing the upper and lower bound (inclusive). Thus, to pick a pseudo-random integer between 1 and 1000:

```
n = random.randint(1, 1000)
```

Now we have, n, such that n is an integer, $n \geq 1$, and $n \leq 1000$.

`random.shuffle()`

Sometimes, we want to shuffle values, for example a deck of cards. `random.shuffle()` will shuffle a mutable sequence (for example, a list) in place. Example:

```
cards = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10',
         'J', 'Q', 'K']
random.shuffle(cards)
```

Now the cards are shuffled.

Comprehension check

1. `random.shuffle()` works with a list. Why wouldn't it work with a tuple? Would it work with a string?
2. Where's the bug in this code?

```
>>> import random
>>> cards = ['A', '2', '3', '4', '5', '6', '7', '8', '9',
...          '10', 'J', 'Q', 'K']
>>> cards = random.shuffle(cards)
>>> print(cards)
None
>>>
```

random.sample()

`random.sample()` is used to sample from a population, without replacement. Let's say we have a list of 26 students (`students`) and we want to select five at random (for whatever reason, perhaps creating a project team for example). In other words, we'd want to sample k elements from the list `students`, ensuring the same student can't be picked more than once, with $k = 5$. Here's how we'd do that with `random.sample()`:

```
>>> import random
>>> students = ['Archibald', 'Beverly', 'Chiara', 'Dolores',
...             'Egbert', 'Francesca', 'Grant', 'Harvey',
...             'Irene', 'Jackie', 'Kate', 'Lourdes', 'Mary',
...             'Nancy', 'Oscar', 'Piotr', 'Quinn', 'Raisa',
...             'Stephanie', 'Terence', 'Ursula', 'Victor',
...             'Wendy', 'Xavier', 'Yvette', 'Zora']
>>> random.sample(students, 5)
['Ursula', 'Archibald', 'Zora', 'Egbert', 'Grant']
```

We refer to the list of elements as the *population*, and the selected elements as the *sample*.

Sampling without replacement can be used to produce a shuffled *copy* of a list (`.shuffle()` shuffles a list in place).

```
>>> random.sample(students, 26)
['Quinn', 'Terence', 'Piotr', 'Francesca', 'Egbert',
 'Archibald', 'Raisa', 'Harvey', 'Grant', 'Lourdes',
 'Dolores', 'Wendy', 'Stephanie', 'Mary', 'Victor',
 'Xavier', 'Jackie', 'Kate', 'Ursula', 'Chiara',
 'Nancy', 'Zora', 'Irene', 'Beverly', 'Oscar', 'Yvette']
```

or generally, for any population of any size:

```
>>> random.sample(population, len(population))
```

Comprehension check

1. Given a list of 100 colors, *e.g.,* `['red', 'blue', ...]`, how would you sample 3 colors from this list?

2. `random.sample()` samples k elements from a population without replacement. Why, then, can the following occur?

```
>>> random.sample(students, 5)
['Zora', 'Terence', 'Beverly', 'Chiara', 'Jackie']
>>> random.sample(students, 5)
['Quinn', 'Egbert', 'Ursula', 'Terence', 'Raisa']
>>>
```

(Notice that Terence appears in both samples.)

3. Would `random.sample()` be appropriate for dealing cards from a deck in a two-player game? Why or why not?

`random.gauss()`

You've likely heard of a *normal distribution* or a *bell curve*. You can sample real values (floats) from a normal distribution using `random.gauss()`. A *Gaussian* or normal distribution is described with two values: the mean and the standard deviation. Let's say we wish to draw a sample from a normal distribution with a mean of 0.0 and a standard deviation of 1.0. Here's how we do that:

```
>>> random.gauss(0.0, 1.0)
0.6445342847271447
```

As of Python 3.11, the default values for mean and standard deviation are as shown: 0.0 and 1.0 respectively. So this works as well, sampling from the same distribution:

```
>>> random.gauss()
0.1889405684443505
```

For all versions of Python prior to 3.11, you'll need to supply arguments for mean and standard deviation in all cases.

Here's how we'd generate 100,000 samples using the default values.

```
samples = []
for _ in range(100_000):
    samples.append(random.gauss())
```

Here's what this might look like if plotted:

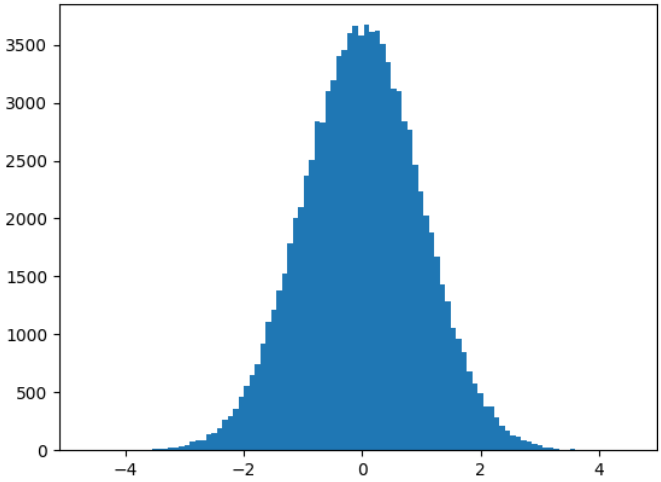


Figure 12.1: Histogram of a normal distribution

Comprehension check

1. Let's say we wanted to get one million samples from a normal distribution with a mean of 42.0 and a standard deviation of 5.0. How would you do this?

Other `random` methods

The random module includes many other methods which include generating random numbers sampled from various distributions, and other nifty tools!

If you are so inclined—especially if you have some probability theory and statistics under your belt—see: `random` — *Generate pseudo-random numbers*: https://docs.python.org/3/library/random.html

12.2 Pseudo-randomness in more detail

I mentioned earlier that the numbers generated and choices made by the `random` module aren't truly random, they're pseudo-random, but what does this mean?

Computers compute. They can't pluck a random number out of thin air. You might think that computation by your computer is deterministic and you'd be right.

So how do we use a deterministic computing device to produce something that appears random, something that has all the statistical properties we need?

Deep down, the `random` module makes use of an algorithm called the *Mersenne twister* (what a lovely name for an algorithm!).[1] You don't need to understand how this algorithm works, but it's useful to understand that it does require an input used as a starting point for its calculations. This input is called a *seed*, and from this, the algorithm produces a sequence of pseudo-random numbers. At each request, we get a new pseudo-random number.

```
>>> import random
>>> random.random()
0.16558225561225903
>>> random.random()
0.20717009610984627
>>> random.random()
0.2577426786448077
>>> random.random()
0.5173312574262303
>>>
```

Try this out. (The sequence of numbers you'll get will differ.)

So where does the seed come in? By default, the algorithm gets a seed from your computer's operating system. Modern operating systems provide a special source for this, and if a seed is not supplied in your code, the `random` module will ask the operating system to supply one.[2]

12.3 Using the seed

Most of the time we want unpredictability from our pseudo-random number generator (or choices). However, sometimes we wish to control the process a little more, for comparability of results.

For example, it would be difficult, if not impossible, to test a program whose output is unpredictable. This is why the `random` module allows us to provide our own seed. If we start the process from the same seed, the sequence of random numbers generated or the sequence of choices made is the same. For example,

```
>>> import random
>>> random.seed(42)  # Set the seed.
>>> random.random()
```

[1] M. Matsumoto and T. Nishimura, 1998, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", *ACM Transactions on Modeling and Computer Simulation*, 8(1).

[2] If you're curious, try this:

```
>>> import os       # interface to the operating system
>>> os.urandom(8)   # request a bytestring of size 8
b'\xa6t\x08=\xa5\x19\xde\x94'
```

This is where the `random` module gets its seed by default. This service itself requires a seed, which the OS gets from a variety of hardware sources. The objective is for the seed to be as unpredictable as possible.

```
0.6394267984578837
>>> random.random()
0.025010755222666936
>>> random.random()
0.27502931836911926
>>> random.seed(42)  # Set the seed again, to the same value.
>>> random.random()
0.6394267984578837
>>> random.random()
0.025010755222666936
>>> random.random()
0.27502931836911926
```

Notice that the sequence of numbers generated by successive calls to `random.random()` are identical: 0.6394267984578837, 0.025010755222666936, 0.27502931836911926, …

Here's another example:

```
>>> import random
>>> results = []
>>> random.seed('walrus')
>>> for _ in range(10):
...    results.append(random.choice(['a', 'b', 'c']))
...
>>> results
['b', 'a', 'c', 'b', 'a', 'a', 'a', 'c', 'a', 'b']
>>> results = []
>>> random.seed('walrus')
>>> for _ in range(10):
...    results.append(random.choice(['a', 'b', 'c']))
...
>>> results
['b', 'a', 'c', 'b', 'a', 'a', 'a', 'c', 'a', 'b']
```

Notice that the results are identical in both instances. If we were to perform this experiment 1,000,000 with the same seed, we'd always get the same result. It looks random, but deep down it isn't.

By setting the seed, we can make the behavior of calls to `random` methods *entirely predictable*. As you might imagine, this allows us to test programs that incorporate pseudo-random number generation or choices.

Try something similar with `random.shuffle()`. Start with a short list, set the seed, and shuffle it. Then re-initialize the list to its original value, set the seed again—with the same value—and shuffle it. Is the shuffled list the same in both cases?

12.4 Exercises

Exercise 01

Use `random.choice()` to simulate a fair coin toss. This method takes an iterable, and at each call, chooses one element of the iterable at random. For example,

```
random.choice([1, 2, 3, 4, 5])
```

will choose one of the elements of the list, each with equal probability.

In a loop simulate 10 coin tosses. Then report the number of heads and the number of tails.

Exercise 02

Use `random.random()` to simulate a fair coin toss. Remember that `random.random()` returns a floating point number in the interval $[0.0, 1.0)$.

In a loop simulate 10 coin tosses. Then report the number of heads and the number of tails.

Exercise 03

Simulate a biased coin toss. You may assume that, in the limit, the biased coin comes up heads 51.7% of the time. Unlike Exercise 01, `random.choice()` won't work because outcomes are not equally probable.

In a loop simulate 10 such biased coin tosses. Then report the number of heads and the number of tails.

Exercise 04

`random.shuffle()` takes some list as an argument and shuffles the list *in-place*. (Remember, lists are mutable, and shuffling in place means that `random.shuffle()` will modify the list you pass in and will return `None`.)

Write a program that shuffles the list

```
['A', 'K', 'Q', 'J', '10', '9', '8', '7', '6',
 '5', '4', '3', '2']
```

and then using `.pop()` in a `while` loop "deal the cards" one at a time until the list is exhausted. Print each card as it is popped from the list.

Exercise 05

The *gambler's ruin* simulates a gambler starting with some amount of money and gambling until they run out. Probability theory tells us they will *always* run out of money—it's just a matter of time.

Write a program which prompts the user for some amount of money and then simulates the gambler's ruin by betting on a fair coin toss. Use an integer value for the money, and wager one unit on each coin toss.

Your program should report the number of coin tosses it took the gambler to go bust.

Exercise 06

Write a program that simulates the throwing of two six-sided dice.

In a loop, simulate the throw, and report the results. For example, if the roll is a two and a five, print "2 + 5 = 7". Prompt the user, asking if they want to roll again or quit.

Exercise 07 (challenge!)

Write a program that prompts the user for a number of throws, n, and then simulates n throws of two six-sided dice. Record the total of dots for each throw. To record the number of dots use a list of integers. Start with a list of all zeros.

```
counts = [0] * 13
# This gets you a list of all zeros like this:
# [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
# We're going to ignore the element at index zero
```

Then, for each throw of the dice, calculate the total number of dots and increment the corresponding element in the list. For example, if the first three throws are five, five, and nine, then counts should look like this

```
[0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0]
```

After completing n rolls, print the result (again, ignoring the element at index 0) and verify with an assertion that the sum of the list equals n.

Exercise 08

In mathematics, one of the requirements of a function is that given the same input it produces the same output. Always.

For example, the square of 2 is always 4. You can't check back later and find that it's 4.1, or 9, or something else. Applying a function to the same argument will always yield the same result. If this is not the case, then it's not a function.

Question: Are the functions in the `random` module truly functions? If so, why? If not, why not?

Exercise 09 (challenge!)

Revisit the gambler's ruin from Exercise 05.

Modify the program so that it runs 1,000 simulations of the gambler's ruin, and keeps track of how many times it took for the gambler to run out of money. However—and this is important—always start with the same amount of money (say 1,000 units of whatever currency you like).

Then calculate the mean and standard deviation of the set of simulations.

The mean, written μ, is given by

$$\mu = \frac{1}{N} \sum_{i=0}^{N-1} x_i$$

where we have a set of outcomes, X, indexed by i, with N equal to the number of elements in X.

The standard deviation, written σ, is given by

$$\sigma = \frac{1}{N} \sum_{i=0}^{N-1} (x_i - \mu)^2.$$

Once that's done, run the program again, separately, for 10,000, 100,000, and 1,000,000 simulations, and record the results.

Does the mean change with the number of simulations? How about the standard deviation?

What does all this tell you about gambling?

Hint: It makes sense to write separate functions for calculating mean and standard deviation.

Chapter 13

File I/O

So far all of the input to our programs and all of the output from our programs has taken place in the console. That is, we've used `input()` to prompt for user input, and we've used `print()` to send output to the console.

Of course, there are a great many ways to send information to a program and to receive output from a program: mice and trackpads, audio data, graphical user interfaces (GUIs, pronounced "gooeys"), temperature sensors, accelerometers, databases, actuators, networks, web APIs (application program interfaces), and more..

Here we will learn how to read data from a file and write data to a file. We call this "file I/O" which is short for "file input and output." This is particularly useful when we have large amounts of data to process.

In order to read from or write to a file, we need to be able to open and close a file. We will do this using a *context manager*.

We will also see new exceptions which may occur when attempting to read from or write to a file, specifically `FileNotFoundError`.

Learning objectives

- You will learn how to read from a file.
- You will learn some of the ways to write to a file.
- You will learn some valuable keyword arguments.
- You will learn about the `csv` file format and Python module, as well as how to read from and write to a `.csv` file.

Terms and Python keywords introduced

- `as`
- context manager
- CSV (file format)
- `FileNotFoundError`
- keyword argument
- `next()`
- `UnicodeDecodeError`
- `with`

13.1 Context managers

A **context manager** is a Python object which controls (to a certain extent) what occurs within a `with` statement. Context managers relieve some of the burden placed on programmers. For example, if we open a file, and for some reason something goes wrong and an exception is raised, we still want to ensure that the file is closed. Before the introduction of the `with` statement (in Python 2.5, almost twenty years ago), programmers often used try/finally statements (we'll see more about `try` when we get to exception handling).

We introduce `with` and context managers in the context of file i/o, because this approach simplifies our code and ensures that when we're done reading from or writing to a file that the file is closed automatically, without any need to explicitly call the `.close()` method. The idiom we'll follow is:

```
with open("somefile.txt") as fh:
    # read from file
    s = fh.read()
```

When we exit this block (that is, when all the indented code has executed), Python will close the file automatically. Without this context manager, we'd need to call `.close()` explicitly, and failure to do so can lead to unexpected and undesirable results.

`with` and `as` are Python keywords. Here, `with` creates the context manager, and `as` is used to give a name to our file object. So once the file is opened, we may refer to it by the name given with `as`—in this instance `fh` (a common abbreviation for "file handle"). Note that what we create here (`fh`) is a new type of object with a scary sounding name: `_io.TextIOWrapper`. You don't have to worry about the details, just understand that what it's not a file itself, and it's not the contents of the file—it's an object through which we interact with a file.

13.2 Reading from a file

Let's say we have a `.txt` file called `hello.txt` in the same directory as a Python file we just created. We wish to open the file, read its content and assign it to a variable, and print that variable to the console. This is how we would do this:

```
>>> with open('hello.txt') as f:
...     s = f.read()
...
>>> print(s)
Hello World!
```

It's often useful to read one line at a time into a list.

```
>>> lines = []
>>> with open('poem.txt') as f:
...     for line in f:
...         lines.append(line)
...
>>> print(lines)
["Flood-tide below me!\n", "I see you face to face\n",
"Clouds of the west--\n", "Sun there half an hour high--\n",
"I see you also face to face.\n"]
```

Now, when we look at the data this way, we see clearly that newline characters are included at the end of each line. Sometimes we wish to remove this. For this we use the string method `.strip()`.

```
>>> lines = []
>>> with open('poem.txt') as f:
...     for line in f:
...         lines.append(line.strip())
...
>>> print(lines)
["Flood-tide below me!", "I see you face to face",
"Clouds of the west--", "Sun there half an hour high--",
"I see you also face to face."]
>>>
```

The string method `.strip()` without any argument removes any leading or trailing whitespace, newlines, or return characters from a string.

If you only wish to remove newlines (`'\n'`), just use `s.strip('\n')` where `s` is some string.

13.3 Writing to a file

So far, the only output our programs have produced is characters printed to the console. This is fine, as far as it goes, but often we have more output than we wish to read at the console, or we wish to store output for future use, distribution, or other purposes. Here we will learn how to write data to a file.

With Python, this isn't difficult. Python provides us with a built-in function `open()` which returns a file object. Then we can read from and write to the file, using this object.

The best approach to opening a file for writing is as follows:

```
with open('hello.txt', 'w') as f:
    f.write('Hello World!')
```

Let's unpack this one step at a time.

The `open()` function takes a file name, an optional *mode*, and other optional arguments we don't need to trouble ourselves with right now. So in the example above, `'hello.txt'` is the file name (here, a quoted

string), and `'w'` is the mode. You may have already guessed that `'w'` means "write", and if so, you're correct!

Python allows for other ways to specify the file, and `open()` will accept any "path-like" object. Here we'll only use strings, but be aware that there are other ways of specifying where Python should look for a given file.

There are a number of different modes, some of which can be using in combination. Quoting from the Python documentation:[1]

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open for updating (reading and writing)

Again, in the code snippet above, we specify `'w'` since we wish to write. We could have written:

```
with open('hello.txt', 'wt') as f:
    f.write('Hello World!')
```

explicitly specifying text mode, but this is somewhat redundant. We will only present reading and writing text data in this text.[2]

The idiom `with open('hello.txt', 'w') as f:` is the preferred approach when reading from or writing to files. We could write

```
f = open('hello.txt', 'w')
f.write('Hello World')
f.close()
```

but then it's our responsibility to close the file when done. The idiom `with open('hello.txt', 'w') as f:` will take care of closing the file automatically, as soon as the block is exited.

Now let's write a little more data. Here's a snippet from a poem by Walt Whitman (taking some liberties with line breaks):

```
fragment = ["Flood-tide below me!\n",
            "I see you face to face\n",
            "Clouds of the west--\n",
            "Sun there half an hour high--\n",
            "I see you also face to face.\n"]
```

[1] https://docs.python.org/3/library/functions.html#open

[2] It is often the case that we wish to write binary data to file, but doing so is outside the scope of this text.

```
with open('poem.txt', 'w') as fh:
    for line in fragment:
        fh.write(line)
```

Here we simply iterate through the lines in this fragment and write them to the file `poem.txt`. Notice that we include newline characters `'\n'` to end each line.

Writing numeric data

The `.write()` method requires a string, so if you wish to write numeric data, you should use `str()` or f-strings. Example:

```
import random

# Write 10,000 random values in the range [-1.0, 1.0)
with open('data.txt', 'w') as f:
    for _ in range(10_000):
        x = (random.random() - 0.5) * 2.0
        f.write(f"{x}\n")
```

Always use `with`

From the documentation:

> Warning: Calling `f.write()` without using the `with` keyword or calling `f.close()` might result in the arguments of `f.write()` not being completely written to the disk, even if the program exits successfully.

Since we can forget to call `f.close()`, use of `with` is the preferred (and most Pythonic) approach.

Comprehension check

1. Try the above code snippets to write to files `hello.txt` and `poem.txt`.
2. Write a program that writes five statements about you to a file called `about_me.txt`.

13.4 Keyword arguments

Some of what we'll do with files involves using *keyword arguments.*

Thus far, when we've called or defined functions, we've only seen *positional arguments.* For example, `math.sqrt(x)` and `list.append(x)` each take one positional argument. Some functions take two or more positional arguments. For example, `math.pow(x, y)`, takes *two* positional arguments. The first is the base, the second is the power. So this function returns `x` raised to the `y` power (x^y). Notice that what's significant here is not the

names of the formal parameters but their *order*. It matters how we supply arguments when calling this function. Clearly, 2^3 (8) is not the same as 3^2 (9). How does the function know which argument should be used as the base and which argument should be used as the exponent? It's all based on their position. The base is the first argument. The exponent is the second argument.

Some functions allow for *keyword* arguments. Keyword arguments follow positional arguments, and are given a name when calling the function. For example, `print()` allows you to provide a keyword argument `end` which can be used to override the default behavior of `print()` which is to append a newline character, `\n`, with every call. Example:

```
print("Cheese")
print("Shop")
```

prints "Cheese" and "Shop" on two different lines, because the default is to append that newline character. However…

```
print("Cheese", end=" ")
print("Shop")
```

prints "Cheese Shop" on a single line (followed by a newline), because in the first call to `print()` the `end` keyword argument is supplied with one blank space, `" "`, and thus, no newline is appended. This is an example of a keyword argument.

In the context of file input and output, we'll use a similar keyword argument when working with CSV files (comma separated values).

```
open('my_data.csv', newline='')
```

This allows us to avoid an annoying behavior in Python's CSV module in some contexts. We'll get into more detail on this soon, but for now, just be aware that we can, in certain cases, use keyword arguments where permitted, and that the syntax is as shown: positional arguments come first, followed by optional keyword arguments, with keyword arguments supplied in the form `keyword=value`. See: The `newline=''` keyword argument, below.

13.5 More on printing strings

Specifying the ending of printed strings

By default, the `print()` function appends a newline character with each call. Since this is, by far, the most common behavior we desire when printing, this default makes good sense. However, there are times when we do not want this behavior, for example when printing strings that are terminated with newline characters (`'\n'`) as this would produce *two* newline characters at the end. This happens often when reading certain data from a file. In this case, and in others where we wish to override the default behavior of `print()`, we can supply the keyword argument,

`end`. The `end` keyword argument specifies the character (or characters) if any, we wish to append to a printed string.

The `.strip()` method

Sometimes—especially when reading certain data from a file—we wish to remove whitespace, including spaces, tabs, and newlines from strings. One approach is to use the `.strip()` method. Without any argument supplied, `.strip()` removes all leading and trailing whitespace and newlines.

```
>>> s = '\nHello     \t   \n'
>>> s.strip()
'Hello'
```

Or you can specify the character you wish to remove.

```
>>> s = '\nHello     \t   \n'
>>> s.strip('\n')
'Hello     \t   '
```

This method allows more complex behavior (but I find the use cases rare). For more on `.strip()` see: https://docs.python.org/3/library/stdtypes.html?highlight=strip#str.strip

13.6 The `csv` module

There's a very common format in use for tabular data, the CSV or *comma separated value* format. Many on-line data sources publish data in this format, and all spreadsheet software can read from and write to this format. The idea is simple: columns of data are separated by commas. That's it!

Here's an example of some tabular data:

Year	FIFA champion
2018	France
2014	Germany
2010	Spain
2006	Italy
2002	Brazil

Here's how it might be represented in CSV format:

```
Year,FIFA champion
2018,France
2014,Germany
2010,Spain
2006,Italy
2002,Brazil
```

Pretty simple.

What happens if we have commas in our data? Usually numbers don't include comma separators when in CSV format. Instead, commas are added only when data are displayed. So, for example, we might have data like this (using format specifiers):

Country	2018 population
China	1,427,647,786
India	1,352,642,280
USA	327,096,265
Indonesia	267,670,543
Pakistan	212,228,286
Brazil	209,469,323
Nigeria	195,874,683
Bangladesh	161,376,708
Russia	145,734,038

and the CSV data would look like this:

```
Country,2018 population
China,1427647786
India,1352642280
USA,327096265
Indonesia,267670543
Pakistan,212228286
Brazil,209469323
Nigeria,195874683
Bangladesh,161376708
Russia,145734038
```

But what if we *really* wanted commas in our data?

Building	Address
Waterman	85 Prospect St, Burlington, VT
Innovation	82 University Pl, Burlington, VT

We'd probably break this into additional columns.

```
Building,Street,City,State
Waterman,85 Prospect St,Burlington,VT
Innovation,82 University Pl,Burlington,VT
```

What if we really, *really* had to have commas in our data? Oh, OK. Here are cousin David's favorite bands of all time:

Band	Rank
Lovin' Spoonful	1
Sly and the Family Stone	2
Crosby, Stills, Nash and Young	3
Earth, Wind and Fire	4
Herman's Hermits	5
Iron Butterfly	6
Blood, Sweat & Tears	7
The Monkees	8
Peter, Paul & Mary	9
Ohio Players	10

Now there's no way around commas in the data. For this we wrap the data including commas in quotation marks.

```
Band,Rank
Lovin' Spoonful,1
Sly and the Family Stone,2
"Crosby, Stills, Nash and Young",3
"Earth, Wind and Fire",4
Herman's Hermits,5
Iron Butterfly,6
"Blood, Sweat & Tears",7
The Monkees,8
"Peter, Paul & Mary",9
Ohio Players,10
```

(We'll save the case of data which includes commas *and* quotation marks for another day.)

We can read data like this using Python's `csv` module.

```
import csv
with open('bands.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(row)
```

This prints:

```
['Band', 'Rank']
["Lovin' Spoonful", '1']
['Sly and the Family Stone', '2']
['Crosby, Stills, Nash and Young', '3']
['Earth, Wind and Fire', '4']
["Herman's Hermits", '5']
['Iron Butterfly', '6']
['Blood, Sweat & Tears', '7']
['The Monkees', '8']
['Peter, Paul & Mary', '9']
['Ohio Players', '10']
```

Notice that we have to create a special object, a CSV reader (`_csv.reader`). We *instantiate* this object by calling the *constructor* function, `csv.reader()`, and we pass to this function the file *object* we wish to read. Notice also that we read each row of our data file into a list, where the columns are separated by commas. That's very handy!

We can write data to a CSV file as well.

```
import csv

bands = [['Deerhoof', 1],
         ['Lightning Bolt', 2],
         ['Radiohead', 3],
         ['Big Thief', 4],
         ['King Crimson', 5],
         ['French for Rabbits', 6],
         ['Yak', 7],
         ['Boygenius', 8],
         ['Tipsy', 9],
         ['My Bloody Valentine', 10]]

with open('bands.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    for item in bands:
        writer.writerow(item)
```

This writes

```
Deerhoof,1
Lightning Bolt,2
Radiohead,3
Big Thief,4
King Crimson,5
French for Rabbits,6
Yak,7
Boygenius,8
Tipsy,9
My Bloody Valentine,10
```

to the file.

The `newline=''` keyword argument

If you're using a Mac or a Linux machine, the `newline=''` keyword argument may not be strictly necessary when opening a file for use with a `csv` reader or writer. However, omitting it could cause problems on a Windows machine and so it's probably best to include it for maximum portability. The Python documentation recommends using it.

Iterating CSV reader objects

Unlike a list, tuple, or string, a CSV reader object can only be iterated once. This is because a CSV reader object is a type of *iterator* (which is different from an iterable).[3] An *iterator* can be iterated only once. Accordingly, if you wish to iterate over a CSV file more than once, you'll need to create a new CSV reader object each time. (Yes, there is another way to rewind and start again, but we won't cover that here.)

13.7 Exceptions

`FileNotFoundError`

This is just as advertised: an exception which is raised when a file is not found. This is almost always due to a typo or misspelling in the filename, or that the correct path is not included.

Suppose there is no file in our file system with the name `some_non-existent_file.foobar`. Then, if we were to try to open a file without creating it, we'd get a `FileNotFoundError`.

```
>>> with open("some_non-existent_file.foobar") as fh:
...     s = fh.read()
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
        'some_non-existent_file.foobar'
```

Usually we can fix this by supplying the correct filename or a complete path to the file.

`UnicodeDecodeError`

A `UnicodeDecodeError` occurs if we try to decode a string that was encoded in a different system. This can occur when reading data from a file or other source. Example (assuming `data.txt` was encoded with, say, Windows-1252 encoding):

```
with open("data.txt") as fh:
    s = fh.read()
```

will result in a `UnicodeDecodeError` if there are any invalid UTF-8 encodings in the input. To fix, specify the correct encoding:

```
with open("data.txt", encoding=cp1252) as fh:
    s = fh.read()
```

[3] When iterating over a static iterable, Python creates an iterator on-the-fly behind the scenes. However it does not do this when iterating over an iterator, since the iterator itself already exists.

For a more detailed explanation of Unicode, `UnicodeDecodeError`, and `UnicodeEncodeError`, see Appendix I: The joy of Unicode (but don't say I didn't warn you—the details may make your head spin).

13.8 Exercises

Exercise 01

Write a program which writes the following lines (including blank lines) to a file called `bashos_frog.txt`.

```
Basho's Frog

The old pond
A frog jumped in,
Kerplunk!

Translated by Allen Ginsberg
```

Once you've run your program, open `bashos_frog.txt` with a text editor or your IDE and verify it has been written correctly. If not, go back and revise your program until it works as intended.

Exercise 02

Download the text file at https://www.uvm.edu/~cbcafier/cs1210/book/data/random_floats.txt, and write a program which reads it and reports how many lines it contains.

Exercise 03

There's a bug in this code.

```
import csv

prices = []

with open("price_list.csv", 'r') as fh:
    reader = csv.reader(fh)
    next(reader)
    for row in reader:
        prices.append(row[1])

average_price = sum(prices) / len(prices)
print(average_price)
```

When run, this program halts on the exception "TypeError: unsupported operand type(s) for +: 'int' and 'str' ". You may assume that the file is well-formed CSV, with item description in the first field and price in USD in the second field. What's wrong, and how can you fix it?

Exercise 04

Write a program which writes 10 numbers to a file, closes the file, then reads the 10 numbers from the file, and verifies the correct result. (Feel free to use assertions to verify.)

Exercise 05

Here's a poem, which is saved with the filename `doggerel.txt`.

```
Roses are red.
Violets are blue.
I cannot rhyme.
Have you ever seen a wombat?
```

There's a bug in the following program, which is supposed to read the file containing a poem.

```
with open("doggerel.txt", 'r') as fh:
    for line in fh:
        print(line)
```

When run, this results in a blank line being printed between every line in the poem.

```
Roses are red.

Violets are blue.

I cannot rhyme.

Have you ever seen a wombat?
```

What's wrong, and how can you fix it?

Chapter 14

Data analysis and presentation

What follows in this chapter is merely a tiny peek into the huge topic of data analysis and presentation. This is not intended to be a substitute for a course in statistics. There are plenty of good textbooks on the subject (and plenty of courses at any university), so what's presented here is just a little something to get you started in Python.

Learning objectives

- You will gain a rudimentary understanding of two important descriptive statistics: the mean and standard deviation.
- You will understand how to calculate these statistics and be able to implement them on your own in Python.
- You will learn the basics of Matplotlib's Pyplot interface, and be able to use `matplotlib.pyplot` to create line, bar, and scatter plots.

Terms introduced

- arithmetic mean
- central tendency
- descriptive statistics
- Matplotlib
- normal distribution
- quantile (including quartile, quintile, percentile)
- standard deviation

14.1 Some elementary statistics

Statistics is the science of data—gathering, analyzing, and interpreting data.

Here we'll touch on some elementary *descriptive statistics*, which involve describing a collection of data. It's usually among the first things one learns within the field of statistics.

The two most widely used descriptive statistics are the *mean* and the *standard deviation*. Given some collection of numeric data, the mean gives us a measure of the *central tendency* of the data. There are several

different ways to calculate the mean of a data set. Here we will present the *arithmetic mean* (average). The standard deviation is a measure of the amount of variation observed in a data set.

We'll also look briefly at *quantiles*, which provide a different perspective from which to view the spread or variation in a data set.

The arithmetic mean

Usually, when speaking of the *average* of a set of data, without further qualification, we're speaking of the arithmetic mean. You probably have some intuitive sense of what this means. The mean summarizes the data, boiling it down to a single value that's somehow "in the middle."

Here's how we define and calculate the **arithmetic mean**, denoted μ, given some set of values, X.

$$\mu = \frac{1}{N} \sum_{i=0}^{N-1} x_i$$

where we have a set of numeric values, X, indexed by i, with N equal to the number of elements in X.

Here's an example: number of dentists per 10,000 population, by country, 2020.[1]

The first few records of this data set look like this:

Country	Value
Bangladesh	0.69
Belgium	11.33
Bhutan	0.97
Brazil	6.68
Brunei	2.38
Cameroon	0.049
Chad	0.011
Chile	14.81
Colombia	8.26
Costa Rica	10.58
Cyprus	8.58
…	…

Assume we have these data saved in a CSV file named `dentists_per_10k.csv`. We can use Python's `csv` module to read the data.

[1] Source: World Health Organization: https://www.who.int/data/gho/data/indicators/indicator-details/GHO/dentists-(per-10-000-population) (retrieved 2023-07-07)

```
data = []
with open('dentists_per_10k.csv', newline='') as fh:
    reader = csv.reader(fh)
    next(reader)  # skip the first row (column headings)
    for row in reader:
        data.append(float(row[1]))
```

We can write a function to calculate the mean. It's a simple one-liner.

```
def mean(lst):
    return sum(lst) / len(lst)
```

That's a complete implementation of the formula:

$$\mu = \frac{1}{N} \sum_{i=0}^{N-1} x_i$$

We take all the x_i and add them up (with `sum()`). Then we get the number of elements in the set (with `len()`) and use this as the divisor. If we print the result with `print(f"{mean(data):.4f}")` we get 5.1391.

That tells us a little about the data set: on average (for the sample of countries included) there are a little over five dentists per 10,000 population. If everyone were to go to the dentist once a year, that would suggest, on average, that each dentist serves a little less than 2,000 patients per year. With roughly 2,000 working hours in a year, that seems plausible. But the mean doesn't tell us much more than that.

To get a better understanding of the data, it's helpful to understand how values are distributed about the mean.

Let's say we didn't know anything about the distribution of values about the mean. It would be reasonable for us to assume these values are *normally distributed*. There's a function which describes the *normal distribution*, and you've likely seen the so-called "bell curve" before.

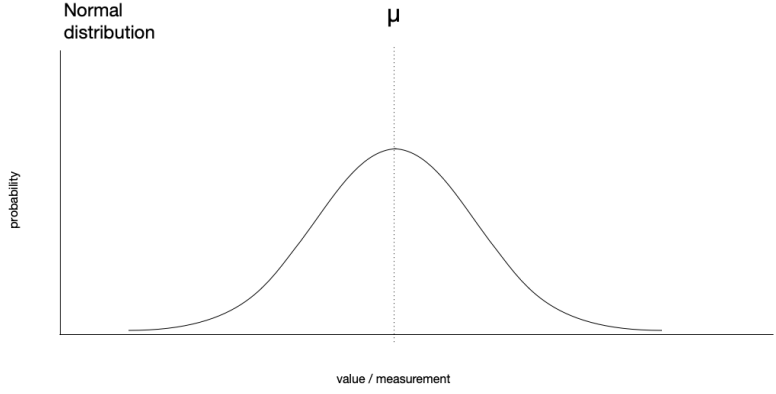


Figure 14.1

On the x-axis are the values we might measure, and on the y-axis we have the probability of observing a particular value. In a normal distribution, the mean is the most likely value for an observation, and the greater the distance from the mean, the less likely a given value. The *standard deviation* tells us how spread out values are about the mean. If the standard deviation is large, we have a broad curve:

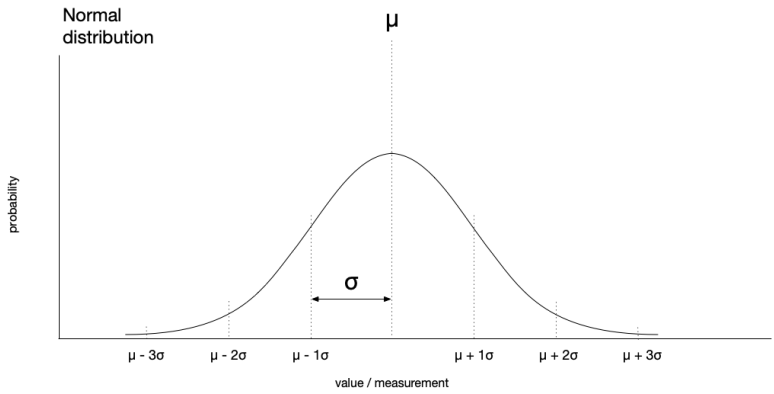


Figure 14.2

With a smaller standard deviation, we have a narrow curve with a higher peak.

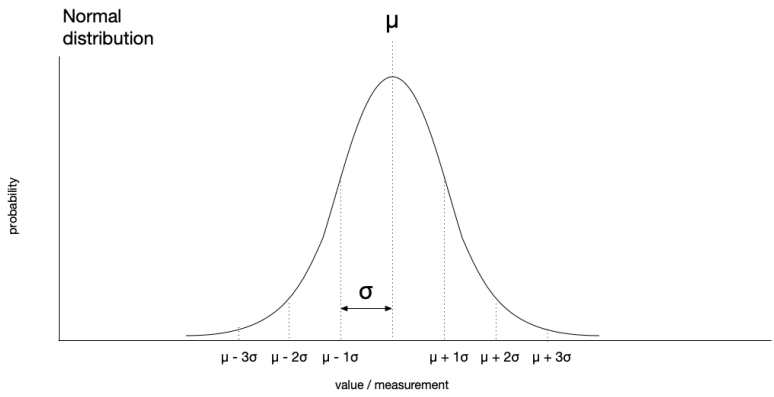


Figure 14.3

The area under these curves is equal.

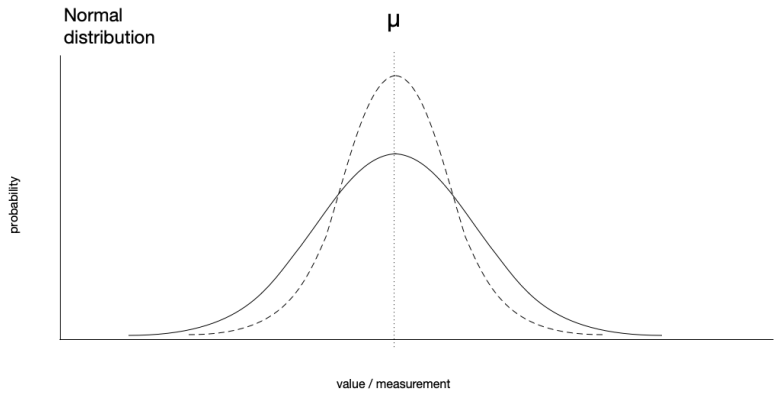


Figure 14.4

If we had a standard deviation of zero, that would mean that every value in the data set is identical (that doesn't happen often). The point is that the greater the standard deviation, the greater the variation there is in the data.

Just like we can calculate the mean of our data, we can also calculate the standard deviation. The **standard deviation**, written σ, is given by

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(x_i - \mu)^2}.$$

Let's unpack this. First, remember the goal of this measure is to tell us how much variation there is in the data. But variation with respect to what? Look at the expression $(x_i - \mu)^2$. We subtract the mean, μ, from each value in the data set x_i. This tells us how far the value of a given observation is from the mean. But we care more about the distance from the mean rather than whether a given value is above or below the mean. That's where the squaring comes in. When we square a positive number we get a positive number. When we square a negative number we get a positive number. So by squaring the difference between a given value and the mean, we're eliminating the sign. Then, we divide the sum of these squared differences by the number of elements in the data set (just as we do when calculating the mean). Finally, we take the square root of the result. Why do we do this? Because by squaring the differences, we stretch them, changing the scale. For example, $5 - 2 = 3$, but $5^2 - 2^2 = 25 - 4 = 21$. So this last step, taking the square root, returns the result to a scale appropriate to the data. This is how we calculate standard deviation.[2]

As we calculate the summation, we perform the calculation $(x_i - \mu)^2$. On this account, we cannot use `sum()`. Instead, we must calculate this in

[2]Strictly speaking, this is the *population* standard deviation, and is used if the data set represents the entire universe of possible observations, rather than just a sample. There's a slightly different formula for the *sample* standard deviation.

a loop. However, this is not too terribly complicated, and implementing this in Python is left as an exercise for the reader.

Assuming we have implemented this correctly, in a function named `std_dev()`, if we apply this to the data and print with `print(f"{std_dev(data):.4f}")`, we get 4.6569.

How do we interpret this? Again, the standard deviation tells us how spread out values are about the mean. Higher values mean that the data are more spread out. Lower values mean that the data are more closely distributed about the mean.

What practical use is the standard deviation? There are many uses, but it's commonly used to identify unusual or "unlikely" observations.

We can calculate the area of some portion under the normal curve. Using this fact, we know that given a normal distribution, we'd expect to find 68.26% of observations within one standard deviation of the mean. We'd expect 95.45% of observations within two standard deviations of the mean. Accordingly, the farther from the mean, the less likely an observation. If we express this distance in standard deviations, we can determine just how likely or unlikely an observation might be (assuming a normal distribution). For example, an observation that's more than five standard deviations from the mean would be very unlikely indeed.

Range	Expected fraction in range
$\mu \pm \sigma$	68.2689%
$\mu \pm 2\sigma$	95.4500%
$\mu \pm 3\sigma$	99.7300%
$\mu \pm 4\sigma$	99.9937%
$\mu \pm 5\sigma$	99.9999%

When we have real-world data, it's not often perfectly normally distributed. By comparing our data with what would be expected if it were normally distributed we can learn a great deal.

Returning to our dentists example, we can look for possible outliers by iterating through our data and finding any values that are greater than two standard deviations from the mean.

```
m = mean(data)
std = std_dev(data)

outliers = []
for datum in data:
    if abs(datum) > m + 2 * std:
        outliers.append(datum)
```

In doing so, we find two possible outliers—14.81, 16.95—which correspond to Chile and Uruguay, respectively. This might well lead us to ask, "Why are there so many dentists per 10,000 population in these particular countries?"

14.2 Python's statistics module

While implementing standard deviation (either for a sample or for an entire population) is straightforward in Python, we don't often write functions like this ourselves (except when learning how to write functions). Why? Because Python provides a statistics module for us.

We can use Python's `statistics` module just like we do with the `math` module. First we import the module, then we have access to all the functions (methods) within the module.

Let's start off using Python's functions for mean and population standard deviation. These are `statistics.mean()` and `statistics.pstdev()`, and they each take an iterable of numeric values as arguments.

```
import csv
import statistics

data = []
with open('dentists_per_10k.csv', newline='') as fh:
    reader = csv.reader(fh)
    next(reader)  # skip the first row
    for row in reader:
        data.append(float(row[1]))

print(f"{statistics.mean(data):.4f}")
print(f"{statistics.pstdev(data):.4f}")
```

When we run this, we see that the results for mean and standard deviation—5.1391 and 4.6569, respectively—are in perfect agreement with the results reported above.

The `statistics` module comes with a great many functions including:

- `mean()`
- `median()`
- `pstdev()`
- `stdev()`
- `quantiles()`

among others.

Using the statistics module to calculate quantiles

Quantiles divide a data set into continuous intervals, with each interval having equal probability. For example, if we divide our data set into quartiles ($n = 4$), then each quartile represents 1/4 of the distribution. If we divide our data set into quintiles ($n = 5$), then each quintile represents 1/5 of the distribution. If we divide our data into percentiles ($n = 100$), then each percentile represents 1/100 of the distribution.

You may have seen quantiles—specifically percentiles—before, since these are often reported for standardized test scores. If your score was in the 80th percentile, then you did better than 79% of others taking the

test. If your score was in the 95th percentile, then you're in the top 5% all those who took the test.

Let's use the `statistics` module to find quintiles for our dentists data (recall that quintiles divide the distribution into five parts).

If we import `csv` and `statistics` and then read our data (as above), we can calculate the values which divide the data into quintiles thus:

```
quintiles = statistics.quantiles(data, n=5)
print(quintiles)
```

Notice that we pass the data to the function just as we did with `mean()` and `pstdev()`. Here we also supply a keyword argument, `n=5`, to indicate we want quintiles. When we print the result, we get

```
[0.274, 2.2359999999999998, 6.590000000000001, 8.826]
```

Notice we have *four* values, which divide the data into *five* equal parts. Any value below 0.274 is in the first quintile. Values between 0.274 and 0.236 (rounding) are in the second quartile, and so on. Values above 8.826 are in the fifth quintile.

If we check the value for the United States of America (not shown in the table above), we find that the USA has 5.99 dentists per 10,000 population, which puts it squarely in the third quartile. Countries with more than 8.826 dentists per 10,000—those in the top fifth—are Belgium (11.33), Chile (14.81), Costa Rica (10.58), Israel (8.88), Lithuania (13.1), Norway (9.29), Paraguay (12.81), and Uruguay (16.95). Of course, interpreting these results is a complex matter, and results are no doubt influenced by *per capita* income, number and size of accredited dental schools, regulations for licensure and accreditation, and other infrastructure and economic factors.[3]

Other functions in the `statistics` module

I encourage you to experiment with these and other functions in the statistics module. If you have a course in which you're expected to calculate means, standard deviations, and the like, you might consider doing away with your spreadsheet and trying this in Python!

14.3 A brief introduction to plotting with Matplotlib

Now that we've seen how to read data from a file, and how to generate some descriptive statistics for the data, it makes sense that we should address visual presentation of data. For this we will use a third-party[4] module: Matplotlib.

[3] I'm happy with my dentist here in Vermont, but I will say she's booking appointments over nine months in advance, so maybe a few more dentists in the USA wouldn't be such a bad thing.

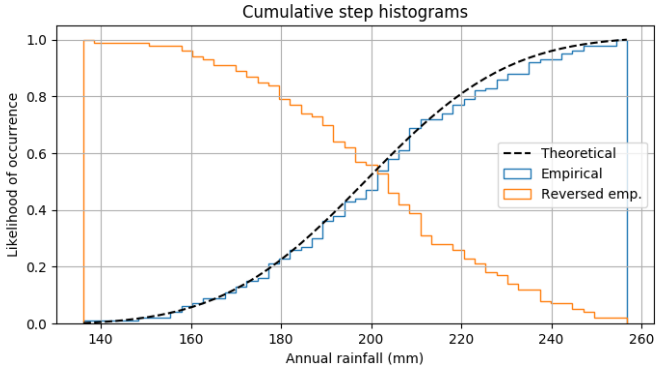
[4] *i.e*, not provided by Python or written by you

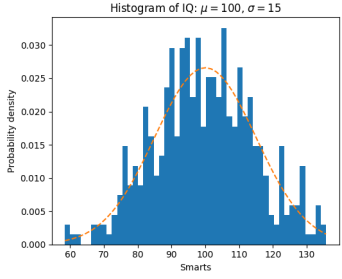
Matplotlib is a feature-rich module for producing a wide array of graphs, plots, charts, images, and animations. It is the *de facto* standard for visual presentation of data in Python (yes, there are some other tools, but they're not nearly as widely used).

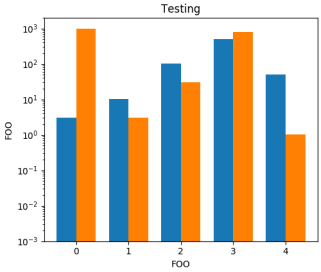
Since Matplotlib is not part of the Python core library (like the `math`, `csv`, and `statistics` modules we've seen so far), we need to install Matplotlib before we can use it.

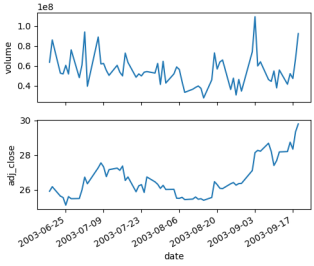
The installation process for third-party Python modules is unlike installing an app on your phone or desktop. Some IDEs (PyCharm, Thonny, VS Code, *etc.*) have built-in facilities for installing such modules. IDLE (the Python-supplied IDE) does not have such a facility. Accordingly, we won't get into the details of installation here (since details will vary from OS to OS, machine to machine), though if you're the DIY type and aren't using PyCharm, Thonny, or VS Code, you may find *Appendix C: `pip` and `venv`* helpful.

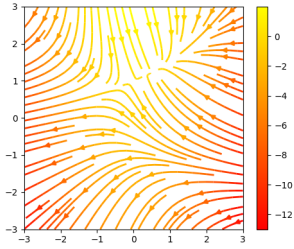
Here are some examples of plots made with Matplotlib (from the Matplotlib gallery at matplotlib.org):











For more examples and complete (very well-written) documentation, visit https://matplotlib.org.

14.4 The basics of Matplotlib

We're just going to cover the basics here. Why? Because Matplotlib has thousands of features *and* it has excellent documentation. So we're just going to dip a toe in the waters.

For more, see:

- Matplotlib website: https://matplotlib.org
- Getting started guide: https://matplotlib.org/stable/users/getting_started
- Documentation: https://matplotlib.org/stable/index.html
- Quick reference guides and handouts: https://matplotlib.org/cheatsheets

The most basic basics

We'll start with perhaps the simplest interface provided by Matplotlib, called `pyplot`. To use `pyplot` we usually import and abbreviate:

```
import matplotlib.pyplot as plt
```

Renaming isn't required, but it is commonplace (and this is how it's done in the Matplotlib documentation). We've seen this syntax before—using `as` to give a name to an object without using the assignment operator (`=`). It's very much like giving a name to a file object when using the `with` context manager. Here we give `matplotlib.pyplot` a shorter name `plt` so we can refer to it easily in our code. This is almost as if we'd written

```
import matplotlib.pyplot

plt = matplotlib.pyplot
```

Almost.

Now let's generate some data to plot. We'll generate random numbers in the interval $(-1.0, 1.0)$.

```
import random

data = [0]
for _ in range(100):
    data.append(data[-1]
                + random.random()
                * random.choice([-1, 1]))
```

So now we've got some random data to plot. Let's plot it.

```
plt.plot(data)
```

That's pretty straightforward, right?

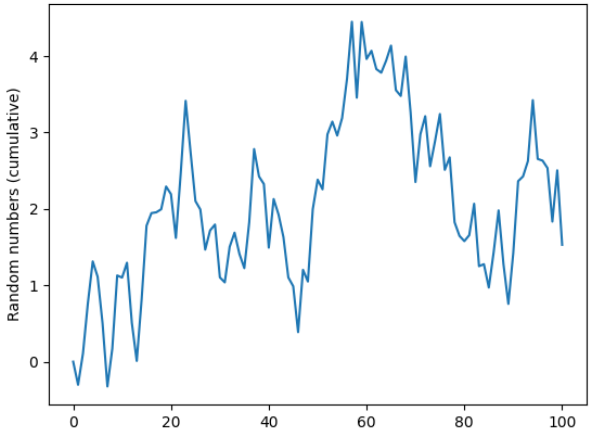
Now let's label our y axis.

```
plt.ylabel('Random numbers (cumulative)')
```

Let's put it all together and display our plot.

```
import random
import matplotlib.pyplot as plt

data = [0]
for _ in range(100):
    data.append(data[-1]
                + random.random()
                * random.choice([-1, 1]))
```

```
plt.plot(data)
plt.ylabel('Random numbers (cumulative)')
plt.show()
```



It takes only one more line to save our plot as an image file. We call the `savefig()` method and provide the file name we'd like to use for our plot. The plot will be saved in the current directory, with the name supplied.

```
import random
import matplotlib.pyplot as plt

data = [0]
for _ in range(100):
    data.append(data[-1]
                + random.random()
                * random.choice([-1, 1]))

plt.plot(data)
plt.ylabel('Random numbers (cumulative)')
plt.savefig('my_plot.png')
plt.show()
```

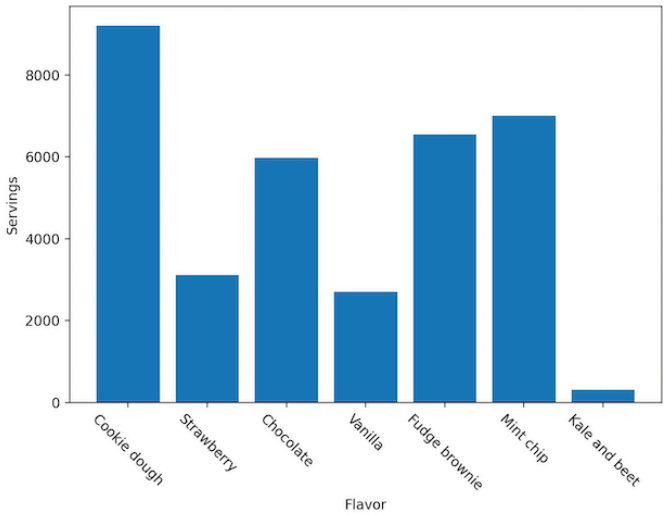
That's it. Our first plot—presented and saved to file.

Let's do another. How about a bar chart? For our bar chart, we'll use this as data (which is totally made up by the author):

Flavor	Servings
Cookie dough	9,214
Strawberry	3,115
Chocolate	5,982
Vanilla	2,707
Fudge brownie	6,553
Mint chip	7,005
Kale and beet	315

Let's assume we have this saved in a CSV file called `flavors.csv`. We'll read the data from the CSV file, and produce a simple bar chart.

```
import csv

import matplotlib.pyplot as plt

servings = []  # data
flavors = []   # labels

with open('flavors.csv') as fh:
    reader = csv.reader(fh)
    for row in reader:
        flavors.append(row[0])
        servings.append(int(row[1]))

plt.bar(flavors, servings)
plt.xticks(flavors, rotation=-45)
plt.ylabel("Servings")
plt.xlabel("Flavor")
plt.tight_layout()

plt.show()
```



Voilá! A bar plot!

Notice that we have two lists: one holding the servings data, the other holding the x-axis labels (the flavors). Instead of `plt.plot()` we use `plt.bar()` (makes sense, right?) and we supply flavors and servings as arguments. There's a little tweak we give to the x-axis labels, we rotate them by 45 degrees so they don't all mash into one another and become illegible. `plt.tight_layout()` is used to automatically adjust the padding around the plot itself, leaving suitable space for the bar labels and axis labels.

Be aware of how `plt.show()` behaves

When you call `plt.show()` to display your plot, Matplotlib creates a window and displays the plot in the window. At this point, your program's execution will pause until you close the plot window. When you close the plot window, program flow will resume.

Summary

Again, this isn't the place for a complete presentation of all the features of Matplotlib. The intent is to give you just enough to get started. Fortunately, the Matplotlib documentation is excellent, and I encourage you to look there first for examples and help.

- https://matplotlib.org

14.5 Exceptions

`StatisticsError`

The `statistics` module has its own type of exception, `StatisticsError`. You may encounter this if you try to find the mean, median, or mode of an empty list.

```
>>> statistics.mean([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/.../python3.10/statistics.py", line 328, in mean
    raise StatisticsError('mean requires at least one data
    point')
statistics.StatisticsError: mean requires at least one data
point
```

This is also raised if you specify less than one quantile.

```
>>> statistics.quantiles([3, 6, 9, 5, 1], n=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/.../python3.10/statistics.py", line 658,
    in quantiles
    raise StatisticsError('n must be at least 1')
statistics.StatisticsError: n must be at least 1
```

`StatisticsError` is actually a more specific type of `ValueError`.

Exceptions when using Matplotlib

There are many different exceptions that could be raised if you make programming errors while using Matplotlib. The exception and how to fix it will depend on context. If you encounter an exception from Matplotlib, your best bet is to consult the Matplotlib documentation.

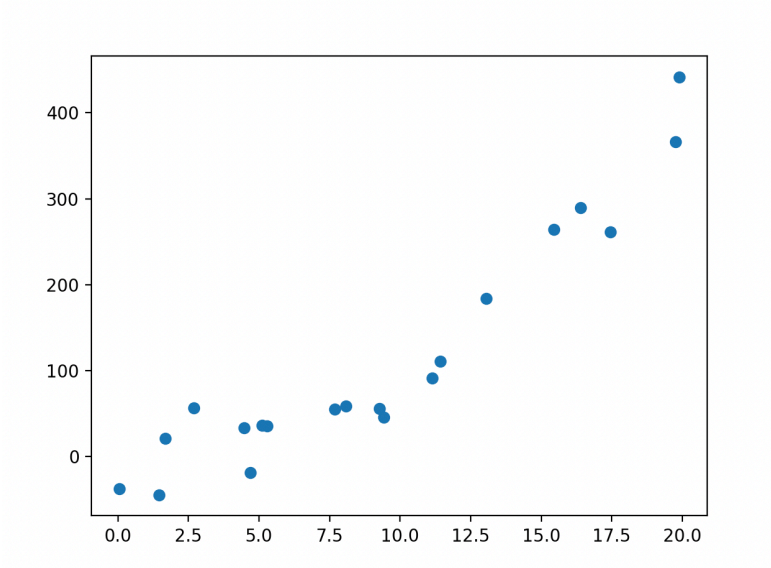
14.6 Exercises

Exercise 01

Try creating your own small data set (one dimension, five to ten elements) and plot it. Follow the examples given in this chapter. Plot your data as a line plot and as a bar plot.

Exercise 02

Matplotlib supports scatter plots too. In a scatter plot, each data point is a pair of values, x and y. Here's what a scatter plot looks like.



Create your own data (or find something suitable on the internet) and create your own scatter plot. x values should be in one list, corresponding y values should be in another list. Both lists should have the exact same number of elements. If these are called `xs` and `ys`, then you can create a scatter plot with

```
plt.scatter(xs, ys)
```

Make sure you display your plot, and save your plot as an image file.

Exercise 03

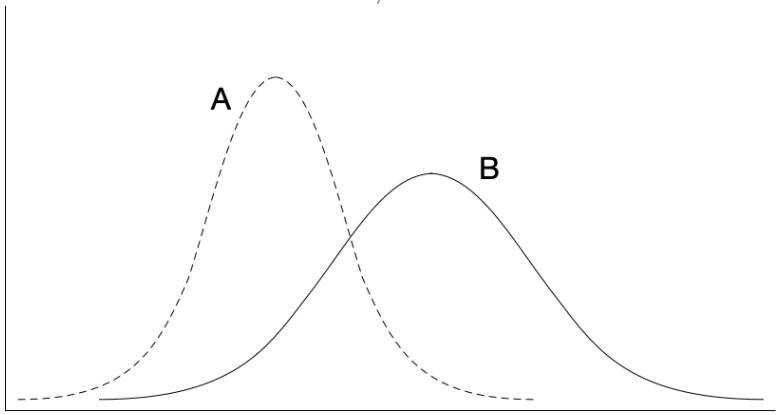
Edwina has calculated the mean and standard deviation of her data set—measurements of quill length in crested porcupines (species: *Hystrix cristata*). She has found that the mean is 31.2 cm and the standard deviation is 7.9 cm.

a. If one of the quills in her sample is 40.5 cm, should she consider this an unusually long quill? Why or why not?

b. What if there's a quill that's 51.2 cm? Should this be considered unusually long? Why or why not?

Exercise 04

Consider the two distributions shown, A and B.



a. Which of these has the greater mean?

b. Which of these has the greater standard deviation?

c. Which of these curves have the greater area under them? (hint: this is a trick question)

Exercise 05

The geometric mean is another kind of mean used in statistics and finance. Rather than summing all the values in the data and then dividing by the number of values, we take the *product* of all the values, and then, if there are N values, we take the Nth root of the result. Typically, this is only used when all values in the data set are positive.

We define the geometric mean:

$$\gamma = \left(\prod_{i=0}^{N-1} x_i \right)^{\frac{1}{N}}$$

where the $\prod$ symbol signifies repeated multiplication. Just as $\sum$ says "add them all up", $\prod$ means "multiply them all together."

We can calculate the Nth root of a number using exponentiation by a fraction as shown. For example, to calculate the cube root of some x, we can use `x ** (1 / 3)`.

Implement a function which calculates the geometric mean of an arbitrary list of positive numeric values. You can verify your result by comparing the output of your function with the output of the `statistics` module's `geometric_mean()` function.

Chapter 15

Exception handling

We've seen a lot of exceptions so far:

- `SyntaxError`
- `IndentationError`
- `AttributeError`
- `NameError`
- `UnboundLocalError`
- `TypeError`
- `IndexError`
- `ValueError`
- `ZeroDivisionError`
- `FileNotFoundError`
- `UnicodeDecodeError`
- `JSONDecodeError`

These are exceptions defined by Python, and which are *raised* when certain errors occur. (There are many, *many* other exceptions that are outside the scope of this textbook.)

When an *unhandled* exception occurs, your program terminates. This is usually an undesired outcome.

Here we will see that some of these exceptions can be handled gracefully using `try` and `except`—these go together, hand in hand. In a `try` block, we include the code that we think *might* raise an exception. In the following `except` block, we *catch* or *handle* certain exceptions. What we do in the `except` block will depend on the desired behavior for your program.

However, we'll also see that some of these—`SyntaxError` and `IndentationError` for example—can't be handled, since these occur when Python is first reading your code, prior to execution.

We'll also see that some of these exceptions only occur when there's a defect in our code that we do not want to handle! These are exceptions like `AttributeError` and `NameError`. Trying to handle these covers up defects in our code that we should repair. Accordingly, there aren't many cases where we'd even want to handle these exceptions.

Sometimes we want to handle `TypeError` or `IndexError`. It's very often the case that we want to handle `ValueError` or `ZeroDivisionError`. It's

almost always the case that we want to handle `FileNotFoundError`. Much of this depends on context, and there's a little art in determining which exceptions are handled, and how they should be handled.

Learning objectives

- You will understand why many of the Python exceptions are raised.
- You will learn how to deal with exceptions when they are raised, and how to handle them gracefully.
- You will learn that sometimes it's not always best to handle every exception that could be raised.

Terms introduced

- exception handling
- "it's easier to ask forgiveness than it is to ask for permission" (EAFP)
- "look before you leap" (LBYL)
- raise (an exception)
- try/except

15.1 Exceptions

By this time, you've seen quite a few exceptions. Exceptions occur when something goes wrong. We refer to this as *raising* an exception.

Exceptions may be raised by the Python interpreter or by built-in functions or by methods provided by Python modules. You may even raise exceptions in your own code (but we'll get to that later).

Exceptions include information about the type of exception which has been raised, and where in the code the exception occurred. Sometimes, quite a bit of information is provided by the exception. In general, a good approach is to start at the last few lines of the exception message, and work backward if necessary to see what went wrong.

There are many types of built-in exceptions in Python. Here are a few that you're likely to have seen before.

`SyntaxError`

When a module is executed or imported, Python will read the file, and try *parsing* the file. If, during this process, the parser encounters a syntax error, a `SyntaxError` exception is raised. `SyntaxError` can also be raised when invalid syntax is used in the Python shell.

```
>>> # if requires a condition and colon
>>> if
  File "<stdin>", line 1
    if
      ^
SyntaxError: invalid syntax
```

Here you see the exception includes information about the error and where the error occurred. The `^` is used to point to a portion of code where the error occurred.

`IndentationError`

`IndentationError` is a subtype of `SyntaxError`. Recall that indentation is significant in Python—we use it to structure branches, loops, and functions. So `IndentationError` occurs when Python encounters a syntax error that it attributes to invalid indentation.

```
if True:
x = 1    # This should be indented!
  File "<stdin>", line 2
    x = 1  # should be indented
    ^
IndentationError: expected an indented block
    after 'if' statement on line 1
```

Again, almost everything you need to know is included in the last few lines of the message. Here Python is informing us that it was expecting an indented block of code immediately following an `if` statement.

`AttributeError`

There are several ways an `AttributeError` can be raised. You may have encountered an `AttributeError` by misspelling the name of a method in a module you've imported.

```
>>> import math
>>> math.foo   # There is no such method or constant in math
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'math' has no attribute 'foo'
```

An `AttributeError` is different from a `SyntaxError` in that it occurs at runtime, and not during the parsing or early processing of code. An `AttributeError` is only related to the availability of *attributes* and not violations of the Python syntax rules.

`NameError`

A `NameError` is raised when Python cannot find an identifier. For example, if you were to try to perform a calculation with some variable `x` without previously having assigned a value to `x`.

```
>>> x + 1   # Without having previously assigned a value to x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Any attempted access of an undefined variable will result in a `NameError`.

```
>>> foo   # Wasn't defined earlier
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
```

`UnboundLocalError`

An *unbound local error* is a more specific kind of `NameError`—one which occurs within a function or method. This type of error is raised when we try to read a value from a variable before the variable has been assigned a value within the body (scope) of a function.

Example:

```
>>> def f(x):
    y = y + 1
    return x + y

>>>
>>> f(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'y' referenced before assignment
```

`IndexError`

Individual elements of a sequence can be accessed using an index into the sequence. This presumes, of course, that the index is valid—that is, there is an element at that index. If you try to access an element of a sequence by its index, and there is no index, Python will raise an `IndexError`.

```
>>> lst = []
>>> lst[2]   # There is no element at index 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

This (above) fails because the list is empty and there is no element at index 2. Hence, 2 is an invalid index and an `IndexError` is raised.

Here's another example:

```
>>> alphabet = 'abcdefghijklmnopqrstuvwxyz'
>>> alphabet[26]  # Python is 0-indexed so z has index 25
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
IndexError: string index out of range
```

TypeError

A `TypeError` is raised when a value of one type is expected and a different type is supplied. For example, sequence indices—for lists, tuples, strings—must be integers. If we try using a `float` or `str` as an index, Python will raise a `TypeError`.

```
>>> lst = [6, 5, 4, 3, 2]
>>> lst['foo']   # Can't use a string as an index
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not str
```

```
>>> lst[1.0]   # Can't use a float as an index
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
```

Python will also raise a `TypeError` if we try to perform operations on operands which are not supported. For example, we cannot concatenate an `int` to a `str` or add a `str` to an `int`.

```
>>> 'foo' + 1  # Try concatenating 1 with 'foo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

```
>>> 1 + 'foo'  # Try adding 'foo' to 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

We cannot calculate the sum of a number and a string. We cannot calculate the sum of any list or tuple which contains a string.

```
>>> sum('Add me up!')  # Can't sum a string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
>>> sum(1)  # Python sum() requires an iterable of numerics
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

By the same token, we cannot get the length of a `float` or `int`.

```
>>> len(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

`ValueError`

A `ValueError` is raised when the *type* of some argument or operand is correct, but the *value* is not. For example, `math.sqrt(x)` will raise a `ValueError` if we try to take the square root of a negative number.

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Note that dividing by zero is considered an arithmetic error, and has its own exception (see below).

`ZeroDivisionError`

Just as in mathematics, Python will not allow us to divide by zero. If we try to, Python will raise a `ZeroDivisionError`. Note that this occurs with floor division and modulus as well (as they depend on division).

```
>>> 10 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> 10 % 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

```
>>> 10 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

`FileNotFoundError`

We've seen how to open files for reading and writing. There are many ways this can go wrong, but one common issue is `FileNotFoundError`. This exception is raised when Python cannot find the specified file. The

file may not exist, may be in the wrong directory, or may be named incorrectly.

```
open('non-existent file')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
    'non-existent file'
```

`UnicodeDecodeError`

A `UnicodeDecodeError` occurs if we try to decode a string that was encoded in a different system. This can occur when reading data from a file or other source. Example (assuming `data.txt` was encoded with, say, Windows-1252 encoding):

```
with open("data.txt") as fh:
    s = fh.read()
```

will result in a `UnicodeDecodeError` if there are any invalid UTF-8 encodings in the input. To fix, specify the correct encoding:

```
with open("data.txt", encoding=cp1252) as fh:
    s = fh.read()
```

`JSONDecodeError`

A `JSONDecodeError` occurs when we try to use `json.loads()` or `json.load()` and the input is not a valid JSON encoding. Example:

```
>>> import json
>>> json.loads("I am not a valid JSON-encoded string!")
Traceback (most recent call last):
  File "<python-input-33>", line 1, in <module>
  ...
json.decoder.JSONDecodeError:
     Expecting value: line 1 column 1 (char 0)
```

15.2 Handling exceptions

So far, what we've seen is that when an exception is raised our program is terminated (or not even run to begin with in the case of a `SyntaxError`). However, Python allows us to handle exceptions. What this means is that when an exception is raised, a specific block of code can be executed to deal with the problem.

For this we have, minimally, a try/except compound statement. This involves creating two blocks of code: a `try` block and an exception handler—an `except` block.

The code in the `try` block is code where we want to guard against unhandled exceptions. A `try` block is followed by an `except` block. The `except` block specifies the type of exception we wish to handle, and code for handling the exception.

Input validation with try/except

Here's an example of input validation using try/except. Let's say we want a positive integer as input. We've seen how to validate input in a `while` loop.

```
while True:
    n = int(input("Please enter a positive integer: "))
    if n > 0:
        break
```

This ensures that if the user enters an integer that's less than one, that they'll be prompted again until they supply a positive integer. But what happens if the naughty user enters something that cannot be converted to an integer?

```
Please enter a positive integer: cheese
Traceback (most recent call last):
  File "/.../code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'cheese'
```

Python cannot convert `'cheese'` to an integer and thus a `ValueError` is raised.

So now what? We put the code that could result in a `ValueError` in a `try` block, and then provide an exception handler in an `except` block. Here's how we'd do it.

```
while True:
    try:
        user_input = input("Enter a positive integer: ")
        n = int(user_input)
        if n > 0:
            break
    except ValueError:
        print(f'"{user_input}" cannot be converted to an int!')

print(f'You have entered {n}, a positive integer.')
```

Let's run this code, and try a little mischief:

```
Enter a positive integer: negative
"negative" cannot be converted to an int!
Enter a positive integer: cheese
"cheese" cannot be converted to an int!
Enter a positive integer: -42
Enter a positive integer: 15
You have entered 15, a positive integer.
```

See? Now mischief (or failure to read instructions) is handled gracefully.

Getting an index with try/except

Earlier, we saw that `.index()` will raise a `ValueError` exception if the argument passed to the `.index()` method is not found in the underlying sequence.

```
>>> lst = ['apple', 'boat', 'cat', 'drama']
>>> lst.index('egg')
Traceback (most recent call last):
  File "/.../code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
ValueError: 'egg' is not in list
```

We can use exception handling to improve on this.

```
lst = ['apple', 'boat', 'cat', 'drama']
s = input('Enter a string to search for: ')
try:
    i = lst.index(s)
    print(f'The index of "{s}" in {lst} is {i}.')
except ValueError:
    print(f'"{s}" was not found in {lst}.')
```

If we were to enter "egg" at the prompt, this code would print:

```
"egg" was not found in ['apple', 'boat', 'cat', 'drama']
```

This brings up the age-old question of whether it's better to check first to see if you can complete an operation without error, or better to try and then handle an exception if it occurs. Sometimes these two approaches are referred to as "look before you leap" (LBYL) and "it's easier to ask forgiveness than it is to ask for permission" (EAFP). Python favors the latter approach.

Why is this the case? Usually, EAFP makes your code more readable, and there's no guarantee that the programmer can anticipate and write all the necessary checks to ensure an operation will be successful.

In this example, it's a bit of a toss up. We could write:

```
if s in lst:
    print(f'The index of "{s}" in {lst} is {lst.index(s)}.')
else:
    print(f'"{s}" was not found in {lst}.')
```

Or we could write (as we did earlier):

```
try:
    print(f'The index of "{s}" in {lst} is {lst.index(s)}.')
except ValueError:
    print(f'"{s}" was not found in {lst}.')
```

Dos and don'ts

Do:

- Keep `try` blocks as small as possible.
- Catch and handle specific exceptions.
- Avoid catching and handling `IndexError`, `TypeError`, `NameError`. When these occur, it's almost always due to a defect in programming. Catching and handling these exceptions can hide defects that should be corrected.
- Use separate `except` blocks to handle different kinds of exceptions.

Don't:

- Write one handler for different exception types.
- Wrap all your code in one big `try` block.
- Use exception handling to hide programming errors.
- Use bare `except:` or `except Exception:`—these are too general and might catch things you shouldn't.

15.3 Exceptions and flow of control

While it's considered *pythonic* to use exceptions and to follow the rule of EAFP ("easier to ask for forgiveness than permission"), it is unwise to use exceptions for controlling the flow of program execution except within very narrow limits.

Here are some rules to follow:

- Keep `try` and `except` blocks as small as possible.
- Handle an exception in the most simple and direct way possible.
- Avoid calling another function from within an `except` block which might send program flow away from the point where the exception was raised.

15.4 Exercises

Exercise 01

> **! Important**
>
> Be sure to save your work for this exercise, as we will revisit these in later exercises!

Here's an example of code which raises a `SyntaxError`:

```
>>> foo bar
  File "<stdin>", line 1
    foo bar
        ^^^
SyntaxError: invalid syntax
```

Here's an example of code which raises a `TypeError`:

```
>>> 1 + []
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Without using `raise`, write your own code that results in the following exceptions:

a. `SyntaxError`
b. `IndentationError`
c. `IndexError`
d. `NameError`
e. `TypeError`
f. `AttributeError`
g. `ZeroDivisionError`
h. `FileNotFoundError`

Exercise 02

Now write a try/except for the following exceptions, starting with code you wrote for exercise 01.

a. `ValueError`

b. `ZeroDivisionError`

c. `FileNotFoundError`

Exercise 03

`SyntaxError` and `IndentationError` should always be fixed in your code. Under normal circumstances, these *can't* be handled. `NameError` and

`AttributeError` almost always arise from programming defects. There's almost never any reason to write try/except for these.

Fix the code you wrote in exercise 01, for the following:

a. `SyntaxError`

b. `IndentationError`

c. `AttributeError`

d. `NameError`

Exercise 04

Usually, (but not always) `IndexError` and `TypeError` are due to programming defects. Take a look at the code you wrote to cause these errors in exercise 01. Does what you wrote constitute a programming defect? If so, fix it.

If you believe the code you wrote constitutes a legitimate case for try/except, write try/except for each of these.

Chapter 16

Dictionaries, sets, and structured data

So far, the data structures we've seen are either flat, one-dimensional objects—lists, tuples—or perhaps lists containing lists, lists containing tuples, and so on. However, in many instances it makes sense to structure our data in different ways. In this chapter, we'll learn about dictionaries, sets, named tuples, and a structured data format known as JSON (pronounced "Jay-son").

Giving structure to our data helps us organize things into logical components. This makes it easier and safer to find, read, write, and update data with our programs.

There are other benefits as well: with dictionaries, we can refer to elements by name (called a *key*), or in the case of named tuples, we can refer to elements by their *field name*. In either case, the concept is similar.

There are many programming problems that are best solved with structured data. Structured data is a first step toward understanding *classes* and *encapsulation*. These are fundamental concepts in what is called object-oriented programming (OOP), a commonly-used programming paradigm with many applications.

Dictionaries are ubiquitous, no doubt due to their usefulness and flexibility. Dictionaries store information in key/value pairs—we look up a value in a dictionary by its key. In this chapter we'll learn about dictionaries: how to create them, modify them, iterate over them and so on.

Sets are similar but not identical to sets you might have seen in a mathematics course. Like mathematical sets, they are unordered and do not contain duplicate elements. Unlike mathematical sets, Python sets must, of necessity, be finite.

Named tuples are like tuples, but we can refer to the elements of a named tuple by *name* or by index—either works. Like tuples, named tuples are immutable. This can make our code more readable and easier to reason about.

We'll also see JSON. JSON is an acronym for "JavaScript object notation." JSON is a widely used format for data interchange and many APIs

(application program interfaces) for web services return data in JSON format. Don't be fooled by the name. Though originally conceived as a way of notating objects in JavaScript, JSON escaped the lab so to speak, and JavaScript isn't required to work with JSON data. Indeed, Python provides its own JSON module that includes tools for reading and writing JSON data, and converting to native Python datastructures.

Learning objectives

- You will learn how to create a dictionary, set, or named tuple.
- You will understand that dictionaries and sets are *mutable*, meaning that their contents may change.
- You will learn how to access individual values in a dictionary by keys and within named tuples by field name.
- You will learn how to iterate over dictionaries.
- You will understand that dictionary keys must be *hashable*.
- You will learn how to read, write, and work with structured data using Python's JSON module.

Terms and Python keywords introduced

- class
- `del`
- dictionary
- field
- field name
- hashable
- JSON
- `JSONDecodeError`
- key
- `KeyError`
- named tuple
- set
- value
- view objects (keys, values, items)

16.1 Introduction to dictionaries and structured data

So far, we've seen tuples, lists, and strings as ordered collections of objects. We've also seen how to access individual elements within a list, tuple, or string by the element's index.

```
>>> lst = ['rossi', 'agostini', 'marquez', 'doohan', 'lorenzo']
>>> lst[0]
'rossi'
>>> lst[2]
'marquez'
```

```
>>> t = ('hearts', 'clubs', 'diamonds', 'spades')
>>> t[1]
'clubs'
```

This is all well and good, but sometimes we'd like to be able to use something other than a numeric index to access elements.

Consider conventional dictionaries which we use for looking up the meaning of words. Imagine if such a dictionary used numeric indices to look up words. Let's say we wanted to look up the word "pianist." How would we know its index? We'd have to hunt through the dictionary to find it. Even if all the words were in lexicographic order, it would still be a nuisance having to find a word this way.

The good news is that dictionaries don't work that way. We can look up the meaning of the word by finding the word itself. This is the basic idea of dictionaries in Python.

A Python dictionary, simply put, is a data structure which associates *keys* and *values*. In the case of a conventional dictionary, each word is a *key*, and the associated definition or definitions are the *values*.

Here's how the entry for "pianist" appears in my dictionary:[1]

> *pianist n.* a person who plays the piano, esp. a skilled or professional performer

Here *pianist* is the key, and the rest is the value. We can write this, with some liberty, as a Python dictionary, thus:

```
>>> d = {'pianist': "a person who plays the piano, " \
...      "esp. a skilled or professional performer"}
```

The entries of a dictionary appear within braces `{}`. The key/value pairs are separated by a colon, thus: `<key>: <value>`, where `<key>` is a valid key, and `<value>` is a valid value.

We can look up values in a dictionary by their key. The syntax is similar to accessing elements in a list or tuple by their indices.

```
>>> d['pianist']
'a person who plays the piano, esp. a skilled or
 professional performer'
```

Like lists, dictionaries are *mutable*. Let's add a few more words to our dictionary. To add a new entry to a dictionary, we can use this approach:

[1] Webster's New World Dictionary of the American Language, Second College Edition.

```
>>> d['cicada'] = "any of a family of large flylike " \
...               "insects with transparent wings"
>>> d['proclivity'] = "a natural or habitual tendency or " \
...                   "inclination, esp. toward something " \
...                   "discreditable"
>>> d['tern'] = "any of several sea birds, related to the " \
...             "gulls, but smaller, with a more slender " \
...             "body and beak, and a deeply forked tail"
>>> d['firewood'] = "wood used as fuel"
>>> d['holophytic'] = "obtaining nutrition by photosynthesis, " \
...                   "as do green plants and some bacteria"
```

Now let's inspect our dictionary.

```
>>> d
{'pianist': 'a person who plays the piano, esp. a skilled or
professional performer', 'cicada': 'any of a family of large
flylike insects with transparent wings', 'proclivity': 'a
natural or habitual tendency or inclination, esp. toward
something discreditable', 'tern': 'any of several sea birds,
related to the gulls, but smaller, with a more slender body
and beak, and a deeply forked tail', 'firewood': 'wood used
as fuel', 'holophytic': 'obtaining nutrition by photosynthesis,
as do green plants and some bacteria'}
```

We see that our dictionary consists of key/value pairs.

key	value
`'pianist'`	`'a person who plays the piano, esp. a skilled or professional performer'`
`'cicada'`	`'any of a family of large flylike insects with transparent wings'`
`'proclivity'`	`'a natural or habitual tendency or inclination, esp. toward something discreditable'`
`'tern'`	`'any of several sea birds, related to the gulls, but smaller, with a more slender body and beak, and a deeply forked tail'`
`'firewood'`	`'wood used as fuel'`
`'holophytic'`	`'obtaining nutrition by photosynthesis, as do green plants and some bacteria'`

We can look up any value with its key.

```
>>> d['tern']
'any of several sea birds, related to the gulls, but smaller,
with a more slender body and beak, and a deeply forked tail'
```

If we try to access a key which does not exist, this results in a `KeyError`.

```
>>> d['bungalow']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'bungalow'
```

This is a new type of exception we haven't seen until now.

We may overwrite a key with a new value.

```
>>> d = {'France': 'Paris',
...      'Mali': 'Bamako',
...      'Argentina': 'Buenos Aires',
...      'Thailand': 'Bangkok',
...      'Australia': 'Sydney'}  # oops!
>>> d['Australia'] = 'Canberra'  # fixed!
```

So far, in the examples above, keys and values have been strings. But this needn't be the case.

There are constraints on the kinds of things we can use as keys, but almost anything can be used as a value.

Here are some examples of valid keys:

```
>>> d = {(1, 2): 'My key is the tuple (1, 2)',
         100: 'My key is the integer 100',
         'football': 'My key is the string "football"'}
```

Values can be almost anything—even other dictionaries!

```
>>> students = {'eporcupi': {'name': 'Egbert Porcupine',
...                          'major': 'computer science',
...                          'gpa': 3.14},
...             'epickle':  {'name': 'Edwina Pickle',
...                          'major': 'biomedical engineering',
...                          'gpa': 3.71},
...             'aftoure':  {'name': 'Ali Farka Touré',
...                          'major': 'music',
...                          'gpa': 4.00}}
```

```
>>> students['aftoure']['major']
'music'
```

```
>>> recipes = {'bolognese': ['beef', 'onion', 'sweet pepper',
...                          'celery', 'parsley', 'white wine',
...                          'olive oil', 'garlic', 'milk',
...                          'black pepper', 'basil', 'salt'],
...            'french toast': ['baguette', 'egg', 'milk',
...                             'butter', 'cinnamon',
...                             'maple syrup'],
```

```
...            'fritters': ['potatoes', 'red onion', 'carrot',
...                         'red onion', 'garlic', 'flour',
...                         'paprika', 'marjoram', 'salt',
...                         'black pepper', 'canola oil']}
```

```
>>> recipes['french toast']
['baguette', 'egg', 'milk', 'butter', 'cinnamon', 'maple syrup']
>>> recipes['french toast'][-1]
'maple syrup'
```

```
>>> coordinates = {'Northampton': (42.5364, -70.9857),
...                'Kokomo': (40.4812, -86.1418),
...                'Boca Raton': (26.3760, -80.1223),
...                'Sausalito': (37.8658, -122.4980),
...                'Amarillo': (35.1991, -101.8452),
...                'Fargo': (46.8771, -96.7898)}
```

```
>>> lat, lon = coordinates['Fargo']  # tuple unpacking
>>> lat
46.8771
>>> lon
-96.7898
```

Restrictions on keys

Keys in a dictionary must be *hashable*. In order for an object to be hashable, it must be immutable, or if it is an immutable container of other objects (for example, a tuple) then all the objects contained must also be immutable. Valid keys include objects of type `int`, `float` (OK, but a little strange), `str`, `bool` (also OK, but use cases are limited). Tuples can also serve as keys as long as they do not contain any mutable objects.

```
>>> d = {0: 'Alexei Fyodorovich', 1: 'Dmitri Fyodorovich',
...      2: 'Ivan Fyodorovich', 3: 'Fyodor Pavlovich',
...      4: 'Agrafena Alexandrovna', 5: 'Pavel Fyodorovich',
...      6: 'Zosima', 7: 'Katerina Ivanovna'}
```

```
>>> d = {True: 'if porcupines are blue, then the sky is pink',
...      False: 'chunky monkey is the best ice cream'}
```

```
>>> d = {'Phelps': 23, 'Latynina': 9, 'Nurmi': 9, 'Spitz': 9,
...      'Lewis': 9, 'Bjørgen': 8}
```

However, these are not permitted:

```
>>> d = {['hello']: 'goodbye'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

```
>>> d = {(0, [1]): 'foo'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Testing membership

Just as with lists we can use the keyword `in` to test whether a particular key is in a dictionary.

```
>>> d = {'jane': 'maine', 'wanda': 'new hampshire',
...      'willard': 'vermont', 'simone': 'connecticut'}
>>> 'wanda' in d
True
>>> 'dobbie' in d
False
>>> 'dobbie' not in d
True
```

16.2 Iterating over dictionaries

If we iterate over a dictionary as we do with a list, this yields the dictionary's *keys*.

```
>>> furniture = {'living room': ['armchair', 'sofa', 'table'],
...              'bedroom': ['bed', 'nightstand', 'dresser'],
...              'office': ['desk', 'chair', 'cabinet']}
...
>>> for x in furniture:
...     print(x)
...
living room
bedroom
office
```

Usually, when iterating over a dictionary we use dictionary *view objects*. These objects provide a dynamic view into a dictionary's keys, values, or entries.

Dictionaries have three different view objects: items, keys, and values. Dictionaries have methods that return these view objects:

- `dict.keys()` which provides a view of a dictionary's keys,
- `dict.values()` which provides a view of a dictionary's values, and
- `dict.items()` which provides tuples of key/value pairs.

Dictionary view objects are all iterable.

Iterating over the keys of a dictionary

```
>>> furniture = {'living room': ['armchair', 'sofa', 'table'],
...              'bedroom': ['bed', 'nightstand', 'dresser'],
...              'office': ['desk', 'chair', 'cabinet']}
>>> for key in furniture.keys():
...     print(key)
...
living room
bedroom
office
```

Note that it's common to exclude `.keys()` if it's keys you want, since the default behavior is to iterate over keys (as shown in the previous example).

Iterating over the values of a dictionary

```
>>> for value in furniture.values():
...     print(value)
...
['armchair', 'sofa', 'table']
['bed', 'nightstand', 'dresser']
['desk', 'chair', 'cabinet']
```

Iterating over the items of a dictionary

```
>>> for item in furniture.items():
...     print(item)
...
('living room', ['armchair', 'sofa', 'table'])
('bedroom', ['bed', 'nightstand', 'dresser'])
('office', ['desk', 'chair', 'cabinet'])
```

Iterating over the items of a dictionary using tuple unpacking

```
>>> for key, value in furniture.items():
...     print(f"Key: '{key}', value: {value}")
...
Key: 'living room', value: ['armchair', 'sofa', 'table']
Key: 'bedroom', value: ['bed', 'nightstand', 'dresser']
Key: 'office', value: ['desk', 'chair', 'cabinet']
```

Some examples

Let's say we wanted to count the number of pieces of furniture in our dwelling.

```
>>> count = 0
>>> for lst in furniture.values():
...     count = count + len(lst)
...
>>> count
9
```

Let's say we wanted to find all the students in the class who are not CS majors, assuming the items in our dictionary look like this:

```
>>> students =
...    {'esmerelda' : {'class': 2024, 'major': 'ENSC', 'gpa': 3.08},
...     'winston': {'class': 2023, 'major': 'CS', 'gpa': 3.30},
...     'clark': {'class': 2022, 'major': 'PHYS', 'gpa': 2.95},
...     'kumiko': {'class': 2023, 'major': 'CS', 'gpa': 3.29},
...     'abeba' : {'class': 2024, 'major': 'MATH', 'gpa': 3.71}}
```

One approach:

```
>>> non_cs_majors = []
>>> for student, info in students.items():
...    if info['major'] != 'CS':
...        non_cs_majors.append(student)
...
>>> non_cs_majors
['esmerelda', 'clark', 'abeba']
```

16.3 Deleting dictionary keys

Earlier we saw that we could use list's `.pop()` method to remove an element from a list, removing either the last element in the list (the default, when no argument is supplied), or at a specific index (if we supply an argument).

Dictionaries are *mutable*, and thus, like lists, they can be changed. Dictionaries also support `.pop()` but it works a little differently than it does with lists. The `.pop()` method for dictionaries *requires* a valid key as an argument. This is because dictionaries don't have the same sense of linear order as a list—everything is based on *keys*.

So this works:

```
>>> d = {'foo': 'bar'}
>>> d.pop('foo')
'bar'
```

but this does not:

```
>>> d = {'foo': 'bar'}
>>> d.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pop expected at least 1 argument, got 0
```

Python also provides the keyword `del` which can be used to remove a key from a dictionary.

```
>>> pets = {'fluffy': 'gerbil', 'george': 'turtle',
...         'oswald': 'goldfish', 'wyatt': 'ferret'}
>>> del pets['oswald']   # RIP oswald :(
>>> pets
{'fluffy': 'gerbil', 'george': 'turtle', 'wyatt': 'ferret'}
```

But be careful! If you do not specify a key, the entire dictionary will be deleted!

```
>>> pets = {'fluffy': 'gerbil', 'george' : 'turtle',
...         'oswald': 'goldfish', 'wyatt': 'ferret'}
>>> del pets  # oops!
>>> pets
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pets' is not defined
```

Notice also that `.pop()` with a key supplied will return the value associated with that key and then remove the key/value pair. `del` will simply delete the entry.

16.4 Hashables

The keys of a dictionary cannot be arbitrary Python objects. In order to serve as a key, an object must be *hashable*.

Without delving into too much technical detail, the reason is fairly straightforward. We can't have keys that might change!

Imagine if that dictionary or thesaurus on your desk had magical keys that could change. You'd never be able to find anything. Accordingly, all keys in a dictionary must be hashable—and not subject to possible change.

Hashing is a process whereby we calculate a number (called a hash) from an object. In order to serve as a dictionary key, this hash value must never change.

What kinds of objects are hashable? Actually most of the objects we've seen so far are hashable.

Anything that is immutable and is not a container is hashable. This includes `int`, `float`, `bool`, `str`. It even includes objects of type `range` (though it would be very peculiar indeed if someone were to use a range as a dictionary key).

What about things that are immutable and are containers? Here we're speaking of tuples. If all the elements of a tuple are themselves immutable, then the tuple is hashable. If a tuple contains a mutable object, say, a list, then it is not hashable.

We can inspect the hash values of various objects using the built-in function, `hash()`.

```
>>> x = 2
>>> hash(x)
2
>>> x = 4.11
>>> hash(x)
253642731013507076
>>> x = 'hello'
>>> hash(x)
1222179648610370860
>>> x = True
>>> hash(x)
1
>>> x = (1, 2, 3)
>>> hash(x)
529344067295497451
```

Now, what happens if we try this on something unhashable?

```
>>> x = [1, 2, 3]
>>> hash(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

What happens if we try an immutable container (tuple) which contains a mutable object (list)?

```
>>> x = (1, 2, [3])
>>> hash(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Now, a tuple can contain a tuple, which may contain another tuple and so on. All it takes is one mutable element, no matter how deeply nested, to make an object unhashable.

```
>>> x = (1, 2, (3, 4, (5, 6, (7, 8, [9]))))
>>> hash(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Finally, it should go without saying that since dictionaries are mutable, they are not hashable, and thus a dictionary cannot serve as a key for another dictionary.

```
>>> x = {'foo': 'bar'}
>>> hash(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

16.5 Counting letters in a string

Here's an example of how we can use a dictionary to keep track of the number of occurrences of letters or other characters in a string. It's common enough in many word guessing and related games that we'd want this information.

Let's say we have this string: "How far that little candle throws its beams! So shines a good deed in a naughty world."[2]

How would we count all the symbols in this? Certainly, separate variables would be cumbersome. Let's use a dictionary instead. The keys in the dictionary are the individual letters and symbols in the string, and the values will be their counts. To simplify things a little, we'll convert the string to lower case before counting.

[2] William Shakespeare, *The Merchant of Venice*, Act V, Scene I (Portia).

```
s = "How far that little candle throws its beams! " \
    "So shines a good deed in a naughty world."

d = {}
for char in s.lower():
    try:
        d[char] += 1
    except KeyError:
        d[char] = 1
```

We start with an empty dictionary. Then for every character in the string, we try to increment the value associated with the dictionary key. If we get a `KeyError`, this means we haven't seen that character yet, and so we add a new key to the dictionary with a value of one. After this code has run, the dictionary, `d`, is as follows:

```
{'h': 5, 'o': 6, 'w': 3, ' ': 16, 'f': 1, 'a': 7, 'r': 3,
 't': 7, 'l': 4, 'i': 4, 'e': 6, 'c': 1, 'n': 4, 'd': 5,
 's': 6, 'b': 1, 'm': 1, '!': 1, 'g': 2, 'u': 1, 'y': 1,
 '.': 1}
```

So we have five 'h', six 'o', three 'w', and so on.

We could write a function that reports how many of a given character appears in the string like this:

```
def get_count(char, d):
    try:
        return d[char]
    except KeyError:
        return 0
```

This function returns the count if `char` is in `d` or zero otherwise.

16.6 Sets

Sets are another mutable type. Python sets are unordered collections of objects. Like the sets you've seen in your mathematics course, each element in a Python set must be distinct. Like dictionary keys, the elements of a Python set must be hashable.

We write set literals with curly braces (like dictionaries), but without the colon since sets aren't comprised of key / value pairs.

```
>>> s1 = {1, 2, 3}
>>> s2 = {'red', 'green', 'blue'}
```

Just like lists and tuples (and other types) there's a set constructor `set()`, which takes some iterable and returns a set constructed from the iterable.

```
>>> set([1, 2, 3])
{1, 2, 3}
>>> set(('red', 'green', 'blue'))  # notice order isn't preserved
{'red', 'blue', 'green'}
>>> set(range(1, 6))
{1, 2, 3, 4, 5}
>>> set({'a': 'b', 'c': 'd'})      # iterates over keys by default
{'a', 'c'}
```

Sets have abundant applications, not the least of which is removing duplicates from a list.

```
>>> list_with_duplicates = [1, 1, 2, 2, 5, 3, 3, 4]
>>> list(set(list_with_duplicates))
[1, 2, 3, 4, 5]
```

If you try to construct a set with any iterable containing duplicates, or with a set literal containing any duplicates, the duplicates will be removed.

```
>>> {1, 2, 2, 3}
{1, 2, 3}
>>> set((0, 0, 0, 0))
{0}
```

As noted, the elements of a set must be hashable, so these will fail:

```
>>> {1, 2, [3, 4]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> {{1: 2}}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

You can create an empty set with the set constructor and no arguments: `set()`. You cannot create an empty set with `{}`—this creates an empty dictionary.

Python's built-in `len()` will give you the number of elements in a set, and the keyword `in` can be used to test membership.

```
>>> s = {'apple', 'bear', 'candy', 'duckling'}
>>> len(s)
4
>>> 'apple' in s
True
>>> 'wombat' in s
False
```

Mutating sets

We can mutate sets with set methods `.add()`, `.remove()`, `.discard()`, and `.pop()` (there are other set methods to mutate sets but these will suffice for now).

```
>>> s = {'apple', 'bear', 'candy', 'duckling'}
>>> s.add('echidna')
>>> s
{'candy', 'duckling', 'echidna', 'apple', 'bear'}
```

If we try to add an element that's already in the set, nothing happens.

```
>>> s = {'apple', 'bear', 'candy', 'duckling'}
>>> s.add('echidna')
>>> s
{'candy', 'duckling', 'echidna', 'apple', 'bear'}
>>> s.add('echidna')
>>> s
{'candy', 'duckling', 'echidna', 'apple', 'bear'}
```

This does not cause an error, it just fails silently.

We can use `.remove()` to remove an element from a set if it exists. If the element does not exist in the set `.remove()` will result in a `KeyError`. If we wish to discard an element if it exists, or fail silently if it does not, we can use `.discard()`.

```
>>> s.remove('candy')
>>> s
{'duckling', 'echidna', 'apple', 'bear'}
>>> s.remove('wombat')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'wombat'
>>> s.discard('wombat')
```

Because sets are unordered, `.pop()` will pop an arbitrary element from a list.

```
>>> s = {'apple', 'bear', 'candy', 'duckling'}
>>> s.pop()
'bear'
>>> s.pop()
'candy'
```

As with dictionaries, we cannot pop from an empty set.

```
>>> s = set()
>>> s.pop()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

Sets are unordered. Can we iterate over a set?

Yes, we can but we don't always get elements in the order we might expect.

```
for element in some_set:    # this is just fine
    print(element)
```

Comprehension check

1. Is order of elements always preserved when constructing a set?
2. Can a tuple containing a list of strings be an element within a set?
3. Write a set literal listing your three favorite movie titles.

Set operations

Now here's where sets get interesting. Python supports basic set operations such as subset, union, intersection, difference, and others.

In mathematics, set A is a *subset* of set B if all the elements of A are also elements of B. We write this $A \subseteq B$. We write this $A \subsetneq B$ if we mean A is a *strict subset* of B—all the elements of A are also elements of B, but there's at least one element in B not in A. (You may have seen the notation $A \subset B$. Alas, this is ambiguous and usage varies by author. Sometimes this is equivalent to $\subseteq$, sometimes this is equivalent to $\subsetneq$. I'm going to stick with the unambiguous forms.)

The *union* of two sets, A and B, is the set containing all elements of A and all elements of B. We write this $A \cup B$.

The *intersection* of two sets, A and B, is the set containing only those elements which are in both A and B. We write this $A \cap B$.

The *difference* of two sets, A and B, written $A \setminus B$ is all the elements in A that are not also in B.

`.issubset()` is used to determine if one set is a subset of another. `.union()` takes the union of two sets. `.intersection()` takes the intersection of two sets. `.difference()` takes the difference of two sets.

```
>>> s1 = {'red', 'green', 'blue'}
>>> s2 = {'cyan', 'blue', 'magenta', 'yellow'}
>>> s3 = {'red', 'green'}
>>> s1.issubset(s2)    # is s1 a subset of s2?
False
>>> s3.issubset(s1)    # is s3 a subset of s1?
True
>>> s1.union(s2)       # take the union of s1 and s2
```

```
{'green', 'magenta', 'cyan', 'red', 'yellow', 'blue'}
>>> s2.union(s1)       # same
{'green', 'magenta', 'cyan', 'red', 'yellow', 'blue'}
>>> s1.intersection(s2)   # take the intersection of s1 and s2
{'blue'}
>>> s2.intersection(s3)   # this gets us the empty set
set()
>>> s1.difference(s2)     # set difference: s1 \ s2
{'red', 'green'}
```

Notice that in the case of `.issubset()` and `.difference()` order is significant, but in the case of `.union()` and `.intersection()` it is not.

```
>>> s1 = {'red', 'green', 'blue'}
>>> s2 = {'cyan', 'blue', 'magenta', 'yellow'}
>>> s3 = {'red', 'green'}
>>> s1.issubset(s3)
False
>>> s3.issubset(s1)
True
>>> s1.difference(s3)
{'blue'}
>>> s3.difference(s1)
set()
>>> s1.union(s2)
{'green', 'magenta', 'cyan', 'red', 'yellow', 'blue'}
>>> s2.union(s1)
{'green', 'magenta', 'cyan', 'red', 'yellow', 'blue'}
>>> s1.intersection(s2)
{'blue'}
>>> s2.intersection(s1)
{'blue'}
```

We can test for equivalence of two sets with `==`, just like you'd expect. Notice that unlike comparing lists and tuples, when comparing sets the order of elements is irrelevant. This is because lists and tuples are ordered, but sets are not.

```
>>> {1, 2, 3} == {3, 2, 1}
True
```

All that matters is whether the two sets contain the same elements, regardless of order.

The empty set is a subset of every set!

In mathematics, the empty set is a subset of any set. Why? Because all the elements in the empty set are elements of any set. Why? Because there aren't any. We call this *vacuously true*. The same holds true in Python—an empty set is a subset of any set.

```
>>> s1 = {'red', 'green', 'blue'}
>>> s2 = {'cyan', 'blue', 'magenta', 'yellow'}
>>> s3 = {'red', 'green'}
>>> empty_set = set()
>>> empty_set.issubset(s1)
True
>>> empty_set.issubset(s2)
True
>>> empty_set.issubset(s3)
True
```

Differences between Python sets and sets in mathematics

In mathematics, there is exactly one empty set—*the* empty set, notated $\emptyset$. In Python, we can create multiple empty sets.

```
>>> s1 = set()
>>> s2 = set()
>>> s1 == s2      # equivalent---they have the same elements
True
>>> s1 is s2      # but not the same object
False
```

The big difference between Python sets and sets in mathematics is that Python sets must be *finite*. Unlike mathematical sets, they cannot have an infinite number of elements. For example, the set of all natural numbers, $\mathbb{N}$, is infinite, as are the sets of all integers ($\mathbb{Z}$), rational numbers ($\mathbb{Q}$), and real numbers ($\mathbb{R}$). For reasons I hope are obvious, we cannot have a set which contains an infinite number of elements in Python (is your computer infinite? No.)

We can represent infinite sets in Python, but not as concrete sets as we've shown here. Representing infinite sets on a finite machine is interesting, but it's outside the scope of this book.

Comprehension check

1. Another way of thinking of set equivalence is that two sets A and B are equivalent if $A \subseteq B$ and $B \subseteq A$. Use this fact to write a Python expression which compares two sets for equivalence without using `==` or `!=`.

2. The *symmetric difference* of two sets A and B consists of all the elements in A or B but not in both. Python provides a method for taking the symmetric difference of two sets, but you don't strictly need it. Write a Python expression that evaluates to the symmetric difference of two sets.

Some practical applications of sets

Let's say we wanted to find the favorite songs two people have in common. Assume we have two sets of song titles: Bibi's 100 favorite songs, and

Liv's 100 favorite songs. To find what they have in common, we could construct a third set in a loop:

```
common_favorites = set()
for s1 in bibis_favorites:
    for s2 in livs_favorites:
        if s1 == s2:
            common_favorites.append(s1)
```

or perhaps a little better:

```
common_favorites = []
for s1 in bibis_favorites:
    if s1 in livs_favorites:
        common_favorites.append(s1)
```

Here's a better way with sets:

```
common_favorites = bibis_favorites.intersect(livs_favorites)
```

Or say you had a card game and your program needed to know what ranks don't appear in either player's hand? Assume `ingrid` is the set of all ranks in Ingrid's hand, and `karen` is the set of all cards in Karen's hand.

```
all_ranks = set("A123456789JQK")
not_in_either = all_ranks.difference(ingrid.union(karen))
```

16.7 Named tuples

Named tuples are like tuples in that they are immutable. However, unlike plain-vanilla tuples, named tuples allow us to give names to the elements they contain. These elements are called *fields*. By giving names to fields we can refer to them by their name, as well as by their index. This can make our code easier to read and to reason about.

To define a named tuple we need to import from the `collections` module. The `collections` module is full of useful varieties of *collections*. Collections are *container datatypes*—objects that contain other objects. It's true that all sequences, including tuples, are containers of a sort, but the ones in the `collections` module have enhanced functionality for special purposes.

Let's see how named tuples work with some practical examples. We've seen coordinates including latitude and longitude, so let's continue along those lines. Let's define a special-purpose `namedtuple` type for coordinates. It makes sense that the fields will be latitude and longitude, and we'll give these the names "lat" and "lon", respectively.

```
>>> from collections import namedtuple
>>> Coordinates = namedtuple('Coordinates', ['lat', 'lon'])
```

Now we've defined a new type of object. We've seen many types of object before: `int`, `list`, `range`, *etc.* This code (above) defines a new class: `Coordinates`. We can use this new class to create objects of type `Coordinates`. How? Well, when we defined this new class, Python automatically equipped it with some features including its own *constructor*. We'll use this constructor to instantiate new objects, but first we need some data.

Here are some coordinates for a few US state capital cities.

City	Latitude	Longitude
Montpelier, VT	44.2601° N	72.5754° W
Concord, NH	43.2201° N	71.5491° W
Augusta, ME	44.3315° N	69.7890° W
Albany, NY	42.6526° N	73.7562° W
Boston, MA	42.3611° N	71.0571° W

When we render west longitudes as floats we use negative numbers (positive values are for longitudes east of the prime meridian). Let's use these data to create some objects.

```
>>> montpelier = Coordinates(44.2601, -72.5754)
>>> concord = Coordinates(43.2201, -71.5491)
>>> augusta = Coordinates(44.3315, -69.7890)
>>> albany = Coordinates(42.6526, -73.7562)
>>> boston = Coordinates(42.3611, -71.0571)
```

Let's unpack this. When we defined the new `Coordinates` class, we specified that these objects would contain two fields, `lat` and `lon`. That's what we did here:

```
>>> Coordinates = namedtuple('Coordinates', ['lat', 'lon'])
```

Then to create instances of our new class, we had to pass the required arguments to the `Coordinates` constructor. That's what we did here:

```
>>> montpelier = Coordinates(44.2601, -72.5754)
```

The first argument passed to the constructor was assigned to the field `lat` and the second argument was assigned to `lon`. We can check this by printing these objects:

```
>>> print(montpelier)
Coordinates(lat=44.2601, lon=-72.5754)
```

This is telling us we have an object of type `Coordinates` where the `lat` field has the value `44.2601` and the `lon` field has the value `-72.5754`. What's great about named tuples is now we can refer to latitude and longitude by name!

```
>>> print("The coordinates of Boston are "
...       f"{boston.lat}° N, {-boston.lon}° W.")
The coordinates of Boston are 42.3611° N, 71.0571° W.
```

Pretty cool, huh?

Named tuples have another benefit. When we define a named tuple, we specify exactly those fields an object may contain, and this is enforced. If we used a plain tuple, nothing would prevent us from creating a tuple with too many or too few elements.

```
>>> sacramento = Coordinates(38.5781)
Traceback (most recent call last):
  File "<python-input-8>", line 1, in <module>
    sacramento = Coordinates(38.5781)
TypeError: Coordinates.__new__() missing 1
           required positional argument: 'lon'
```

So we can't create a `Coordinates` object without the necessary value for longitude. This also fails (trying to add elevation):

```
>>> sacramento = Coordinates(38.5781, -121.4944, 8.1)
Traceback (most recent call last):
  File "<python-input-9>", line 1, in <module>
    sacramento = Coordinates(38.5781, -121.4944, 8.1)
TypeError: Coordinates.__new__() takes 3 positional
           arguments but 4 were given
```

Now, this error message might seem to be incorrect. When we construct objects of this type there's another argument which precedes the ones we supply, but we'll ignore that for now, and if you ever learn object-oriented programming you'll know why. For now, just trust that this is indeed correct.

So named tuples allow us to refer to fields by name rather than by index, *and* they have the added benefit of enforcing a particular structure for our data: a `Coordinates` object must have exactly two fields, the first being the latitude, the second being longitude. *This kind of constraint helps prevent many defects in programming.*

Let's expand on this and create a new named tuple type which generalizes a bit, including a field for location name (location needn't be a city) and elevation, as well as latitude and longitude.

```
>>> Location = namedtuple('Location', ['name', 'lat',
...                                    'lon', 'elevation'])
```

Now let's instantiate some new objects:

```
>>> montpelier = Location('Montpelier', 44.2601, -72.5754, 147.8)
>>> concord = Location('Concord', 43.2201, -71.5491, 88.4)
>>> augusta = Location('Augusta', 44.3315, -69.7890, 13.7)
>>> albany = Location('Albany', 42.6526, -73.7562, 6.1)
>>> boston = Location('Boston', 42.3611, -71.0571, 6.4)
```

Now, let's create a function which computes the great circle distance between locations.

```
from collections import namedtuple
import math

Location = namedtuple('Location', ['name', 'lat',
                                   'lon', 'elevation'])

def distance(c1, c2):
    """Use haversine formula to compute great circle
    distance in km between two locations, c1 and c2. """
    EARTH_RADIUS = 6_371.2  # km
    lat1 = math.radians(c1.lat)
    lon1 = math.radians(c1.lon)
    lat2 = math.radians(c2.lat)
    lon2 = math.radians(c2.lon)
    d_lat = lat2 - lat1
    d_lon = lon2 - lon1
    a = (math.sin(d_lat / 2) ** 2
         + math.cos(lat1) * math.cos(lat2)
         * math.sin(d_lon / 2) ** 2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    return c * EARTH_RADIUS
```

This uses the *haversine formula*. If you're curious about how it works see: https://en.wikipedia.org/wiki/Haversine_formula. But for now, just see how we've used named tuples to implement this. It's easy to get confused with indices (which comes first, latitude or longitude?). But with named fields, we can't make that mistake (another benefit that helps reduce the likelihood of introducing bugs into our code).

```
>>> distance(concord, montpelier)
142.0268062952325
```

That's in kilometers, so that's about 89 miles. Looks correct. Now let's test distance to Sacramento, CA.

```
>>> sacramento = Location('Sacramento', 38.5781, -121.4944, 8.1)
>>> distance(sacramento, montpelier)
4066.6463034784333
```

4,067 kilometers is about 2,527 miles. Yup. That checks out too.

Matters of style and program structure

I know. You're thinking "What's up with the identifier `Location` and why do we define this before our function definitions?" Great question. For most of this book we haven't worried about classes (new types of objects we can define ourselves), but with named tuples we now have classes. Where do they fit in the program structure? Right between constants (if any) and function definitions (if any). Also, we use InitialCaps format for naming classes, hence `Coordinates` and `Location`.

Trying to access a non-existent field

If you try to access a field within a named tuple that does not exist, you'll get an `AttributeError`.

```
>>> from collections import namedtuple
>>> Xyz = namedtuple('Xyz', ['x', 'y', 'z'])
>>> point = Xyz(5.2, 3.9, 2.2)
>>> point.q    # there is no field `q`
Traceback (most recent call last):
  File "<python-input-3>", line 1, in <module>
    point.q
AttributeError: 'Xyz' object has no attribute 'q'
```

When should I use named tuples?

Whenever you have multiple tuples that hold the same kinds of data, you should consider using named tuples. For a single object it's not worth the trouble, but if you have multiple instances and you want to enforce the number and order of fields they contain, and you want to refer to fields by name, you should consider named tuples. Some use cases:

- Any objects with coordinates, for example, latitude and longitude; or x, y, and z coordinates in three-dimensional space, *etc.*
- Read-only records from a database or CSV file.
- MIDI messages / events in a music system ("MIDI" is a standard for musical instrument digital interface). MIDI messages include note-on, note-off, and other controls. For example, a note-on message would include the pitch, velocity, and MIDI channel for a given musical note (MIDI supports multiple channels for multiple instruments).
- Abundant applications in physics and engineering, *e.g.*,
  - resistors with fields for resistance, tolerance, power rating, temperature coefficient of resistance (TCR), or other parameters;
  - particles with mass, charge, and spin;
  - materials with specific properties like density, ductility, elasticity, or tensile strength.

Benefits of named tuples

- well-defined fields
- field number enforced
- immutability
- improved readability with names
- lighter weight than full-scale object-oriented programming
- can help reduce likelihood of defects

Comprehension check

1. Let's say we were reading records from a database (don't worry about how we do that just yet), and we wanted to store read-only data about students as named tuples. How would you define a named tuple to represent a student record with fields: given name, surname, date of birth, home state, and university email address?

2. Assuming we have a valid definition of a `Student` named tuple class, how would we instantiate an object for our old acquaintance Egbert Porcupine? Let's say Egbert's birthday is 1960-01-01 (January 1 1960), his home state is MI (Michigan), and his email address is eporcupi@uvm.edu.

3. Consider the advantages and disadvantages of using a dictionary or a named tuple to represent a student record. Under what conditions would a dictionary be a more appropriate choice? Under what conditions would a named tuple be the better choice?

16.8 Structured data with JSON

JSON (JavaScript object notation) is a widely-used format for representing structured data. Though it was originally developed for use with JavaScript, no JavaScript is needed to use this format. Python has incorporated support for JSON in its standard releases since version 2.6 (2008). JSON has become popular because it can be read easily by humans and computers alike.

JSON can include key / value pairs like Python dictionaries. JSON can include sequences like lists and other values.

So what is JSON exactly? It's a way of representing objects—dictionaries, lists, values—as strings.

For example, we can take a Python dictionary containing key / value pairs, with values that might be lists, tuples, integers, strings, whatever, and turn that into a JSON string. We call this *encoding*. Then we can save that string to a file, or send it over the internet.

In the other direction, we can read strings from a file and convert them into Python dictionaries containing key / value pairs, with values that might be lists, tuples, integers, strings, whatever. We call this *decoding*.

Let's take a look at some small examples. Here's a little Python dictionary representing Spotify streams of top Noah Kahan songs (as of June 2025).[3]

```
d = {'Stick Season': {'total': 1527026407,
                      'daily': 903012},
     'Northern Attitude': {'total': 474141182,
                           'daily': 529964},
     'Dial Drunk': {'total': 465714487,
                    'daily': 355062},
     'Hurt Somebody': {'total': 407002777,
                       'daily': 137980}}
```

Now let's convert this to a JSON string using Python's `json` module. We convert to a string using the `.dumps()` function (short for "dump string"; we refer to encoding a JSON string as "dumping" and decoding as "loading".)

```
import json

# given d (above)
s = json.dumps(d)
print(s)
```

This encodes the string and when we print it, here's what we see:

```
{"Stick Season": {"total": 1527026407, "daily": 903012},
"Northern Attitude": {"total": 474141182, "daily": 529964},
"Dial Drunk": {"total": 465714487, "daily": 355062},
"Hurt Somebody": {"total": 407002777, "daily": 137980}}
```

Look at that. Surprise! The JSON-encoded string looks exactly like a dictionary!

What happens if we JSON encode a list?

```
s = json.dumps(["red", "green", "blue"])
print(s)
```

This prints:

```
["red", "green", "blue"]
```

Amazing! This string looks just like a list!

So you see, there's not a lot of mystery about JSON encoding. That's part of its beauty.

Let's go in the other direction and decode JSON strings.

[3] Data from https://kworb.net/spotify (June 2025)

```
import json

s = """{"Stick Season": {"total": 1527026407, "daily": 903012},
"Northern Attitude": {"total": 474141182, "daily": 529964},
"Dial Drunk": {"total": 465714487, "daily": 355062},
"Hurt Somebody": {"total": 407002777, "daily": 137980}}"""
# Notice s is a string, NOT a dictionary (using triple-quotes
# for multi-line string), and let's say we read this from
# a file or got it from some web API. Now we can turn it into
# a Python dict.

d = json.loads(s)
# Now we have the decoded dictionary. Let's confirm:

print(d['Hurt Somebody'])
# Prints: {'total': 407002777, 'daily': 137980}
print(f"{d['Stick Season']['total']:,d}")
# Prints: 1,527,026,407
```

Here we used `json.loads()` to decode a string and return a Python dictionary.

Let's construct a complete example:

1. Encode a Python dictionary as a JSON string.
2. Write the string to a file.
3. Check the contents of the file.
4. Read the contents of the file and decode to convert back to a Python dictionary.

```
import json

data = {'Egbert': {'born': 1960,
                   'favorite foods': ['grubs', 'pumpkins']},
        'Edwina': {'born': 1968,
                   'favorite foods': ['pickles', 'kale']}}

with open("test.json", 'w') as fh:
    s = json.dumps(data)
    fh.write(s)   # write json to file

s = None      # destroy s so we can test file contents
data = None   # destroy data so we can test json.loads()

with open("test.json", 'r') as fh:
    s = fh.read()
    print(s)                            # 1

data = json.loads(s)
print(data['Egbert']['favorite foods'])  # 2
print(data['Edwina']['born'])            # 3
```

In the above example:

- #1 prints a valid JSON string of the encoded data.
- #2 prints `['grubs', 'pumpkins']`
- #3 prints `1968`

By making this round trip, we confirm everything works as expected. Notice that what we encoded was a dictionary, and what we got back was a dictionary. The original dictionary had string keys, and what we got back were string keys. The values of the original dictionary were themselves dictionaries, and that's exactly what we got back. The values of the nested dictionaries were integers and lists of strings, and that's exactly what we got back. So encoding an object to JSON and recovering the object by decoding *preserves the structure of the original object*!

We can also write and read JSON encoded data from a file, skipping the steps of first encoding to a string, or of reading file contents to a string. To do this we use `json.dump()` instead of `json.dumps()`, and `json.load()` instead of `json.loads()`. `json.dump()` takes two arguments, the first is the data you wish to encode and the second is an open file handle. With `json.load()` we supply a file handle as an argument. Here's revised code:

```
import json

data = {'Egbert': {'born': 1960,
                   'favorite foods': ['grubs', 'pumpkins']},
        'Edwina': {'born': 1968,
                   'favorite foods': ['pickles', 'kale']}}

with open("test.json", 'w') as fh:
    json.dump(data, fh)

data = None   # destroy data so we can test json.loads()

with open("test.json", 'r') as fh:
    data = json.load(fh)

print(data['Egbert']['favorite foods'])  # 1
print(data['Edwina']['born'])            # 2
```

- #1 prints `['grubs', 'pumpkins']`
- #2 prints `1968`

This is the preferred way of writing and reading JSON data with files.

What happens if we try to load something invalid?

`json.loads()` and `json.load()` are used to load JSON-encoded data and return the appropriate objects. It's reasonable to ask what happens if the strings we try to load arent't valid JSON. In this event we get a `JSONDecodeError`—a new kind of exception.

For example, while Python isn't very fussy about whether we use single- or double-quotes to quote a string, JSON has some strong opinions about this.

```
>>> json.loads('"foo"')
'foo'
>>> json.loads("'foo'")
Traceback (most recent call last):
  File "<python-input-21>", line 1, in <module>
    json.loads("'foo'")
    ~~~~~~~~~~^^^^^^^^^
  ...
json.decoder.JSONDecodeError:
     Expecting value: line 1 column 1 (char 0)
```

Here `json.loads()` expects the JSON-encoded string to be wrapped in single quotes, and then there's a string within that. However, `json.loads()` doesn't care about which we use for this:

```
>>> json.loads('17')
17
>>> json.loads("17")
17
```

What else might constitute an invalid string? Here are some examples, each of which would result in a `JSONDecodeError`:

- `"[=]"`
- `(4, 5, 6)`
- `__name__`

Try these and others at the shell.

Pretty-printing JSON

Pretty-printing JSON is … pretty easy. All you need to do is specify an indentation (*e.g.*, `indent=4`) and `json.dumps()` will format with the specified indentation. When you do this, the indentation and newlines are automatically added to the JSON-encoded string, so when you print it's nicely formatted.

```
>>> import json
>>> data = {'Egbert': {'born': 1960,
...                    'favorite foods': ['grubs', 'pumpkins']},
...         'Edwina': {'born': 1968,
...                    'favorite foods': ['pickles', 'kale']}}
>>> s = json.dumps(data, indent=4)   # indent keyword argument
>>> print(s)
{
    "Egbert": {
        "born": 1960,
        "favorite foods": [
            "grubs",
            "pumpkins"
        ]
    },
    "Edwina": {
        "born": 1968,
        "favorite foods": [
            "pickles",
            "kale"
        ]
    }
}
```

Why use JSON?

If we want to save or send data, we must somehow convert to a string (or some binary format, which we won't get into here). We use JSON when we want to preserve structure when saving data to disk or passing it around across the internet. For example, JSON gives us something that plain CSV data cannot. CSV is fine for flat tables of rows and columns, but it cannot handle any other structure effectively. We can't use CSV as a format to store dictionaries, nested lists, or anything other than a tabular structure.

Limitations of JSON

JSON is not without its limitations though. There are some things that cannot be encoded as JSON strings, and some things for which we cannot preserve the original structure. (The JSON module allows for user customization of encoding and decoding but that's beyond the scope of what we'll cover here.)

Common, simple objects such as `int`, `float`, `bool`, `str`, `list`, `tuple`, `dict`, and `NoneType` are all fair game for encoding.

The `NoneType` object is encoded as the string `'null'`, and decoding a string `'null'` yields the `NoneType` object `None`, so be careful if you have strings with the value `'null'`—they will be decoded as `None`.

Boolean literals `True` and `False` will be encoded as `'true'` and `'false'`, so be careful with values `'true'` and `'false'`—they will be decoded as `True` and `False` respectively.

Sets, like `{1, 2, 3}`, cannot be encoded.

Tuples, like `(1, 2, 3)`, are encoded as a list would be:

```
>>> import json
>>> json.dumps((1, 2, 3))
'[1, 2, 3]'
```

So if you JSON encode a tuple, and then decode the resulting string, you'll get back a list, *not* a tuple.

```
>>> import json
>>> json.loads(json.dumps((1, 2, 3)))
[1, 2, 3]
```

JSON requires that strings used as keys be double-quoted (despite the fact that strings can be single or double quoted in Python). For example, here the first succeeds, but the second fails:

```
>>> import json
>>> json.loads('{"a": "b"}')
{'a': 'b'}
>>> json.loads("{'a': 'b'}")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
json.decoder.JSONDecodeError:
     Expecting property name enclosed in double quotes:
     line 1 column 2 (char 1)
```

We cannot encode things like `range` or `enumerate` objects or functions. If we try, we get a `TypeError`.

```
>>> import json
>>> json.dumps(range(10))
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
    json.dumps(range(10))
    ~~~~~~~~~~^^^^^^^^^^^
  ...
TypeError: Object of type range is not JSON serializable
>>> json.dumps(len)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
TypeError: Object of type builtin_function_or_method
           is not JSON serializable
```

What about named tuples? What do you think would happen when we try to JSON encode one of these?

It works, but perhaps not in the way you'd expect, and we don't get a complete round-trip.

```
>>> from collections import namedtuple
>>> import json
>>> Xyz = namedtuple('Xyz', ['x', 'y', 'z'])
>>> xyz = Xyz(1, 2, 3)
>>> json.dumps(xyz)
'[1, 2, 3]'
>>> result = json.loads(json.dumps(xyz))
>>> result
[1, 2, 3]
```

The fields of the named tuple are encoded and decoded as a list. Does this mean we can't use named tuples with JSON? Of course not, but it does mean that if we want to recover a named tuple from a JSON-encoded string we need to take care of that ourselves.

```
>>> Xyz = namedtuple('Xyz', ['x', 'y', 'z'])
>>> xyz = Xyz(1, 2, 3)
>>> result = json.loads(json.dumps(xyz))
>>> Xyz(*result)
Xyz(x=1, y=2, z=3)
```

You may ask: What's up with the `*` in `*result`? That's called a *splat* (formally, a *starred expression*), and this operator unpacks the sequence into its individual elements.[4]

```
>>> lst = [1, 2, 3]
>>> print(*lst)
1 2 3
```

There's another alternative that makes use of the standard features of named tuples. Named tuples can be converted to dictionaries using the `._asdict()` method ("as dictionary").

```
>>> Xyz = namedtuple('Xyz', ['x', 'y', 'z'])
>>> xyz = Xyz(1, 2, 3)
>>> xyz._asdict()
{'x': 1, 'y': 2, 'z': 3}
>>> result = json.loads(json.dumps(xyz._asdict()))
>>> Xyz(**result)
Xyz(x=1, y=2, z=3)
```

This can be more robust than the earlier approach because we preserve the field names rather than depending on their order. Again, you may ask: What's up with the `**` in `**result`? First we had one star, now

[4]The operator '*' when used to unpack values is called a "splat" because someone thought that '*' looks like a squashed bug: SPLAT!

we have two? That's called a *kwarg splat* (kwarg is short for keyword argument), and it unpacks a dictionary into keyword arguments. So if `result` is the dictionary `{'x': 1, 'y': 2, 'z': 3}`, then `Xyz(**result)` is equivalent to `Xyz(x=1, y=2, z=3)`.

```
>>> Xyz(x=1, y=2, z=3)
Xyz(x=1, y=2, z=3)
```

Resources

For more on JSON and Python's JSON module, see:

- Official JSON website[5]
- Python's JSON documentation[6]

16.9 Exceptions

`KeyError`

If you try to read or pop or delete a key from a dictionary which does not exist, a `KeyError` is raised. This is similar to the `IndexError` you've seen in the cases of lists and tuples.

If you encounter a `KeyError` it means the specified key does not exist in the dictionary.

```
>>> furniture = {'living room': ['armchair', 'sofa', 'table'],
...              'bedroom': ['bed', 'nightstand', 'dresser'],
...              'office': ['desk', 'chair', 'cabinet']}
>>> furniture['kitchen']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'kitchen'
```

If you try to remove an element from a set with `.remove()` and the element does not exist within the set, you'll get a `KeyError`.

```
>>> silly_words = {'foo', 'bar', 'baz', 'quux'}
>>> silly_words.remove('garply')
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
    silly_words.remove('garply')
    ~~~~~~~~~~~~~~~~~~^^^^^^^^^^
KeyError: 'garply'
```

[5] https://www.json.org
[6] https://docs.python.org/3/library/json.html

If you want to remove an element from a set, and you aren't sure it exists, and don't want to use `in` to check first or a `try` / `except`, then you can use `.discard()`.

```
>>> silly_words = {'foo', 'bar', 'baz', 'quux'}
>>> silly_words.remove('garply')
>>> silly_words
{'quux', 'foo', 'baz', 'bar'}
```

`TypeError`

If you try to add to a dictionary a key which is not hashable, Python will raise a `TypeError`:

```
>>> d = {[1, 2, 3]: 'cheese'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

If you try to add an element to a set which is not hashable, Python will raise a `TypeError`:

```
>>> s = set()
>>> s.add(["I", "am", "not", "hashable!"])
Traceback (most recent call last):
  File "<python-input-7>", line 1, in <module>
    s.add(["I", "am", "not", "hashable!"])
    ~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: unhashable type: 'list'
```

If you try to create a named tuple, supplying the wrong number of arguments to the constructor, again, you'll get a `TypeError`.

```
>>> from collections import namedtuple
>>> Xyz = namedtuple('Xyz', ['x', 'y', 'z'])
>>> point = Xyz(5.2, 3.9)
Traceback (most recent call last):
  File ".../interactiveshell.py", line 3553, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  ...
TypeError: Xyz.__new__()
     missing 1 required positional argument: 'z'
```

If you try to JSON encode a type which cannot be encoded, Python will raise a `TypeError`:

```
>>> import json
>>> json.dumps(enumerate([1, 2, 3]))
```

```
Traceback (most recent call last):
  File "<python-input-9>", line 1, in <module>
    json.dumps(enumerate([1, 2, 3]))
    ~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^
  ...
TypeError: Object of type enumerate is not JSON serializable
```

`AttributeError`

Named tuples have fixed fields and field names. So if you try to access a field using a name that does not exist for the given type, you'll get an `AttributeError`.

```
>>> from collections import namedtuple
>>> Car = namedtuple('Car', ['make', 'model', 'year', 'color'])
>>> lesters_car = Car('Pontiac', 'Firebird', 1970, 'red')
>>> lesters_car.hp
Traceback (most recent call last):
  File "<python-input-7>", line 1, in <module>
    lesters_car.hp
AttributeError: 'Car' object has no attribute 'hp'
```

`JSONDecodeError`

A `JSONDecodeError` occurs when we try to use `json.loads()` or `json.load()` to decode a string which is not a valid JSON-encoded string.

```
>>> import json
>>> json.loads("Dolittle: I do not understand the human race!")
Traceback (most recent call last):
  File "<python-input-29>", line 1, in <module>
    json.loads("Dolittle: I do not understand the human race!")
    ~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 ...
json.decoder.JSONDecodeError:
     Expecting value: line 1 column 1 (char 0)
```

16.10 Exercises

Exercise 01

Create a dictionary for the following data:

Student	NetID	Major	Courses
Porcupine, Egbert	eporcupi	CS	CS1210, CS1080, MATH2055, ANTH1100
Pickle, Edwina	epickle	BIOL	BIOL1450, BIOL1070, MATH2248, CHEM1150
Quux, Winston	wquux	ARTS	ARTS2100, ARTS2750, CS1210, ARTH1990
Garply, Mephista	mgarply	ENSC	ENSC2300, GEOG1170, FS2020, STAT2870

a. What do you think should serve as keys for the dictionary?

b. Should you use a nested dictionary?

c .What are the types for student, netid, major, and courses?

Exercise 02

Now that you've created the dictionary in Exercise 01, write queries for the following. A *query* retrieves selected data from some data structure. Here the data structure is the dictionary created in exercise 01. Some queries may be one-liners. Some queries may require a loop.

1. Get Winston Quux's major.
2. If you used NetID as a key, then write a query to get the name associated with the NetID epickle. If you used something else as a key, then write a query to get Edwina Pickle's NetID.
3. Construct a list of all students taking CS1210.

Exercise 03

Dictionary keys must be unique. We cannot have duplicate keys in a dictionary.

What do you think happens if we were to enter this into the Python shell?

```
d = {'foo': 'bar', 'foo': 'baz'}
```

What do you think is the value of `d`? Guess first, then check!

Exercise 04

Create this dictionary:

```
d = {'foo': 'bar'}
```

Now pass `d` as an argument to a function which has a single formal parameter, `d_` Within the function modify the dictionary (mutate it), but return `None` (how you modify it is up to you). What happens to `d` in the outer scope?

Note: Don't assign a new value to `d_`. Just mutate it by adding a key, removing a key, changing a value, *etc.*

Exercise 05

Write a function that takes a string as an argument and returns a dictionary containing the number of occurrences of each character in the string.

Exercise 06

a. Write a function that takes an arbitrary dictionary and returns the keys of the dictionary as a list.

b. Write a function that takes an arbitrary dictionary and returns the values of the dictionary as a list.

Exercise 07

Consider this named tuple class, `Track`, representing a song.

```
from collections import namedtuple

Track = namedtuple('Track', ['name', 'artist',
                             'album', 'year'])
```

a. How would you create an instance of this type with the following data: Anti-Hero, Taylor Swift, Midnights, 2022.

b. What if we wanted to add a field for Spotify streams? How would you modify the definition above?

c. What are the pros and cons of adding a field for Spotify streams? Would this be a good design choice? Why or why not?

Exercise 08

While tuples are immutable, and we cannot change the objects they contain, if a tuple contains a mutable object like a list, we can mutate the list.

Consider this named tuple:

```
from collections import namedtuple

Course = namedtuple('Course', ['department', 'number',
                               'name', 'students'])
c = Course('CS', '1210', 'Introduction to Programming', [])
```

a. Would the following work?

```
c.students.append("Mephista Garply")
```

b. Would the following work?

```
c.students = ["Mephista Garply", "Winston Quux"]
```

c. How about this?

```
c.students.sort()
```

Exercise 09

Consider this code:

```
import json

data = {1: {'question': 'The number 7 is prime', 'answer': True},
        2: {'question': 'The number 12 is prime', 'answer': False}}

with open('prime_quiz.json', 'w') as fh:
    json.dump(data, fh)

# Then later on we read the data...

with open('prime_quiz.json', 'r') as fh:
    data = json.load(fh)

for _, q in data.items():
    if q['answer']:
        print(f"{q['question']} is true")
    else:
        print(f"{q['question']} is false")
```

Remember that JSON encodes `True` as `'true'` and `False` as `'false'`. Will this code work as intended or will it print that both numbers are prime because the string `'false'` is truthy?

Exercise 10

Which of these succeed and which fail? If they fail, *why* do they fail?

a. `json.loads('{"a": "b"}')`

b. `json.loads("{'a': 'b'}")`

c. `json.loads('187')`

d. `json.loads("x")`

e. `json.loads('"x"')`

f. `json.dumps("abc")`

g. `json.dumps("'abc'")`

h. `json.dumps([1, 2, 3, 4, 5][1:4])`

Chapter 17

Graphs

We have looked at the *dictionary* data structure, which associates *keys* with *values*, and we've looked at some examples and use cases. Now that we understand dictionaries, we're going to dive into *graphs*. Graphs are a collection of vertices (nodes) connected by edges, that represent relationships between the vertices (nodes). We'll see that a dictionary can be used to represent a graph.

Learning objectives

- You will learn some of the terms associated with graphs.
- You will learn how to represent data using a graph.
- You will learn about searching a graph using breadth-first search.

Terms introduced

- graph
- vertices (nodes)
- edge
- neighbor (common edge)
- adjacent
- breadth-first search

17.1 Introduction to graphs

Graphs are a very versatile data structure. They can be used to represent game states, positions on a map with routes, people in a social network, and so on. Here we will consider graphs that represent positions on a map with routes and that represent friendships in a social network.

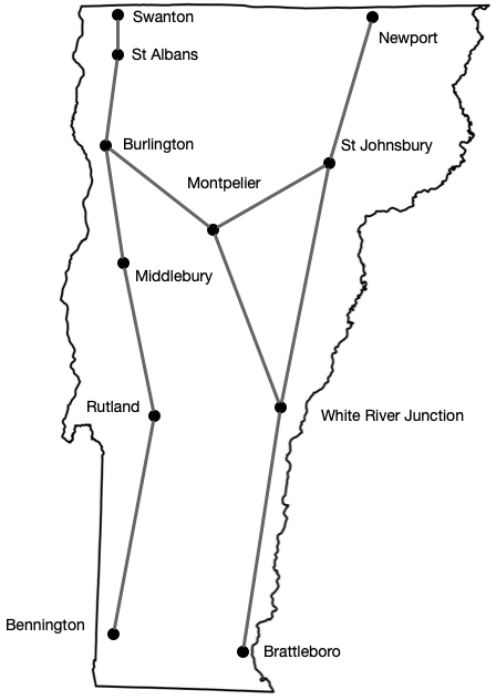
Let's start with maps. Consider a minimal example of two towns in Michigan, Ann Arbor and Ypsilanti.

Here, in the language of graphs, we have two vertices (a.k.a., nodes) representing the towns, Ann Arbor and Ypsilanti, and an edge connecting the vertices, indicating that a route exists between them. We can travel

along this route from Ann Arbor to Ypsilanti, and from Ypsilanti to Ann Arbor. Notice the symmetry. This is because the edge between Ann Arbor and Ypsilanti is *undirected*, which is reasonable, since the highway that connects them allows traffic in both directions.[1]

We refer to vertices which share a common edge as *neighbors*. We also refer to vertices which share a common edge as *adjacent*. So in the example above, Ann Arbor and Ypsilanti are neighbors. Ann Arbor is adjacent to Ypsilanti, and Ypsilanti is adjacent to Ann Arbor.

Here's a little more elaborate example:



In this example, Burlington is adjacent to St Albans, Montpelier and Middlebury. Rutland is adjacent to Middlebury and Bennington. (Yes, we're leaving out a lot of detail for simplicity's sake.)

So the question arises: how do we represent a graph in our code? There are several ways, but what we'll demonstrate here is what's called the *adjacency list* representation.

We'll use a dictionary, with town names as keys and the values will be a list of all towns adjacent to the key.

For example, Montpelier is adjacent to St Johnsbury, White River Junction and Burlington, so the dictionary entry for Montpelier would look like this:

[1] There are what are called *directed* edges, like one-way streets, but we'll only deal with undirected edges in this text.

```
ROUTES = {'Montpelier': ['Burlington', 'White River Junction',
                         'St Johnsbury']}
```

A complete representation of adjacencies, given the map above is:

```
ROUTES = {
    'Burlington': ['St Albans', 'Montpelier', 'Middlebury'],
    'Montpelier': ['Burlington', 'White River Junction',
                   'St Johnsbury'],
    'White River Junction': ['Montpelier', 'Brattleboro',
                             'St Johnsbury'],
    'Brattleboro': ['White River Junction'],
    'Newport': ['St Johnsbury'],
    'St Albans': ['Burlington', 'Swanton'],
    'St Johnsbury': ['Montpelier', 'Newport',
                     'White River Junction'],
    'Swanton': ['St Albans'],
    'Middlebury': ['Burlington', 'Rutland'],
    'Rutland': ['Middlebury', 'Bennington'],
    'Bennington': ['Rutland']
}
```

Notice that Montpelier, St Johnsbury, and White River Junction are all neighbors with each other. This is called a *cycle*. If a graph doesn't have any cycles, it is called an *acyclic* graph. Similarly, if a graph contains at least one cycle, then it is called a *cyclic* graph. So, this is a cyclic graph.

17.2 Searching a graph: breadth-first search

It is often the case that we wish to search such a structure. A common approach is to use what is called breadth-first search (BFS). Here's how it works (in the current context):

We keep a list of towns that we've visited, and we keep a queue of towns yet to visit. Both the list of towns and the queue are of type `list`. We choose a starting point (here it doesn't matter which one), and we add it to the list of visited towns, and to the queue. This is how we begin.

Then, in a while loop, as long as there are elements in the queue:

- pop a town from the front of the queue
- for each neighboring town, if we haven't already visited the town:
  - we append the neighboring town to the list of visited towns
  - we append the neighboring town to the back of the queue

At some point, the queue is exhausted (once we've popped the last one off), and we only add unvisited towns to the queue. So this algorithm will terminate.

Once the algorithm has terminated, we have a list of the visited towns, in the order they were visited.

Needless to say, this isn't a very sophisticated approach. For example, we don't consider the distances traveled, and we have a very simple graph. But this suffices for a demonstration of BFS.

A worked example

Here's a complete, worked example of breadth-first search. It might help for you to go over this while checking the map (above).

Say we choose St Johnsbury as a starting point. Thus, the list of visited towns will be St Johnsbury, and the queue will contain only St Johnsbury. Then, in a `while` loop...

First, we pop St Johnsbury from the front of the queue, and we check its neighbors. Its neighbors are Montpelier, Newport, and White River Junction so we append Montpelier, Newport, and White River Junction to the list of visited towns, and to the queue. At this point, the queue looks like this:

```
['Montpelier', 'Newport', 'White River Junction']
```

At the next iteration, we pop Montpelier from the front of the queue. Now, when we check Montpelier's neighbors, we find Burlington, White River Junction, and St Johnsbury. St Johnsbury has already been visited and so has White River Junction, so we leave them be. However, we have not visited Burlington, so we append it to the list of visited towns and to the queue. At this point, the queue looks like this:

```
['Newport', 'White River Junction', 'Burlington']
```

At the next iteration, we pop Newport from the front of the queue. We check Newport's neighbors and find only St Johnsbury, so there's nothing to append to the queue. At this point, the queue looks like this:

```
['White River Junction', 'Burlington']
```

At the next iteration we pop White River Junction from the front of the queue. White River Junction is adjacent to Montpelier (already visited), Brattleboro, and St Johnsbury (already visited). So we append Brattleboro to visited list and to the queue. At this point, the queue looks like this:

```
['Burlington', 'Brattleboro']
```

At the next iteration, Burlington is popped from the queue. Now we check Burlington's neighbors, and we find St Albans, Montpelier (already visited), and Middlebury. Montpelier we've already visited, but St Albans and Middlebury haven't been visited yet, so we append them to the list of visited towns and to the queue. At this point, the queue looks like this:

```
['Brattleboro', 'St Albans', 'Middlebury']
```

At the next iteration, Brattleboro is popped from the front of the queue. Brattleboro is adjacent to White River Junction (already visited). At this point, the queue looks like this:

```
['St Albans', 'Middlebury']
```

At the next iteration, we pop St Albans from the front of the queue. We check St Alban's neighbors. These are Burlington (already visited) and Swanton. So we append Swanton to the visited list and to the queue. At this point, the queue looks like this:

```
['Middlebury', 'Swanton']
```

At the next iteration, we pop Middlebury from the queue, and we check its neighbors. Its neighbors are Burlington (already visited) and Rutland. Rutland is new, so we append it to the visited list and to the queue. At this point, the queue looks like this:

```
['Swanton', 'Rutland']
```

At the next iteration, we pop Swanton from the front of the queue. Swanton's only neighbor is St Albans and that's already visited so we do nothing. At this point, the queue looks like this:

```
['Rutland']
```

At the next iteration we pop Rutland from the front of the queue. Rutland's neighbors are Bennington and Middlebury. We've already visited Middlebury, but we haven't visited Bennington, so we add it to the list of visited towns and to the queue. At this point, the queue looks like this:

```
['Bennington']
```

At the next iteration we pop Bennington. Bennington's sole neighbor is Rutland (already visited), so we do nothing. At this point, the queue looks like this:

```
[]
```

With nothing added to an empty queue, the queue remains empty, and the while loop terminates. (Remember: an empty list is falsey.) At this point we have a complete list of all the visited towns in the order they were visited:

St Johnsbury, Montpelier, Newport, White River Junction, Burlington, Brattleboro, St Albans, Middlebury, Swanton, Rutland, and Bennington.

Why don't we just iterate over the entire route map?

In the example above, the graph is connected—meaning that every town can be reached from every other town by some route. However, this is not always the case. One reason we don't just iterate over all keys in

a dictionary representing routes is that if we did so, we'd discover *all* towns on the map, rather than only those that can be reached from a given starting point.

Consider highways connecting towns and cities in the Hawai'ian islands. You can't drive from Hilo to Honolulu—they're on different islands.

Sometimes we have graphs in which some vertices are not reachable from all other vertices. This can occur in directed graphs (graphs in which the edges have directions, like one-way streets), or in *disconnected* graphs (graphs that contain vertices or components that aren't connected to other components).

Applications

Search algorithms on graphs have abundant applications, including solving mazes and finding routes, web crawlers, games like tic-tac-toe and various board games, and almost anything involving a network of some kind.

Supplemental reading

- https://en.wikipedia.org/wiki/Breadth-first_search
- https://en.wikipedia.org/wiki/Depth-first_search

17.3 Exercises

Exercise 01

Take a look at this dictionary representing a graph (using adjacency list representation):

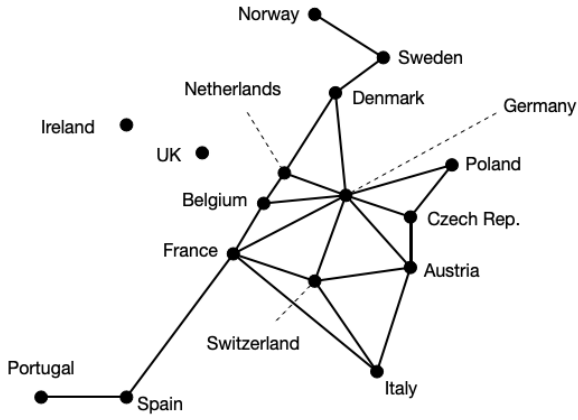
```
FRIENDS = {
    'Alessandro': ['Amelia'],
    'Amelia': ['Sofia', 'Emma', 'Daniel', 'Alessandro'],
    'Ava': ['Mia'],
    'Sofia': ['Amelia', 'Selim', 'Olivia'],
    'Daniel': ['Amelia', 'Emma', 'Selim', 'Olivia'],
    'Emma': ['Amelia', 'Daniel', 'Selim'],
    'Selim': ['Sofia', 'Daniel', 'Emma', 'Ethan'],
    'Olivia': ['Sofia', 'Ethan', 'Daniel'],
    'Ethan': ['Olivia', 'Selim'],
    'Amara': ['Isabella', 'Benjamin'],
    'Benjamin': ['Amara', 'Isabella'],
    'Isabella': ['Amara', 'Benjamin'],
    'Mia': ['Ava', 'James'],
    'James': ['Mia']
}
```

a. On a sheet of paper, perform a breadth-first search, starting with Ethan. Write down the order in which the vertices are visited, and revise the queue as it changes. Draw the graph if you think it will help you. Are all people (nodes) visited by breadth-first search? If not, why not?

b. Is there a symmetry with respect to friendships? How can we see this in the adjacency lists?

Exercise 02

Consider this graph (a portion of Western Europe; edges indicate that one country can be reached from another by driving, without taking a ferry or airplane; dotted lines aren't edges—they're only there to associate labels with certain vertices):



a. Write the adjacency list representation of this graph using a dictionary.

b. Is this graph cyclic or acyclic? Why?

c. Does it matter that countries as shown in the graph aren't in their exact relative positions as they would be in a map of Europe? Why or why not?

d. Is this graph connected or disconnected?

Exercise 03 (challenge!)

In the case of undirected graphs (the only kind shown in this text) edges do not have a direction, and thus if A is connected to B then B is connected to A. We see this in the adjacency list representation: if Selim is a friend of Emma, then Emma is a friend of Selim.

Write a function which takes the adjacency list representation of any arbitrary graph and returns `True` if this symmetry is correctly represented in the adjacency list representation and `False` otherwise. Such a function could be used to *validate* an encoding of an undirected graph.

Appendix A

Glossary

absolute value

The *absolute value* of a number is its distance from the origin (0), that is, the magnitude of the number regardless of its sign. In mathematics this is written with vertical bars on either side of the number or variable in question. Thus

$$|4| = 4$$
$$|-4| = 4$$

and generally,

$$|x| = \begin{cases} x & x \geq 0 \\ -x & x < 0. \end{cases}$$

The absolute value of any numeric literal, variable, or expression can be calculated with the Python built-in, `abs()`.

accumulator

An *accumulator* is a variable that's used to hold a cumulative result within a loop. For example, we can calculate the sum of all numbers from 1 to 10, thus

```
s = 0  # s is the accumulator
for n in range(1, 11):
    s += n  # s is incremented at each iteration
```

adjacent

We say two vertices A, B, in a graph are *adjacent* if an edge exists in the graph with endpoints A and B.

alternating sum

An *alternating sum* is a sum which alternates addition and subtraction depending on the index or parity of the term to be summed. For example:

```
s = 0
for n in range(1, 11):
    if n % 2:     # n is odd
        s -= n    # subtract n
    else:         # n must be even
        s += n    # add n
```

implements the alternating sum

$$0 - 1 + 2 - 3 + 4 - 5 + 6 - 7 + 8 - 9 + 10$$

or equivalently

$$\sum_{i=0}^{10} (-1)^i \, x_i.$$

Alternating sums appear quite often in mathematics.

argument

An *argument* is a value (or variable) that is passed to a function when it is called. An argument is then assigned to the corresponding formal parameter in the function definition. For example, if we call Python's built-in `sum()` function, we must provide an argument (the thing that's being summed).

```
s = sum([2, 3, 5, 7, 11])
```

Here, the argument supplied to the `sum()` function is the list `[2, 3, 5, 7, 11]`. See: *formal parameter*.

Most arguments that appear in this text are *positional arguments*, that is, the formal parameters to which they are assigned depend on the order of the formal parameters in the function definition. The first argument is assigned to the first formal parameter (if any). The second argument is assigned to the second formal parameter, and so on. The number of positional arguments must agree with the number of positional formal parameters, otherwise a `TypeError` is raised.

See also: keyword argument.

arithmetic mean

The *arithmetic mean* is what's informally referred to as the average of a set of numbers. It is the sum of all the numbers in the set, divided by the number of elements in the set. Generally,

$$\mu = \frac{1}{N} \sum_{i=0}^{N-1} x_i.$$

This can be implemented in Python:

```
m = sum(x) / len(x)
```

where `x` is some list or tuple of numeric values.

arithmetic sequence

An *arithmetic sequence* of numbers is one in which the difference between successive terms is a constant. For example, 1, 2, 3, 4, 5,... is an arithmetic sequence because the difference between successive terms is the constant, 1. Other examples of arithmetic sequences:

$$2, 4, 6, 8, 10, ...$$
$$7, 10, 13, 16, 19, 22, ...$$
$$44, 43, 42, 41, 40, 39, ...$$

Python's `range` objects are a form of arithmetic sequence. The stride (which defaults to 1) is the difference between successive terms. Example:

```
range(3, 40, 3)
```

when iterated counts by threes from three to 39.

assertion

An *assertion* is a statement about something you believe to be true. "At the present moment, the sky is blue." is an assertion (which happens to be true where I am, as I write this). We use assertions in our code to help verify correctness, and assertions are often used in software testing. Assertions can be made using the Python keyword *assert* (note that this is a keyword and *not* a function). Example:

```
assert 4 == math.sqrt(16)
```

assignment

We use *assignment* to bind values to names. In Python, this is accomplished with the assignment operator, `=`. For example:

```
x = 4
```

creates a name `x` (presuming `x` has not been defined earlier), and associates it with the value, 4.

binary code

Ultimately, all instructions that are executed on your computer and all data represented on your computer are in *binary code*. That is, as sequences of 0s and 1s.

bind / bound / binding

When we make an assignment, we say that a value is *bound* to a name—thus creating an association between the two. For example, the assignment `n = 42` *binds* the name `n` to the value 42.

In the case of operator precedence, we say that operators of higher precedence *bind* more strongly than operators of lower precedence. For example, in the expression `1 + 2 * 3`, the subexpression `2 * 3` is evaluated before performing addition because `*` binds more strongly than `+`.

Boolean connective

The *Boolean connectives* in Python are the keywords `and`, `or`, and `not`. See: *Boolean expressions*, and Chapter 8 Branching.

Boolean expression

A *Boolean* expression is an expression with one or more Boolean variables or literals (or variables or literals with truth value), joined by zero or more Boolean connectives. The Boolean connectives in Python are the keywords `and`, `or`, and `not`. For example:

```
(x1 or x2 or x3) and (not x2 or x3 or x4)
```

is a Boolean expression, but so is the single literal

```
True
```

It's important to understand that in Python, almost everything has a truth value (we refer to this as truthiness or falsiness), and that Boolean expressions needn't evaluate to only `True` or `False`.

branching

It is through *branching* that we implement conditional execution of portions of our code. For example, we may wish some portion of our code to be executed only if some condition is true. Branching in Python is implemented with the keywords `if`, `elif`, and `else`. Examples:

```
if x > 1:
   print("x is greater than one")
```

and

```
if x < 0:
   print("x is less than zero")
elif x > 1:
   print("x is greater than one")
else:
   print("x must be between 0 and 1, inclusive")
```

It's important to understand that in any given `if`/`elif`/`else` compound statement, only one branch is executed—the first for which the condition is true, or the `else` clause (if there is one) in the event that none of the preceding conditions is true.

Try/except can be considered another form of branching. See: Exception Handling (Chapter 15).

bug

Simply put, a *bug* is a defect in our code. I hesitate to call syntax errors "bugs". It's best to restrict this term to semantic defects in our code (and perhaps unhandled exceptions which should be handled). In any event, when you find a bug, fix it!

bytecode

Bytecode is an intermediate form, between source code (written by humans) and binary code (which is executed by your computer's central processor unit (CPU)). Bytecode is a representation that's intended to be efficiently executed by an interpreter. Python, being an interpreted language, produces an intermediate bytecode for execution on the Python virtual machine (PVM). Conversion of Python source code to intermediate bytecode is performed by the Python compiler.

call (see: invoke)

When we wish to use a previously-defined function or method, we *call* the function or method, supplying whatever arguments are required, if any. When we call a function, flow of execution is temporarily passed to the function. The function does its work (whatever that might be), and then flow of control and a value are returned to the point at which the function was called. See: Chapter 5 Functions, for more details.

```
print("Hello, World!")  # call the print() function
x = math.sqrt(2)        # call the sqrt() function
```

Note: *call* is synonymous with *invoke*.

camel case

Camel case is a naming convention in which an identifier composed of multiple words has the first letter of each interior word capitalized. Sometimes stylized thus: camelCase. See: PEP 8 for appropriate usage.

central tendency

In statistics, the arithmetic mean is one example of a measure of *central tendency*, measuring a "central" or (hopefully) "typical" value for a data set. Another aspect of central tendency is that values tend to cluster around some central value (for example, mean).

comma separated values (CSV)

The *comma separated values* format (CSV) is a commonly used way to represent data that is organized into rows and columns. In this format,

commas are used to separate values in different columns. If commas appear within a value, say in the case of a text string, the value is delimited with quotation marks. Python provides a convenient module—the `csv` module for reading, parsing, and iterating rows in a CSV file.

command line interface (CLI)

Command line interfaces (abbreviated CLI) are user interfaces where the user interacts with a program by typing commands or responding to prompts in text form, or viewing data and responses from the program in similar format. All the programs in this book make use of a command line interface.

comment

Strictly speaking, *comments* are text appearing in source code which is not read or interpreted by the language, but instead, is solely for the benefit of human readers. Comments in Python are delimited by the octothorpe, `#`, a.k.a. hash sign, pound sign, *etc.* In Python, any text on a given line following this symbol is ignored by the interpreter.

In general, it is good practice to leave comments which explain *why* something is as it is. Ideally, comments should not be necessary to explain *how* something is being done (as this should be evident from the code itself).

comparable

Objects are *comparable* if they can be ordered or compared, that is, if the comparison operators—`==`, `<=`, `>=`, `<`, `>`, and `!=` can be applied. If so, the operands on either side of the comparison operators are comparable. Instances of many types can be compared among themselves (any string is comparable to all other strings, any numeric is comparable to all other numerics), but some types cannot be compared with other objects of the same type. For example, we cannot compare objects of type `range` or `function` this way. In most cases, objects of different types are not comparable (with different numeric types as a notable exception). For example, we cannot compare strings and integers.

compiler

A *compiler* is a program which converts source code into machine code, or, in the case of Python, to bytecode, which can then be run on the Python virtual machine.

concatenation

Concatenation is a kind of joining together, not unlike coupling of railroad cars. Some types can be concatenated, others cannot. Strings and lists are two types which can be concatenated. If both operands are a string, or both operands are a list, then the `+` operator performs concatenation (rather than addition, if both operands were numeric).

Examples:

```
>>> 'dog' + 'food'
'dogfood'
>>> [1, 2, 3] + ['a', 'b', 'c']
[1, 2, 3, 'a', 'b', 'c']
```

condition

Conditions are used to govern the behavior of `while` loops and `if`/`elif`/`else` statements. Loops or branches are executed when a condition is true—that is, it evaluates to `True` or has a truthy value. Note: Python supports *conditional expressions*, which are shorthand forms for `if`/`else`, but these are not presented in this text.

congruent

In the context of modular arithmetic, we say two numbers are *congruent* if they have the same remainder with respect to some given modulus. For example, 5 is congruent to 19 modulo 2, because `5 % 2` leaves a remainder of 1 and `19 % 2` also leaves a remainder of 1. In mathematical notation, we indicate congruence with the $\equiv$ symbol, and we can write this example as $5 \equiv 19 \pmod 2$. In Python, the expression `5 % 2 == 19 % 2` evaluates to `True`.

console

Console refers either to the Python shell, or Python input/output when run in a terminal.

constructor

A *constructor* is a special function which constructs and returns an object of a given type. Python provides a number of built-in constructors, for example, `int()`, `float()`, `str()`, `list()`, `tuple()`, *etc.* These are often used in Python to perform conversions between types. Examples:

- `list(('a', 'b', 'c'))` returns the list: `['a', 'b', 'c']`.
- `list('apple')` ‘`explodes'' the string and returns the list:`[‘a’, ‘p’, ‘p’, ‘l’, ‘e’]‘.
- `int(42.1)` returns an integer, truncating the portion to the right of the decimal point: `42`.
- `int('2001')` returns the integer: `2001`.
- `float('98.6')` returns the float: `98.6`.
- `tuple([2, 4, 6, 8])` returns the tuple: `(2, 4, 6, 8)`.

Other constructors include `set()`, `dict()`, `bool()`, `range()` and `enumerate()`.

context manager

Context managers are created using the `with` keyword, and are commonly used when working with file I/O. Context managers take over some of the work of opening and closing a file, or ensuring that a file is closed at the end of a `with` block, regardless of what else might occur within that block. Without using a context manager, it's up to the programmer to ensure that a file is closed. Accordingly, `with` is the preferred idiom whenever opening a file.

Context managers have other uses, but these are outside the scope of this text.

cycle (within a graph)

A *cycle* exists in a graph if there is more than one path of edges from one vertex to another. We refer to a graph containing a cycle or cycles as *cyclic*, and a graph without any cycles as *acyclic*.

delimiter

A *delimiter* is a symbol or symbols used to set one thing apart from another. What delimiters are used, how they are used, and what they delimit depend on context. For example, single-line and inline comments in Python are delimited by the `#` symbol at the start and the newline character at the end. Strings can be delimited (beginning and end) with apostrophes, `'`, quotation marks, `"`, or triple quotation marks, `"""`. In a CSV file, columns are delimited by commas.

dictionary

A *dictionary* is a mutable type which stores data in key/value pairs, like a conventional dictionary. Keys must be unique, and each key is associated with a single value. Dictionary values can be just about anything, including other dictionaries, but keys must be *hashable*. Dictionaries are written in Python using curly braces, with colons used to separate keys and values. Dictionary entries (elements) are separated by commas.

Examples:

- `{1: 'odd', 2: 'even'}`
- `{'apple': 'red', 'lime': 'green', 'lemon': 'yellow'}`
- `{'name': 'Egbert', 'courses': ['CS1210', 'CS2240', 'CS2250'], 'age': 19}`

directory (file system)

A *directory* is a component of a file system which contains files, and possibly other directories. It is the nesting of directories (containment of one directory within another) that gives a file system its hierarchical structure. The top-level directory in a disk drive or volume is called the *root* directory.

In modern operating system GUIs, directories are visually represented as folders, and may be referred to as folders.

dividend

In the context of division (including floor division, `//` and the modulo operator, `%`), the *dividend* is the number being divided. For example, in the expression `21 / 3`, the dividend is `21`.

divisor

In the context of division (including floor division, `//` and the modulo operator, `%`), the *divisor* is the number dividing the dividend. For example, in the expression `21 / 3`, the divisor is `3`. In the context of the modulo operator and modular arithmetic, we also refer to this as the modulus.

driver code

Driver code is an informal way of referring to the code which drives (runs) your program, to distinguish it from function definitions, constant assignments, or classes defined by the programmer. Driver code is often fenced behind the `if` statement: `if __name__ == '__main__':`

docstring

A *docstring* is a triple-quoted string which includes information about a program, module, function, or method. These should not be used for inline comments. Unlike inline comments, docstrings are read and parsed by the Python interpreter. Example at the program (module) level:

```
"""
My Fabulous Program
Egbert Porcupine <eporcupi@uvm.edu>
This program prompts the user for a number and
returns the corresponding frombulation coefficient.
"""
```

Or at the level of a function:

```
def successor(n_):
    """Given some number, n_, returns its successor. """
    return n_ + 1
```

dunder

Special variables and functions in Python have identifiers that begin and end with two underscores. Such identifiers are referred to as *dunders*, a descriptive *portmanteau* of "double underscore." Examples include the variable `__name__` and the special name `'__main__'`, and many other identifiers defined by the language. These are generally used to indicate visually that these are system-defined identifiers.

dynamic typing / dynamically typed

Unlike statically typed languages (for example, C, C++, Java, Haskell, OCaml, Rust, Scala), Python is *dynamically typed*. This means that we can rebind new values to existing names, even if these values are of a different type than were bound in previous assignments. For example, in Python, this is A-OK:

```
x = 1         # now x is bound to a value of type int
x = 'wombat'  # now x is bound to a value of type str
x = [3, 5, 7] # ...and now a list
```

This demonstrates *dynamic typing*—at different points in the execution of this code `x` is associated with values of different types. In many other languages, this would result in type errors.

edge (graph)

A graph consists of vertices (also called nodes) and *edges*. An *edge* connects two vertices, and each edge in a graph has exactly two endpoints (vertices). All edges in graphs in this text are *undirected*, meaning that they have no direction. If an edge exists between vertices A and B, then A is adjacent to B and B is adjacent to A.

empty sequence

The term *empty* applies to any sequence which contains no elements. For example, the empty string `""`, the empty list `[]`, and the empty tuple `(,)`—these are all *empty sequences*. We refer to these using the definite article ("the"), because there is only one such object of each type. Of note, empty sequences are falsey.

entry point

The *entry point* of a program is the point at which code begins execution. In Python, the dunder `__main__` is the name of the environment where top-level code is run, and thus indicates the entry point. Example:

```
"""
Some docstring here
"""

def square(x_):
    return x_ * x_

# This is the entry point, where execution of code begins...
if __name__ == '__main__':
    x = float(input("Enter a number: "))
    x_squared = square(x)
    print(f"{x} squared is {x_squared}")
```

escape sequence

An *escape sequence* (within a string) is a substring preceded by a `\` character, indicating that the following character should be interpreted literally and not as a delimiter. Example: `print('The following apostrophe isn\'t a delimiter')`

Escape sequences are also used to represent special values that otherwise can't be represented within a string. For example, `\n` represents a new line and `\t` represents a tab.

Euclidean division

This is the kind of division you first learned in primary school, before you learned about decimal expansion. A calculation involving *Euclidian division* yields a quotient and a remainder (which may be zero). In Python, this is implemented by two separate operators, one which yields the quotient (without decimal expansion), and the other which yields the remainder. These are `//` (a.k.a. floor division) and `%` (a.k.a., modulo) respectively.

So, for example, in primary school, when asked to divide 25 by 3, you'd answer 8 remainder 1. In Python:

```
quotient = 25 // 3   # quotient gets 8
remainder = 25 % 3   # remainder gets 1
```

evaluate / evaluation

Expressions are *evaluated* by reducing them to a value. For example, the *evaluation* of the expression `1 + 1` yields 2. The evaluation of larger expressions proceeds by order of operator precedence or order of function calls. So, for example,

```
>>> import math
>>> x = 16
>>> y = 5
>>> int(math.sqrt(x) + 4 * y + 1)
25
```

The call to `math.sqrt()` is evaluated first (yielding 4.0). The multiplication `4 * y` is evaluated next (yielding 20). Then addition is performed (4.0 + 20 + 1, yielding 25.0). Finally, the `int` constructor is called, and the final evaluation of this expression is 25.

When literals are evaluated, they evaluate to themselves.

exception

An *exception* is an error that occurs at run time. This occurs when code is syntactically valid, but it contains semantic defects (bugs) or receives unexpected values. When Python detects such an error, it raises an exception. If the exception is not handled, program execution terminates.

Python provides a great many built-in exceptions which help in diagnosing errors. Examples: `TypeError`, `ValueError`, `ZeroDivisionError`, *etc.* Exceptions can be handled using `try` and `except`.

exception handling

Exception handling allows the programmer to anticipate the possibility of certain exceptions that might arise during execution and provide a fallback or means of responding to the exception should it arise. For example, we often use exception handling when validating input from the user.

```
while True:
    response = input("Enter a valid number greater than zero: ")
    try:
        x = float(response)
        if x > 0:
            break
    except ValueError:
        print(f"Sorry I cannot convert {response} to a float!")
```

Exception handling should be as limited and specific as possible. For example, you should never use a bare `except:` or `except Exception:`.

Some types of exception, for example `NameError` should never be handled, as this would conceal a serious programming defect which should be addressed by revising code.

expression

An *expression* is any syntactically valid code that can be evaluated, that is, an expression yields a value. Expressions can be simple (for example, a single literal) or complex (composed of literals, variables, operators, function calls, *etc*).

falsey

When used as conditions or in Boolean expressions, most everything in Python has a truth value. If something has a truth value that is treated as if it were false, we say such a thing is *falsey*. Falsey things include (but are not limited to) empty sequences, `0`, and `0.0`.

Fibonacci sequence

The Fibonacci sequence is a sequence of natural numbers starting with 0, 1, in which each successive element is the sum of the two preceding elements. Thus the Fibonacci sequence begins 0, 1, 1, 2, 3, 5, 8, 13, 21, … The Fibonacci sequence gets its name from Leonardo of Pisa (*c.* 1170–*c.* 1245 CE) whose nickname was Fibonacci. The Fibonacci sequence was known to Indian mathematicians (notably Pingala) as early as 200 BCE, but history isn't always fair about naming these things.

file

A *file* is a component of a file system that's used to store data. The computer programs you write are saved as files, as are all the other documents, programs, and other resources you may have on your computer. Files are contained in directories.

floating-point

A *floating-point* number is a number which has a decimal point. These are represented differently from integers. Python has a type, `float` which is used for floating-point numbers.

floor division

Floor division calculates the Euclidean quotient, or, if you prefer, the largest integer that is less than or equal to the result of floating-point division. For example, `8 // 3` yields 2, because 2 is the largest integer less than 2.666…(which is the result of `8 / 3`).

floor function

The floor function returns the largest integer less than or equal to the argument provided. In mathematical notation, we write

$$\lfloor x \rfloor$$

and, for example $\lfloor 6.125 \rfloor = 6$.

Python implements this function in the `math` module. Example: `math.floor(6.125)` yields 6.

flow chart

A *flow chart* is a tool used to represent the possible paths of execution and flow of control in a program, function, method, or portion thereof. See: Appendix E: Flow Charts.

format specifier

Format specifiers are used within f-string replacement fields to indicate how the interpolated content should be formatted, for example, indicating precision of floating-point numbers; aligning text left, right or center, *etc.* There is a complete "mini-language" of format specifiers. Within the replacement field, the format specifier follows the interpolated expression, separated by a colon. Format specifiers are optional.

folder (see: directory)

Folder is synonymous with directory.

free variable

A *free variable* is a variable which occurs within a code bock, but which is not defined within that code block.

See: Chapter 5 Functions, and the section on the `global` keyword in Chapter 10.

f-string

f-string is short for *formatted string literal*. f-strings are used for string interpolation, where values are substituted into replacement fields within the f-string (with optional formatting specifiers). The syntax for an f-string requires that it be prefixed with the letter "f", and replacement fields appear within curly braces. For example, `f"The secret number is {num}."` In this case, the replacement field is `{num}`, and the string representation of the value associated with the identifier `num` is substituted into the string. So, if we had `num = 42`, then this string becomes "The secret number is 42."

function

In Python, a *function* is a named block of code which performs some task or returns some value. We distinguish between the definition of a function, using the keyword `def`, and calls to the function, which result in the function being executed and then returning to the point at which it was called. Functions may have zero or more formal parameters. Formal parameters receive arguments when the function is called.

All Python functions return a value (the default, if there is no explicit return statement, is for the function to return `None`).

See: Chapter 5 Functions.

graph

A *graph* consists of a set of vertices and a set of edges. Trivially, the empty graph is one with no vertices and thus no edges. A graph with two or more vertices may include edges (assuming we exclude self-edges which is common). An edge connects two vertices.

Graphs are widely used in computer science to represent networks of all kinds as well as mathematical and other objects.

graphical user interface

A *graphical user interface* (abbreviated GUI, which is pronounced "gooey") is an interface with graphical elements such as windows, buttons, scroll bars, input fields, and other fancy doo-dads. These are distinguished from command line interfaces, which lack these elements and are text based.

hashable

Dictionary keys must be *hashable*. Internally, Python produces a hash (a number) corresponding to each key in a dictionary and uses this to

look up elements in the dictionary. In order for this to work, such hashes must be stable—they may not change. Accordingly, Python disallows the use of non-hashable objects as dictionary keys. This includes mutable objects (lists, dictionaries) or immutable containers of mutable objects (for example, a tuple containing a list). Most other objects are hashable: integers, floats, strings, *etc.*

heterogeneous

Heterogeneous means "of mixed kind or type". Python allows for heterogeneous lists or tuples, meaning that these can contain objects of different types. For example, this is just fine in Python: `['a', 42, False, [x], 101.9]`.

identifier

An *identifier* is a name we give to an object in Python. For many types, we give a name by assignment. Example:

```
x = 1234567
```

gives the value 1234567 the name (identifier) `x`. This allows us to refer to the value 1234567 in our code by its identifier, `x`.

```
print(x)
z = x // 100
# etc
```

We give identifiers to functions using the `def` keyword.

```
def successor(n):
    return n + 1
```

Now this function has the identifier (name) `successor`.

For restrictions on identifiers, see the Python documentation at https://docs.python.org/3/reference/lexical_analysis.html (section 2.3. Identifiers and keywords).

immutable

If an object is *immutable* it means that the object cannot be changed. In Python, `int`, `float`, `str`, and `tuple` are all immutable types. It is true that we can reassign a new value to a variable, but this is different from changing the value itself.

import

We *import* a module for use in our own code by using Python's `import` keyword. For example, if we wish to use the `math` module, we must first import it thus:

```
import math
```

impure functions

Impure functions are functions with side effects. Side effects include reading from or writing to the console, reading from or writing to a file, or mutating (changing) a non-local variable. See: *pure function*.

incremental development

Incremental development is a process whereby we write, and presumably test, our programs in small increments.

index / indices

All sequences have *indices* (the plural of *index*). To each element in the sequence, there corresponds an index. We can use an index to access an individual element within a sequence. For this we use square bracket notation (which is distinct from the syntax used to create a list). For example, given the list

```
>>> lst = ['dog', 'cat', 'gerbil']
```

we can access individual elements by their index. Python is (like the majority of programming languages) zero-indexed so indices start at zero.

```
>>> lst[0]
'dog'
>>> lst[2]
'gerbil'
>>> lst[1]
'cat'
```

Lists, being mutable, support indexed writes as well as indexed reads, so this works:

```
>>> lst[2] = 'hamster'
```

But this won't work with tuples or strings (since they are immutable).

I/O

I/O is short for input/output. This may refer to console I/O, file I/O, or some other form of input and output.

instance / instantiate

An *instance* is an object of a given type once created. For example, 1 is an instance of an object of type `int`. More often, though, we speak of

instances as objects returned by a constructor. For example, when using the `csv` module, we can *instantiate* CSV reader and writer objects by calling the corresponding constructors, `csv.reader()` and `csv.writer()`. What are returned by these constructors are *instances* of the corresponding type.

integrated development environment (IDE)

Integrated development environments are convenient (but not essential) tools for writing code. Rather than a mere text editor, they provide syntax highlighting, hints, and other facilities for writing, testing, debugging, and running code. Python provides its own IDE, called IDLE (integrated development and learning environment) and there are many commercial or open source IDEs: Thonny (worth a look if you're a beginner), JetBrains Fleet, JetBrains PyCharm, Microsoft VS Code, and many others.

interactive mode

Interactive mode is what we're using when we're interacting at the Python shell. At the shell, you'll see the Python prompt >>>. Work at the shell is not saved—so it's suitable for experimenting but not for writing programs. See: *script mode*, below.

interpolation

In this text, when we mention interpolation, we're always talking about string interpolation (and not numeric interpolation or frame interpolation). See: *string interpolation*.

interpreter

An *interpreter* is a program that executes some form of code without it first being compiled into binary machine code. In Python, this is done by the Python interpreter, which interprets bytecode produced by the Python compiler.

invoke (see: call)

Invoke is synonymous with call.

iterable / iterate

An *iterable* object is one we may *iterate*, that is, it is an object which contains zero or more elements, which are ordered and can be taken one at a time. A familiar form of iteration is dealing from a deck of cards. The deck contains zero or more elements (52 for a standard deck). The deck is ordered—which doesn't mean the cards are sorted, it just means that each card has a unique position within the deck (depending on how the deck is shuffled). If we were to turn over the cards from the top of the deck one at a time, until there were no more cards left, we'd have

iterated over the deck. At the first iteration, we might turn over the six of clubs. At the second iteration, we might turn over the nine of diamonds. And so on. That's iterating an iterable.

Iterables in Python include objects of type `str`, `list`, `tuple`, `dict`, `range`, and `enumerate` (there are others as well). When we iterate these objects, we get one element at a time, in the order they appear within the object.[1] When we iterate in a loop, unless there are specific exit conditions (for example, `return` or `break`) iteration continues until the iterable is exhausted (there are no more elements remaining to iterate). Going back to the example of our deck of cards, once we've turned over the 52nd card, we've exhausted the deck, there's nothing left to turn over, and iteration ceases.

keyword

Certain identifiers are reserved by the syntax of any language as *keywords*. Keywords always have the same meaning and cannot be changed by the user. Python keywords we've seen in this text are: `False`, `None`, `True`, `as`, `assert`, `break`, `def`, `del`, `elif`, `else`, `except`, `for`, `if`, `import`, `in`, `not`, `or`, `pass`, `return`, `try`, `while`, and `with`. Feel free to try redefining any of these at the Python shell—you won't succeed.

keyword argument

A keyword argument is an argument provided to a function which is preceded by the name. (Note: keyword arguments have absolutely nothing to do with Python keywords.) Keyword arguments are specified in the function definition. Defining functions with keyword arguments is not covered in this text, but there are a few instances of using keyword arguments, notably:

1. The optional `end` keyword argument to the `print()` function, which allows the user to override the default ending of printed strings (which is the newline character, `\n`).
2. The optional `newline` keyword argument to the `open()` function (which is a little hack to prevent ugly behavior on certain Windows machines).

Keyword arguments, if any, always follow positional arguments.

lexicographic order

Lexicographic order is how Python orders strings and certain other type by default when sorting. This is, essentially, how strings would appear if in a conventional dictionary. For example, when sorting two words, the first letters are compared. If the first letters are different, then the

[1] Dictionaries are a bit of a special case here, since the order in which elements are added to a dictionary is not necessarily the order in which they appear when iterating, but there *is* an underlying order. Moreover, Python does provide an `OrderedDict` type (provided by the `collections` module) which preserves the order of addition when iterating. But these are minor points.

word which contains the first letter which appears earlier in the alphabet appears before the other in the sort. If the first letters are the same, then the second letters are compared, and so on. If the number of letters is different, but all letters that can be compared are the same, then the shorter word appears before the other in the sort. So, for example the word `'sort'` would appear before the word `'sorted'` in a sorted list of words.

list

A list (type `list`) is a mutable container for other objects. Lists are sequences, meaning that their contents are ordered (each object in the list has a specific position within the list). Lists are iterable, meaning that we can iterate over them in a `for` loop. We can create a list by assigning a list literal to a variable:

```
cheeses = ['gouda', 'brie', 'mozzarella', 'cheddar']
```

or we may create a list from some other iterable object using the list constructor:

```
# This creates the list ['a', 'b', 'c']
letters = list(('a', 'b', 'c'))
# This creates the list ['w', 'o', 'r', 'd']
letters = list('word')
```

The list constructor can also be used to make a copy of a list.

literal

A literal is an actual value of a given type. `1` is a literal of the `int` type. `'muffin'` is a literal of the `str` type. `['foo', 'bar', 123]` is a literal of the `list` type. During evaluation, literals evaluate to themselves.

local variable

A *local variable* is one which is defined within a limited scope. The local variables we've seen in this text are those created by assignment within a function.

loop

A *loop* is a structure which is repeated zero or more times, depending on the condition if it's a `while` loop, or depending on the iterable being iterated if it's a `for` loop. `break` and (in some cases) `return` can be used to exit a loop which might otherwise continue.

Python supports two kinds of loops: `while` loops which continue to execute as long as some condition is true, and `for` loops which iterate some iterable.

Matplotlib

Matplotlib is a widely used library for creating plots, animations, and other visualizations of data in Python. It is *not* part of the Python standard library, and so it must be installed before it can be used.

For more, see: https://matplotlib.org.

method

A *method* is a function which is bound to objects of a specific type.[2] For example, we have list methods such as `.append()`, `.pop()`, and `.sort()`, string methods such as `.upper()`, `.capitalize()`, and `.strip()`, and so on. Methods are accessed by use of the dot operator (as indicated in the identifiers above). So to sort a list, `lst`, we use `lst.sort()`; to remove and return the last element from a non-empty list, `lst`, we use `lst.pop()`; to return a capitalized copy of a string, `s`, we use `s.capitalize()`; and so on.

module

A *module* is a collection of Python objects and code. Examples in this book include the `math` module, the `csv` module, the `json` module, the `collections` module, and the `statistics` module. In order to use a module, it must be imported, for example, `import math`. Imported modules are given *namespaces*, and we access functions within a module using the dot operator. For example, when importing the `math` module, the namespace is `math` and we access a function within that namespace thus: `math.sqrt(2)`, as one example.

You can import programs you write yourself if you wish to reuse functions written in another program. In cases like this, your program can be imported as a module.

modulus

When using the modulo operator, we refer to the second operand as the *modulus*. For example, in the case of `23 % 5` the modulus is `5`.

See also: Euclidean division and congruent.

Monte Carlo

Monte Carlo method makes use of repeated random sampling (from some distribution), in order to solve certain classes of problems. For example, we can use the Monte Carlo method to approximate π. The Monte Carlo method is widely used in physics, economics, operations management, and many other domains.

[2] Strictly speaking, a method is a function defined with a class.

mutable

An object (type) is mutable if it can be changed after it's been created. Lists and dictionaries are mutable, whereas objects of type `int`, `float`, `str` and `tuple` are not.

name (see: identifier)

Name is synonymous with *identifier*.

namespace

Namespaces are places where Python objects are stored, and these are very much like but not identical to dictionaries. For example, like dictionary keys, identifiers within a namespace are unique (you can't have two different variables named `x` in the same namespace).

Most often you're working in the global namespace. However, functions, and modules you import have their own namespaces. In the case of functions, a function's namespace is created when the function is called, and destroyed when the function returns. In the case of modules, we refer to elements within a module's namespace using the dot operator (see: Module).

node (graph; see: vertex)

In the context of graphs, *node* and vertex are synonymous.

object

Pretty much anything in Python that's not a keyword or operator or punctuation is an object. Every object has a type, so we have objects of type `int`, objects of type `str`, and so on. Functions are objects of type `function`.

If you learn about object-oriented programming you'll learn how to define your own types, and instantiate objects of those types.

operator

An operator is a special symbol (or combination of symbols) that takes one or more operands and performs some operation such as addition, multiplication, *etc.*, or some kind of comparison. Operators include (but are not limited to) `+`, `-`, `*`, `**`, `/`, `//`, `%`, `=`, `==`, `> <`, `>=`, `<=`, and `!=`. Operators that have a single operand are called *unary operators*, for example, unary negation. Operators that take two operands are called *binary operators*.

Some operators perform different operations depending on the type of their operands. This is called *operator overloading*. Example: If both operands are numeric, `+` performs addition; if both operands are strings, or both operands are lists, `+` performs concatenation.

parameter (and formal parameter)

A function definition may include one or more *parameter* (also called *formal parameter*). These parameters provide names for arguments when the function is called. For example:

```
def square(x):
    return x * x
```

In this example, `x` is the formal parameter, and in order to call the function we must supply a corresponding argument.

```
y = square(12)
y = square(some_variable)
```

The examples above show function calls supplying arguments which are assigned to the formal parameter in the function definition.

Formal parameters exist only within a functions namespace (which is destroyed upon return). See: namespace.

PEP 8

PEP 8 is the official Python style guide. See: https://peps.python.org/pep-0008/

pseudo-random

It is impossible for a computer to produce a truly random number (whatever that might actually be). Instead, they can produce *pseudo-random* numbers which appear random and approximate certain distributions. Pseudo-random number generation is implemented in Python's `random` module. See: Chapter 12 Randomness, games, and simulations, for more.

pure function

A *pure function* is a function without side effects. Furthermore, the output of a pure function (the value returned), depends solely on the argument(s) supplied and the function definition, and given the same argument a pure function will always return precisely the same result. In this regard, *pure functions* are akin to mathematical functions. See: *impure function*.

quantile

In statistics, a *quantile* is a set of points which divide a distribution or data set into intervals of equal probability. For example, if we divide our data into quartiles (a quantile of four parts), then each of the four parts has equal probability. Note that if dividing into n parts we need $n - 1$ values to do so.

Some quantiles have special names. For example, we call the value that divides a distribution, sample or population into two equally probable parts a *median*. If we divide into 100 parts we call that *percentiles*.

quotient

A *quotient* is the result of division, whether floor division or floating-point division. In the case of `/` and `//`, the value yielded is called the quotient.

random walk

A *random walk* is a mathematical object which describes a path in some space (say, the integers) taken by a repeated random sequence of steps. For example, if the space in question is the integers, if we start at zero, we could take a random walk by repeatedly flipping a fair coin and adding one if the toss comes up heads and subtracting one if the toss comes up tails.

A random walk needn't be constrained to a single dimension. For example, we could model the motion of a particle suspended in a fluid (Brownian motion) by a random walk in three-dimensional space.

Random walks are used in modeling many phenomena in engineering, physics, chemistry, ecology, economics, and other domains.

read-evaluate-print loop (REPL)

A *read-evaluate-print* loop is an interactive interface which allows a user to type commands or expressions at a prompt, have them performed or evaluated, and then see the result printed to the console. This is performed in a loop, allowing the user to continue indefinitely until they choose to terminate the session. The *Python shell* is an example.

refactor

To *refactor* code is to rewrite with the objective of improving readability and maintainability. In some cases this results in altered functionality but that is not the primary objective of most refactorings. When should you refactor? Early and often!

remainder

The *remainder* is the quantity left over after performing Euclidean (floor) division. The remainder of such an operation must be in the interval $[0, m)$ where m is the divisor (or modulus). Notice that this is a half-open interval. For example if we divide 31 by 6, the remainder is 1. This is implemented in Python with the modulo operator, `%`. See also: modulo, and relevant sections in Chapter 4.

replacement field

Within an f-string, a *replacement field* indicates where expressions are to be interpolated within the string. Replacement fields are delimited by curly braces. Example: `f"Hello {name}, it's nice to meet you!"`

representation error

Representation error of floating-point numbers is the inevitable result of the fact that the real numbers are infinite and the representation of numbers in a computer is finite—there are infinitely more real numbers than can be represented on a computer using a finite number of bits.

return value

All Python functions return a value, and we call this the *return value*. For example, `math.sqrt()` returns the square root of the argument supplied (with some restrictions). As noted, all Python functions return a value, though in some cases the value returned is `None`. Functions which return `None` include (but are not limited to) `print()` and certain list methods which modify a list in place (for example, `.append()`, `.sort()`).

rubberducking

Rubberducking is a process whereby a programmer tries to explain their code to a rubber duck, and in so doing (hopefully) solves a problem or realizes what needs to be done in order to fix a bug. Rubber ducks are particularly useful in this respect in that they listen without interruption, and, not being too bright, they require the simplest possible explanation from the programmer. If you get stuck, *talk to the duck*!

run time

Run time (or sometimes *runtime*) refers to the time at which a program is run.

scope

Scope refers to the visibility or lifetime of a name (or identifier). For example, variables first defined in assignment statements within a function or formal parameters of a function are *local* to that function, and when the function returns and its namespace is destroyed such local variables are out of scope.

We often refer to *inner scope* as the scope within the body of a function or method, and *outer scope* to the code outside the body of a function.

See also: Chapter 5 Functions, and glossary entry for *namespace*.

script mode

Script mode refers to the mode of operation at work when we run a program that we'd previously written and saved. This is distinct from *interactive mode* which takes place in the Python shell.

seed

A *seed* is a starting point for calculations used to generate a pseudo-random number (or sequence of pseudo-random numbers). Usually, when using functions from the `random` module, we allow the random number generator to use the seed provided by the computer's operating system (which is designed to be as unpredictable as possible). Sometimes, however, and especially in cases where we wish to test code which includes the use of pseudo-random numbers, we explicitly set the seed to a known value. In doing so, we can reproduce the sequence of pseudo-random numbers. See: Chapter 12 Randomness, games, and simulations.

semantics

Semantics refers to the meaning of our code as distinct from the *syntax* required by the language. Bugs are defects of semantics—our code doesn't mean (or do) what we intend it to mean (or do).

sequence unpacking

Sequence unpacking is a language feature which allows us to unpack values within a sequence to individual variables. Examples:

```
>>> lst = [1, 2, 3]
>>> a, b, c = lst
>>> a
1
>>> b
2
>>> c
3
>>> coordinates = (44.47844, -73.19758)
>>> lat, lon = coordinates
>>> lat
44.47844
>>> lon
-73.19758
```

In order for sequence unpacking to work, there must be exactly the same number of variables on the left-hand side of the assignment operator as there are elements to unpack in the sequence on the right-hand side. Accordingly, sequence unpacking is not useful (or perhaps impossible) if there are a large number of elements to be unpacked or the number of elements is not known. This is why we see examples of *tuple unpacking* more than we do of *list unpacking* (since lists are mutable, we can't always know how many elements they contain).

Tuple unpacking is the preferred idiom in Python when working with `enumerate()`. It is also handy for swapping variables.

shadowing

Shadowing occurs when we use the same identifier in two different scopes, with the name in the inner scope shadowing the name in the outer scope. In the case of functions, which have their own namespace, shadowing is permitted (it's syntactically legal) and Python is not confused about identifiers. However, *even experienced programmers are often confused by shadowing*. It not only affects the readability of the code, but it can also lead to subtle defects that can be hard to pin down and fix. Accordingly, shadowing is discouraged (this is noted in PEP 8).

Here's an example:

```
def square(x):
    x = x * x
    return x

if __name__ == '__main__':
    x = float(input("Enter a number and I'll square it: "))
    print(square(x)
```

One confusion I've seen among students arises from using the same name `x`. For example, thinking that because `x` is assigned the result of `x * x` in the body of the function, that the return value is unnecessary:

```
def square(x):
    x = x * x
    return x

if __name__ == '__main__':
    x = float(input("Enter a number and I'll square it: "))
    square(x)
    print(x)  # this prints the x here from the outer scope!
```

The first example, above, is correct (despite shadowing) and the program prints the square of the number entered by the user. In the second example, however, the program does not print the square of the number entered by the user—instead it prints the number that was originally entered by the user.[3]

side effect

A *side effect* is an observable behavior of a function other than simply returning a result. Examples of side effects include printing or prompting the user for information, or mutating a mutable object that's been passed to the function (which affects the object in the outer scope).

Whenever writing a function, any side effects should be included by design and never inadvertently. Hence, pure functions are preferred to impure functions wherever possible.

[3]Yeah, OK, if the user enters 0 or 1, the program will print the square of the number, but as they say, even a broken clock tells the right time twice a day! That's not much consolation, though when the user enters 3 and expects 9 as a result.

slice / slicing

Python provides a convenient notation for extracting a subset from a sequence. This is called *slicing*. For example, we can extract every other letter from a string:

```
>>> s = 'omphaloskepsis'
>>> s[::2]
'opaokpi'
```

We can extract the first five letters, or the last three letters:

```
>>> s[:5]
'ompha'
>>> s[-3:]
'sis'
```

We call the result of such operations *slices*. In general, the syntax is `s[<start>:<stop>:<stride>]`, where `s` is some sequence. As indicated in the above examples, stride is optional.

See: Chapter 10 Sequences for more.

snake case

Snake case is a naming convention in which all letters are lower case, and words are separated by underscores (keeping everything down low, like a snake slithering on the ground). Your variable names and function names should be all lower case or snake case. Sometimes stylized as *snake_case*.

See: PEP 8.

standard deviation

Standard deviation is a measure of variability in a distribution, sample or population. Standard deviation is calculated with respect to the mean.

See: Chapter 14 Data analysis and presentation, for details on how standard deviation is calculated.

statement

"A statement is either an expression or one of several constructs with a keyword, such as `if`, `while` or `for`".[4] Examples include assignment statements, branching statements, loop control statements, and `with` statements. A statement may, however, contain an expression, for example `if x > 0:` or `y = 2 * x + 3`.

For example, consider a simple assignment statement:

```
>>> x = 1
>>>
```

[4]Source: Python Language Reference

Notice that nothing is printed to the console after making the assignment. This is because `x = 1` is a statement and not an expression. Expressions have evaluations, but statements do not.

Don't confuse matters by thinking, for example, that the control statement of a `while` loop has an evaluation. It does not. It is true that the *condition* must be an expression, but the control statement itself does not have an evaluation (nor do `if` statements, `elif` statements, *etc.*).

static typing / statically typed

Some languages (not Python) are *statically typed*. In the case of *static typing* the type of all variables must be known *at compile time* and while the value of such variables may change, their types cannot. Examples of statically typed languages: C, C++, Haskell, OCaml, Java, Rust, Go, *etc.*

stride

Stride refers to the step size in the context of `range` objects and slices. In both cases, the default is 1, and thus can be excluded by the syntax. For example, both `x[0:10]` and `range(0, 10)` are syntactically valid. If, however, we wish to use a different stride, we must supply the argument explicitly, for example, `x[0:10:2]` and `range(0, 10, 2)`.

string

A *string* is a sequence, an ordered collection of characters (or more precisely Unicode code points).

string interpolation

String interpolation is the substitution of values into a string containing some form of placeholder. Python supports more than one form of string interpolation, but most examples given in this text make use of f-strings for string interpolation. There are some use cases which justify the use of earlier, so-called C-style string interpretation, but f-strings have been the preferred method for most other use cases since their introduction in 2002 with Python 3.6.

summation

A *summation* (in mathematical notation, with $\sum$) is merely the addition of all terms in some collection (list, tuple, *etc.*). Sometimes, a summation is nothing more than adding a bunch of numbers. In such cases, Python's built-in `sum()` suffices. In other cases, it is the result of some calculation which must be summed, say, for example, summing the squares of all numbers in some collection. In cases like this, we implement the summation in a loop.

syntax

The *syntax* of a programming language is the collection of all rules which determine what is permitted as valid code. If your code contains syntax errors, a `SyntaxError` (or subclass thereof) is raised.

terminal

Most modern operating systems provide a *terminal* (or more strictly speaking a terminal emulator) which provides a text-based command line interface for issuing commands.

The details of how to open a terminal window will vary depending on your operating system.

top-level code environment

The *top-level code environment* is where top-level code is executed. We specify the top-level code environment (perhaps unwittingly) when we run a Python program (either from the terminal or within an IDE). We distinguish the top-level environment from other modules which we may import as needed to run our code.

See: *entry point*, and Chapter 9 Structure, development, and testing.

truth value

Almost everything (apart from keywords) in Python has a *truth value* even if it is not strictly a Boolean or something that evaluates to a Boolean. This means that programmers have considerable flexibility in choosing conditions for flow of control (branching and loop control).

For example, we might want to perform operations on some list, but only if the list is non-empty. Python obliges by treating a non-empty as having a "true" truth value (we say it's truthy) and by treating an empty list as something "false" or "falsey." Accordingly, we can write:

```
if lst:
    # Now we know the list is not empty and we can
    # do whatever it is we wish to do with the
    # elements of the list.
```

See: Chapter 8 Branching and Boolean expressions, especially sections covering truthy and falsey.

truthy

In Python, many things are treated as if they evaluated to `True` when used as conditions in `while` loops or branching statements. We call such things *truthy*. This includes numerics with non-zero values, and any non-empty sequence, and many other objects.

tuple

A *tuple* is an immutable sequence of objects. They are similar to lists in that they can contain heterogeneous elements (or none at all), but they differ from lists in that they cannot be changed once created.

See: Chapter 10 Sequences for details.

type

Python allows for many different kinds of object. We refer to these kinds as *types*. We have integers (`int`), floating-point numbers (`float`), strings (`str`), lists (`list`), tuples (`tuple`), dictionaries (`dict`), functions (`function`), and many other types. An object's type determines not just how it is represented internally in the computer's memory, but also what kinds of operations can be performed on objects of various types. For example, we can divide one number by another (provided the divisor isn't zero) but we cannot divide a string by a number or by another string.

type inference

Python has limited *type inference*, called "duck typing" (which means if it looks like a duck, and quacks like a duck, chances are pretty good it's a duck). So when we write

```
x = 17
```

Python knows that the identifier `x` has been assigned to an object of type `int` (we don't need to supply a type annotation as is required in, say, C or Java).

However, as noted in the text, Python doesn't care a whit about the types of formal parameters or return values of functions. Some languages can infer these types as well, and thus can ensure that programmers can't write code that calls a function with arguments of the wrong type, or returns the wrong type from a function.

Unicode

Unicode is a widely-used standard for encoding symbols (letters, glyphs, and others). Python has provided full Unicode support since version 3.0, which was released in 2008. For purposes of this textbook, it should suffice that you understand that Python strings can contain letters, letters with diacritic marks (accents, umlauts, *etc.*), letters from different alphabets (from Cyrillic to Arabic to Thai), symbols from non-alphabetic writing systems (Chinese, Japanese, hieroglyphs, Cuneiform, Cherokee, Igbo), mathematical symbols, and a tremendous variety of bullets, arrows, icons, dingbats—even emojis!

unpacking

See: *sequence unpacking*.

variable

Answering the question, What is a *variable*? can get a little thorny. I think it's most useful to think of a variable as a name bound to a value forming a pair—two things, tightly connected.

Take the result of this assignment

```
animal = 'porcupine'
```

Is the variable just the name, `animal`? No. Is the variable just the value, `'porcupine'`? No. It really is these two things *together*: a name attached to a value.

Accordingly, we can speak of a variable as having a name *and* a value.

What's the name of this variable? `animal`.

What's the value of this variable? `'porcupine'`.

We sometimes speak of the *type* of a variable, and while names do not have types in Python, values do.

vertex (graph)

As noted elsewhere, a graph consists of a set of *vertices* (the plural of *vertex*), and a set of edges (which connect vertices).

If we were to represent a highway map with a graph, the cities and towns would be represented by vertices, and the highways connecting them would be the edges of the graph.

Appendix B

Mathematical notation

Note: What follows is *mathematical notation* used in this book, and should not be confused with language elements of Python.

An *ellipsis*, ..., can be read as "and so on." For example, $1, 2, 3, \ldots, 100$ denotes the list of all numbers from 1 to 100, and $1 + 2 + 3 + \cdots + 100$ denotes the sum of all integers from 1 to 100 (inclusive).

Braces are used to denote *sets*, for example, $\{4, 12, 31\}$ is the set containing the elements 4, 12, and 31.

$\in$ denotes *membership* in a set. For example, $4 \in \{4, 12, 31\}$.

$\notin$ is used to indicate that some object is *not* an element of some set. For example, $7 \notin \{4, 12, 31\}$.

We say the set A is a *subset* of B if all the elements of A are also in B. For this we write $A \subseteq B$ to indicate the possibility of A and B being equal (they contain exactly the same elements).

We say the set A is a *strict subset* of B if all the elements of A are also in B, and there's at least one element in B not in A For this we write $A \subsetneq B$.

The empty set, written $\emptyset$ is the set with no elements at all.

$\mathbb{N}$ denotes the set of all *natural numbers*, that is, $0, 1, 2, 3, \ldots$

$\mathbb{Z}$ denotes the set of all *integers*, that is, $\ldots -2, -1, 0, 1, 2, \ldots$

$\mathbb{R}$ denotes the set of all *real numbers*. Unlike the integers, real numbers can be used to measure continuous quantities.

Sometimes we describe a set by stating the properties that must hold for its members. In such cases, we use the vertical bar, $|$, which can be read "such that." For example, $\{x \in \mathbb{R} \mid x \geq 0\}$ is the set of all real numbers greater than or equal to zero.

$\mathbb{Q}$ is the set of all *rational* numbers, that is, $\mathbb{Q} = \{\frac{a}{b} \mid a, b \in \mathbb{Z}, b \neq 0\}$.

The *union* of two sets A and B, written $A \cup B$, is the set of all elements that are either in A or in B or possibly both.

The *intersection* of two sets A and B, written $A \cap B$, is the set of all elements that are in A and in B.

The *set difference* of two sets A and B, written $A \setminus B$, is the set of all elements in A that are not also in B.

$\equiv$ denotes *congruence*. We write $a \equiv b \pmod{m}$ to indicate that a is congruent to b {modulo} m. For example, $5 \equiv 1 \pmod{2}$, and $72 \equiv 18 \pmod{9}$.

$\circ$ is the *composition operator*. For example, $f \circ g$ is the composition of functions f and g. With this notation, composition is performed right-to-left, so it may be helpful to read $f \circ g$ as "f applied after g."

Square brackets denote *closed intervals*, the elements of the interval usually determined by context. For example, $[0, 12] = \{n \in \mathbb{Z} \mid 0 \leq n \leq 12\}$ and $[\pi, 2\pi] = \{x \in \mathbb{R} \mid \pi \leq x \leq 2\pi\}$.

Parentheses denote *open intervals*—those in which the endpoints are not included. For example, $(0, 12) = \{n \in \mathbb{Z} \mid 0 < n < 12\}$ and $(\pi, 2\pi) = \{x \in \mathbb{R} \mid \pi < x < 2\pi\}$ (note the strict inequalities).

Half-open intervals are denoted with a square bracket on one side and a parenthesis on the other. For example, $[0, 12) = \{n \in \mathbb{Z} \mid 0 \leq n < 12\}$ and $(\pi, 2\pi] = \{x \in \mathbb{R} \mid \pi < x \leq 2\pi\}$.

Subscripts are used to denote individual elements within a set or sequence. For example, x_i is the ith element of the sequence X, and we call i the *index* of the element. Note: In this text, indices start with 0.

Σ denotes a *summation*. For example, given the set $X = \{2, 9, 3, 5, 1\}$, $\Sigma\ x_i$ is the sum of all elements of X, that is $2 + 9 + 3 + 5 + 1 = 20$. Sometimes, an operation or operations are applied to the elements of a summation. For example, given the set $X = \{1, 2, 3\}$, $\Sigma\ x_i^2$ is the sum of the squares of all elements of X, that is $1^2 + 2^2 + 3^2 = 1 + 4 + 9 = 14$.

Π (upper-case) denotes a *repeated product*. For example, given the set $X = \{2, 9, 3, 5, 1\}$, $\Pi\ x_i$ is the product of all elements of X, that is $2 \times 9 \times 3 \times 5 \times 1 = 270$.

π (lower-case) is a mathematical constant, the ratio of a circle's circumference to its diameter. This is approximately equal to 3.141592653589793.

In statistics, μ is used to denote the mean of a sample, population, or distribution. You may have seen $\bar{x}$ in other texts. These are different notations for the same thing.

In statistics, σ denotes the standard deviation of a sample, population, or distribution (σ^2 denotes the variance).

$\pm$ denotes "plus or minus" for example, $\mu \pm 2.5\sigma$ or $-b \pm \sqrt{b^2 - 4ac}$.

Appendix C

`pip` and `venv`

Introduction

This covers creation of virtual environments and installing packages for use in your own projects. If you are using an IDE other than IDLE, chances are that your IDE has an interface for managing packages and virtual environments. The instructions that follow are intended more for people who are not using an IDE other than IDLE, or who are the DIY type, or simply those who are more interested in how Python and the Python ecosystem work. What follows is for users of Python version 3.4 or later.

PyPI, `pip`, and `venv`

There is a huge repository of modules you can use in your own projects. This repository is called PyPI—the Python Package Index—and it's hosted at https://pypi.org/.

> The Python Package Index (PyPI) is a repository of software for the Python programming language.
>
> PyPI helps you find and install software developed and shared by the Python community.
>
> –From the PyPI website

There you'll find modules for just about anything—integration with cloud computing services, scientific computing, machine learning, accessing and reading web-based information, creating games, cryptography, and more. There are almost half a million projects hosted on PyPI. It's often the case that we want to install packages hosted on PyPI, which do not ship with Python. Fortunately, there are tools that make this less challenging than it might be otherwise. Two of the most useful of these are `pip` and `venv`.

`pip` is the package installer for Python. You can use this to install any packages listed on PyPI. `pip` will manage all dependencies for you. What are dependencies? It's often the case that one package on PyPI requires another (sometimes dozens), and each of those might require other packages. `pip` takes care of all that for you. `pip` identifies all packages

that might be needed for the package you request and will automatically download and install them, usually with a single command.

The other tool presented here is `venv`. This is, perhaps, a little more abstract and can often confuse beginners, but in most cases it's not too complicated. What `venv` does is create a virtual environment where you can install packages using `pip`. First, we'll walk through the reasons behind virtual environments and how to create one using `venv`. Then we'll see how to activate that virtual environment, install packages using `pip`, and start coding.

Of course, you can follow the directions at https://packaging.python.org/en/latest/tutorials/installing-packages/#creating-and-using-virtual-environments and https://pip.pypa.io/en/stable/installation/, but if you want a somewhat more gentle introduction, read on.

What the heck is a *virtual environment* and why do I want one?

Depending on your operating system and operating system version, your computer may have come with Python installed, or Python may be installed to some protected location. Because of this, we don't usually want to monkey with the OS-installed Python environment (in fact, doing so can break certain components of your OS). So installing packages to your OS-installed (or otherwise protected) Python environment is usually a bad idea. This is where virtual environments come in. With `venv` you can create an isolated Python environment for your own programming projects within which you can change Python versions, install and delete packages, *etc.*, *without ever touching your OS-installed Python environment.*

Another reason you might want to use `venv` is to isolate and control dependencies on a project-by-project basis. Say you're collaborating with Jane on a new game written in Python. You both want to be able to run and test your code in the same environment. With `venv`, again, you can create a virtual environment just for that project, and share instructions as to how to set up the environment. That way, you can ensure if your project uses package XYZ, that you and Jane both have the exact same version of package XYZ in your respective virtual environments. If you work with multiple collaborators on multiple projects, this kind of isolation and control is essential.

So that's why we want to use `venv`. Virtual environments allow us to create isolated installations of Python, along with installed modules. We often create virtual environments for specific projects—so that installing or uninstalling modules for one project does not break the other project or your OS-installed Python.

A virtual environment isn't anything magical. It's just a directory (folder) which contains (among other things) its own version of Python, installed modules, and some utility scripts.

> The `venv` module supports creating lightweight "virtual environments", each with their own independent set of Python packages installed in their site directories. A virtual environment is created on top of an existing Python installation, known as the virtual environment's "base" Python, and may

optionally be isolated from the packages in the base environment, so only those explicitly installed in the virtual environment are available.

When used from within a virtual environment, common installation tools such as `pip` will install Python packages into a virtual environment without needing to be told to do so explicitly.

The syntax for creating a virtual environment is straightforward. At a command prompt,

```
$ python -m venv [name of or path to environment]
```

where `$` represents the command prompt, and we substitute in the name of the environment we wish to create. So if we want to create a virtual environment called "cs1210" we'd use this command:

```
$ python -m venv cs1210
```

This creates a virtual environment named cs1210 in the current directory (you may wish to create your virtual environment elsewhere, but that's up to you).

> If you get an error complaining that there is no `python`…
>
> On some systems, `python` might be named `python3`. If you find yourself in that situation, just substitute `python3` or, on some OSs, `py`, for `python` wherever it appears in the instructions.

In order to use a virtual environment it must be *activated*. Once activated, any installations of modules will install the modules into the virtual environment. Python, when running in this virtual environment will have access to all installed modules.

On macOS

```
$ . ./cs1210/bin/activate
(cs1210) $
```

On Windows (with PowerShell)

```
PS C:\your\path\here >  .\cs1210\Scripts\activate
(cs1210) PS C:\your\path\here >
```

Notice that in each case after activation, the command prompt changed. The prefix, in this case `(cs1210)` indicates the virtual environment that is currently active.

When you're done using a virtual environment you can *deactivate* it. To deactivate, use the `deactivate` command.

If you work in multiple virtual environments on different projects you can just deactivate one virtual environment, activate another, and off you go.

For more on virtual environments and `venv` see: https://docs.python.org/3/library/venv.html.

OK. I have my virtual environment activated. Now what?

Installation, upgrading, and uninstallation of third-party Python modules is done with `pip`. `pip` is an acronym for *package installer for Python*.

> pip is the preferred installer program. Starting with Python 3.4, it is included by default with the Python binary installers.
>
> – Python documentation

When you ask `pip` to install a module, it will fetch the necessary files from PyPI—the public repository of Python packages—then build and install to whatever environment is active. For example, we can use `pip` to install the `colorama` package (`colorama` is a module that facilitates displaying colored text).

First, before using `pip` make sure you have a virtual environment active. Notice that in the examples that follow, this virtual environment is called `my_venv` (what you call yours is up to you). Example:

```
(my_venv) $ pip install colorama
Collecting colorama
  Using cached colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Installing collected packages: colorama
Successfully installed colorama-0.4.6

(my_venv) $
```

At this point, the `colorama` package is installed and ready for use. Pretty easy, huh?

To see what packages you have in your virtual environment you can use `pip freeze`. This will report all installed packages, along with dependencies and version information. You can save this information to a file and share this with others. Then all they need to do is install using this list with `pip install -r <filename here>` (`requirements.txt` is commonly used).

For more information and documentation, see: https://docs.python.org/3/installing/index.html

Appendix D

File systems

Introduction

When we write source code in Python (or any other language) our code is saved in a *file* or *files*. Oftentimes, we want to read data from a file or write data to a file. Accordingly, we must have some basic understanding of files and *file systems*.

While file systems will vary somewhat based on your operating system—macOS, Linux, or Windows—file systems share common features. First, they include files and *directories* (also called "folders"), second, they are organized *hierarchically*, and third, there is some *physical device* on which they are stored.

The physical device on which files and directories are stored is usually a *hard disk drive* (HDD) or *solid state drive* (SSD). Your laptop or desktop computer will have at least one of these.

Files and directories stored on such a device are arranged hierarchically in a tree-like structure. This means that directories can contain files or other directories. Each physical device will have a "root" directory, and may have multiple directories as branches. You may think of files as leaves on a branch. The location of each element in the file system is represented as a *path* going from the root to the element in question.

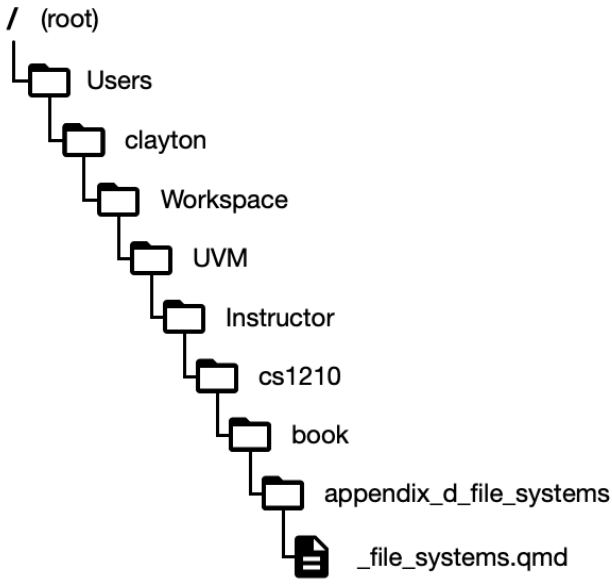
For example, the file on my laptop that contains the text you are reading right now has the path

```
/Users/clayton/Workspace/UVM/Instructor/cs1210/book/...
...appendix_d_file_systems/_file_systems.qmd
```

The root directory is signified by the first `/`. Subsequent slashes separate *segments* in the path. All the intermediate elements are directory names. The last element is the actual file: `_file_systems.qmd`.

This is to say that on my laptop's SSD, there's the root directory. The root directory contains a directory called `Users`. Within the `Users` directory, there's another directory called `clayton`, within that, a directory called `Workspace`, and so on.

All modern operating systems provide a *graphical user interface* for browsing and interacting with the file system (for example, searching for files; creating, moving, deleting, or renaming files or directories; *etc.*), as well as a text-based *command line interface*.



The details of such a tree will vary from machine to machine and from operating system to operating system, but the basic idea is the same: there's a hierarchy of directories, and directories contain other directories or files or both.

You may have noticed that the "tree" representing the file system is drawn upside down, with the root at the top. This is typical for all operating systems. Every element in your file system has a unique path, indicating how to get from the root to that element, be it a directory or file.

Filename extensions

Filenames usually include an *extension* which is used to indicate the type of file it is. For example, Python files have a `.py` extension, as in `hello_world.py`. Other file types make use of different extensions: `.txt` for plain-text files, `.md` for Markdown files, `.csv` for comma-separated value files, and so on.

It's important to note that changing a filename extension doesn't change the content of a file—you can't convert Python source code to a PDF by changing the extension from `.py` to `.pdf`.

You should also be aware that operating systems (macOS, Linux, Windows) *associate* filename extensions with application software, and that operating systems will use this information to launch a particular application associated with a given filename extension when a file is double-clicked in the GUI file browser. For example, if you have a data file in CSV format with a `.csv` extension, double-clicking it will most likely launch Excel if you're using Microsoft Windows, or Numbers if you're using macOS. Sometimes this behavior is desirable, sometimes it

is not. All operating systems allow you to change such associations if you wish.

The good, the bad, the ugly

One nice thing about modern operating systems is that they provide great search tools to find files in your file system. Just enter a keyword, or portion of a file name, and the file browser will show you your file. This is quite handy, and in many cases it eliminates the need to browse the file system to a particular location in search of a file.

One not-so-nice thing about modern operating systems is that they provide great search tools to find files in your file system. This means that casual users often have no idea where a file is actually located—what directory it's contained in, or its full path. The convenience of search functionality can obscure (or make entirely invisible) the hierarchical structure of your file system.

Differences between operating systems

While all modern file systems share the general features outlined above, there are some minor differences. For example, macOS and Linux use "`/`" to separate path segments (in a command line interface), while Windows uses "`\`". Windows assigns *drive letters* to disk drives, the default for the boot disk on a Windows machine being `C`. So on a Windows machine, the path to this file (given above), might be something like this instead:

```
C:\Users\clayton\Workspace\UVM\Instructor\cs1210\book\...
...appendix_d_file_systems\_file_systems.qmd
```

Different operating systems provide different GUIs, but they behave similarly. macOS has the Finder, Windows has its File Explorer, and most Linux distributions offer a choice of file browser: Nautilus, Dolphin, Thunar, Nemo, and Konqueror to name a few. Again, while these are somewhat different, they all provide similar interfaces to similar file systems.

You should familiarize yourself with the file browser for your OS. Make sure you can:

- browse to any given location in your file system,
- find a file without using the built-in search functionality,
- rename a file,
- delete a file,
- create a new, empty folder.

Modern IDEs and browsing files

Many integrated development environments (IDEs) provide a built-in file browser that can be displayed as a pane within the IDE window. Some IDEs commonly used to write Python code include JetBrains' PyCharm, Microsoft's Visual Studio Code, the open source Spyder IDE, and Thonny. All have built-in file browsers.

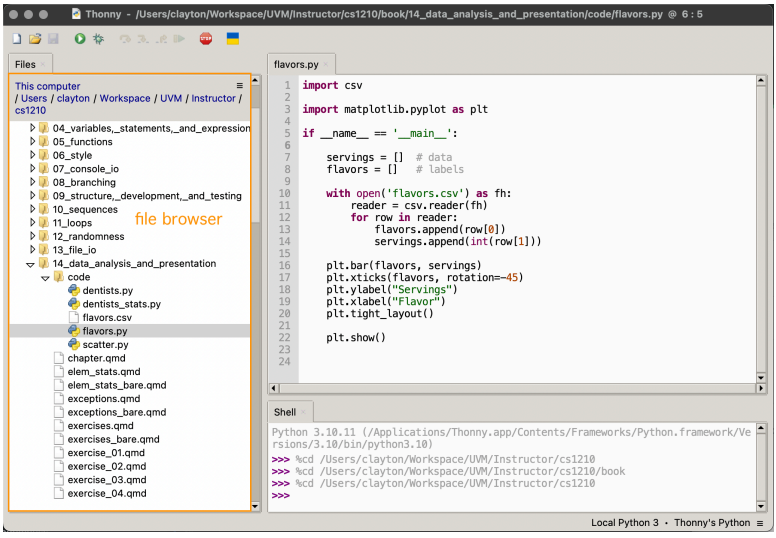


Figure D.1: File browser in Thonny

Some of these file browsers support drag-and-drop operations (for example, to move a file from one directory to another, to move a file from outside a project into a project, *etc.*). Others do not.

You should familiarize yourself with the file browser within the IDE of your choice. Make sure you can:

- browse to a given file,
- create a new Python file (with `.py` extension),
- open a plain-text file (for example, either `.txt` or `.csv` file extension), and
- rename, move, and delete files.

Appendix E

Flow charts

Flow charts are a convenient and often used tool for representing the behavior of a program (or portion thereof) in a diagram. They can help us reason about the behavior of a program before we start writing it. If we have a good flow chart, this can be a useful "blueprint" for a program.

We'll begin with the basics. The most commonly used elements of a flow chart are:

- ellipses, which indicate the start or end of a program's execution,
- rectangles, which indicate performing some calculation or task,
- diamonds, which indicate a decision, and
- directed edges (a.k.a. arrows), which indicate process flow.

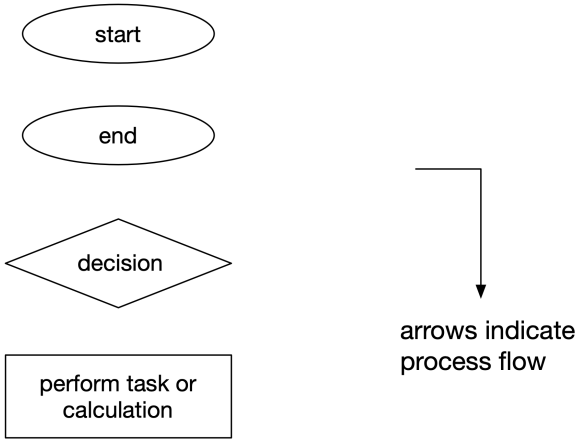


Figure E.1: Basic flow chart elements

Diamonds (decisions) have one input and two outputs. It is at these points that we test some condition. We refer to this as branching—our

path through or flow chart can follow one of two branches. If the condition is true, we take one branch. If the condition is false, we take the other branch.

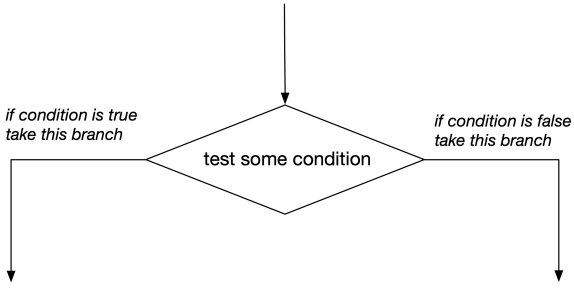


Figure E.2: Branches

A minimal example

Here's a minimal example—a program which prompts a user for an integer, `n`, and then, if `n` is even, the program prints "n is even", otherwise the program prints "n is odd!"

In order to determine whether `n` is even or odd, we'll perform a simple test: We'll calculate the remainder with respect to modulus two and compare this value to zero.[1] If the comparison yields `True` then we know the remainder when dividing by two is zero and thus, `n` must be even. If the comparison yields `False` then we know the remainder is one and thus, `n` must be odd. (This assumes, of course, that the user has entered a valid integer.)

Here's what the flow chart for this program looks like:

[1] Remember, if we have some integer n, then it must be the case that either $n \equiv 0 \mod 2$ or $n \equiv 1 \mod 2$. Those are the *only* possibilities.

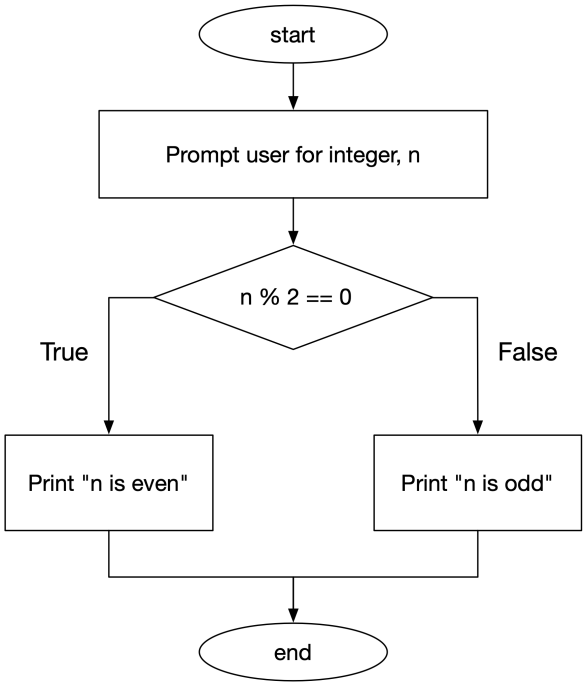


Figure E.3: Even or odd

- We start at the top (the ellipse labeled "start").
- From there, we proceed to the next step: prompting the user for an integer, `n`.
- Then we test to see if the remainder when we divide `n` by two equals zero.
  - If it does, we follow the left branch, and we print "n is even!"
  - Otherwise, we follow the right branch, and we print "n is odd!"
- Finally, our program ends.

Here it is, in Python:

```
"""
CS 1210
Even or odd?
"""

n = int(input('Please enter an integer: '))

if n % 2 == 0:
    print('n is even!')
else:
    print('n is odd!')
```

The branching takes place here:

```
if n % 2 == 0:
    print('n is even!')
else:
    print('n is odd!')
```

Notice there are two branches:

- the `if` clause—the portion that's executed if the expression `n % 2 == 0` evaluates to `True`; and
- the `else` clause—the portion that's executed if the expression `n % 2 == 0` evaluates to `False`.

Another example: Is a number positive, negative, or zero?

Let's say we want to decide if a number is positive, negative, or zero. In this instance, there are *three* possibilities. How do we do this with comparisons that only yield `True` or `False`? The answer is: *with more than one comparison*!

First we'll check to see if the number is greater than zero. If it is, it's positive.

But what if it is *not* greater than zero? Well, in that case, the number could be negative or it could be zero. There are no other possibilities. Why? Because we've already ruled out the possibility of the number being positive (by the previous comparison).

Here's a flow chart:

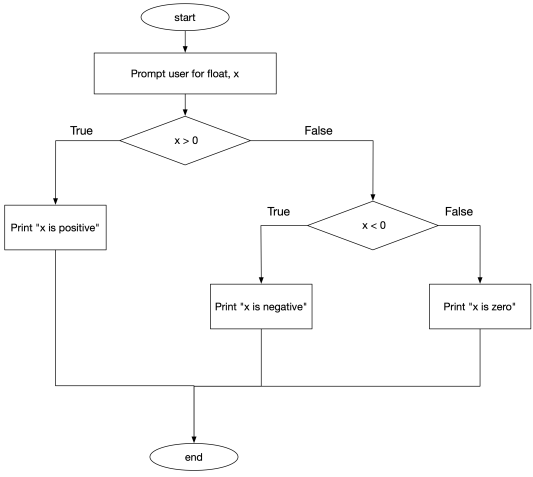


Figure E.4: Positive, negative, or zero

As before, we start at the ellipse labeled "start." Then we prompt the user for a float, `x`. Then we reach the first decision point: Is `x` greater

than zero? If it is, we know `x` is positive, we follow the left branch, we print "x is positive", and we're done.

If `x` is not positive, we follow the right branch. This portion of the flow chart is only executed if the first test yields `False` (that is, `x` is *not* greater than zero). Here, we're faced with another choice: Is `x` *less than* zero? If it is, we know `x` is negative, we follow the left branch (from our second decision point), we print "x is negative", and we're done.

There's one last branch: the one we'd follow if `x` is neither positive nor negative—so it *must* be zero. If we follow this branch, we print "x is zero", and we're done.

Here it is in Python,

```
"""
CS 1210
Positive, negative, or zero?
Using nested if.
"""

x = float(input('Please enter an real number: '))

if x > 0:
    print('x is positive!')
else:
    if x < 0:
        print('x is negative!')
    else:
        print('x is zero!')
```

This structure, which is quite common, is called a "nested if" statement.

Python provides us with another, equivalent way to handle this. We can implement the flow chart for our three-way decision using `elif`, thus:

```
"""
CS 1210
Positive, negative, or zero?
Using elif.
"""

x = float(input('Please enter an real number: '))

if x > 0:
    print('x is positive!')
elif x < 0:
    print('x is negative!')
else:
    print('x is zero!')
```

Both programs—the one with the nested `if` and the one with `elif`—correctly implement the program described by our flow chart. In some cases, the choice is largely a matter of taste. In other cases, we may have reason to prefer one or the other. All things being equal (in terms of

behavior), I think the `elif` solution presented here is the more elegant of the two.

In any event, I hope you see that flow charts are a useful tool for diagramming the desired behavior of a program. With a good flow chart, implementation can be made easier. You should feel free to draw flow charts of programs before you write your code. You may find that this helps clarify your thinking and provides a plan of attack.

Appendix F

Code for cover artwork

Here is the code used to generate the tessellated pattern used on the cover of this book. It's written in Python, and makes use of the DrawSVG package. It's included here because it's a concise example of a complete Python program which includes docstrings (module and function), imported Python modules, an imported third-party module, constants, functions and functional decomposition, and driver code. This is all written in conformance to PEP 8 (or as close as I could manage with the constraint of formatting for book page).

```
"""
Cover background for ITPACS.
Clayton Cafiero <cbcafier@uvm.edu>

Requires DrawSVG package.
To install DrawSVG: `$ pip install "drawsvg[raster]~=2.2"`
See: https://pypi.org/project/drawsvg/

DrawSVG requires Cairo on host system.
Ubuntu: `$ sudo apt install libcairo2`
macOS: `$ brew install cairo`
Anaconda: `$ conda install -c anaconda cairo`
See: https://www.cairographics.org/

For info on SVG 2: https://svgwg.org/svg2-draft/
"""
import math  # 'cause we need a little trig
import drawsvg as draw

# A lovely selection of named SVG colors.
COLORS = ['tomato', 'salmon', 'lightsalmon', 'coral',
          'orangered', 'sandybrown', 'orange']

TRAPEZOID_LEG = 40  # Everything is derived from this...
TRAPEZOID_HEIGHT = math.sin(math.radians(60)) * TRAPEZOID_LEG
```

```
TRAPEZOID_BASE_SEGMENT = TRAPEZOID_LEG / 2
TRAPEZOID_BASE = TRAPEZOID_LEG + 2 * TRAPEZOID_BASE_SEGMENT
TILE_HEIGHT = TRAPEZOID_LEG * math.sqrt(3)
# Taking overlap into consideration we tile @ 120 x 105
COLS = math.ceil(1950 / 120) + 2
ROWS = math.ceil(2850 / 105) + 2


def draw_tile(rotate, fill):
    """If we choose the right starting point for drawing the tile,
    and rotate about that point, we don't have to worry about x, y
    translation. """
    transform = f'rotate({rotate}, 0, 0)'  # angle, center_x, center_y
    return draw.Lines(0, 0,
                      TRAPEZOID_BASE, 0,
                      TRAPEZOID_BASE_SEGMENT + TRAPEZOID_LEG,
                      TRAPEZOID_HEIGHT,
                      TRAPEZOID_BASE_SEGMENT, TRAPEZOID_HEIGHT,
                      0, TILE_HEIGHT,
                      -TRAPEZOID_LEG, TILE_HEIGHT,
                      close=True, stroke_width=0.5,
                      stroke='gray', fill=fill,
                      fill_opacity=1.0, transform=transform)


def draw_cube(trans_x, trans_y, colors_):
    """Assemble a cube from three rotated tiles. """
    transform = f'translate({trans_x}, {trans_y})'
    c = draw.Group(fill='none', stroke='black',
                   transform=transform)
    c.append(draw_tile(0, colors_[0]))
    c.append(draw_tile(120, colors_[1]))
    c.append(draw_tile(240, colors_[2]))
    return c


if __name__ == '__main__':

    d = draw.Drawing(1950, 2850, origin=(0, 0))
    translate_y = 0.0

    for i in range(ROWS):
        if i % 2:
            translate_x = 0.0
        else:
            translate_x = TRAPEZOID_BASE_SEGMENT + TRAPEZOID_LEG
        for j in range(COLS):
            # rotate colors
            colors = [COLORS[(i + j) % len(COLORS)],
                      COLORS[(i + j + 4) % len(COLORS)],
```

```
                    COLORS[(i + j + 8) % len(COLORS)]]
            d.append(draw_cube(translate_x, translate_y, colors))
            translate_x += TRAPEZOID_BASE + TRAPEZOID_LEG
        translate_y += TILE_HEIGHT + TRAPEZOID_HEIGHT

    d.set_pixel_scale(1.0)  # Set number of pixels per geometry unit
    d.save_svg('cover.svg')
    d.save_png('cover.png')
```

Appendix G

Code smells for beginners

Code smells? Seriously? What on earth is a "code smell"?

Have you ever opened some mystery container in your refrigerator and gotten a whiff of something nasty? That's a fridge smell: an indicator that something isn't right.

A **code smell** is an indicator that there's something wrong with your code. A code smell isn't the same as a bug. Your code might still run, even if it's a little smelly. But a code smell is often a sign that code is brittle or in need of refactoring. Accordingly, you should be aware of common code smells so when you catch a whiff, you'll take a second look at your code to see what might be improved.

Some code smells

- Duplicated code
- Long function
- Magic numbers
- Shotgun surgery

There are many other well-known code smells but these will suffice to get you started.

Duplicated code

Whenever you have identical or near identical code, that's often a sign that the duplicated code could be turned into a function and called as needed. Here's an example:

```
player_a = []
for _ in range(5):
    # simulate rolls of five dice
    roll = random.choice([1, 2, 3, 4, 5, 6])
    player_a.append(roll)

player_b = []
for _ in range(5):
    # simulate rolls of five dice
    roll = random.choice([1, 2, 3, 4, 5, 6])
    player_b.append(roll)
```

Here we have a serious code smell, but there's an easy fix. All we need to do is extract the duplicated code and turn it into a function.

```
def get_rolls():
    # simulate rolls of five dice
    lst = []
    for _ in range(5):
        roll = random.choice([1, 2, 3, 4, 5, 6])
        lst.append(roll)
    return lst

player_a = get_rolls()
player_b = get_rolls()
```

Much better, but now, with this code duplication eliminated and a new function created, we can see how easy it would be to make the function more flexible, to accommodate dice with a number of faces other than six and to accommodate a different number of rolls. Where before we'd have to make changes in two places, now we can do it all in one place!

```
def get_rolls(faces, rolls):
    """
    Simulate rolls of a number of dice
    with variable number of faces
    """
    lst = []
    for _ in range(rolls):
        roll = random.choice(range(1, faces + 1))
        lst.append(roll)
    return lst

player_a = get_rolls(6, 5)
player_b = get_rolls(6, 5)
```

Better still. Now, if we want three rolls of dodecahedral (12-face) dice we need only change the last two lines.

```
player_a = get_rolls(12, 3)
player_b = get_rolls(12, 3)
```

There's actually an acronym for a sound principle of software design—DRY: don't repeat yourself.

Long function

This one's not quite so clear cut. What exactly is too long? This will vary from case to case, but a good general rule is that if a function is longer than, say, 25 lines then it's likely ripe for refactoring.

Why is this so? It's often the case that long functions are long because they're doing too much, or are performing a computation which would best be implemented in a number of steps. Good functions have clear and limited responsibilities—a function should do one thing and do it well.

Here's one example of a long function.

```
import math

def compute_entropy():
    while True:
        positive = input("Enter the number of positive samples: ")
        try:
            positive = int(positive)
            if positive >= 0:
                break
            print("Value must be >= 0!")
        except ValueError:
            print(f"Cannot convert {positive} to an integer")
    while True:
        negative = input("Enter the number of negative samples: ")
        try:
            negative = int(negative)
            if negative >= 0:
                break
            print("Value must be >= 0!")
        except ValueError:
            print(f"Cannot convert {negative} to an integer")
    total = positive + negative
    if total == 0:
        print("Cannot compute entropy with zero samples!")
        return None
    term_1 = - (positive / total) * math.log2(positive / total)
    term_2 = - (negative / total) * math.log2(negative / total)
    print(f"Total entropy is {term_1 + term_2:.3f}.")
    return term_1 + term_2

compute_entropy()
```

This code works in most instances. There is a bug, but it's hard to see because this function is too large. This function is trying to do too much. Yes, there's some duplicated code, but this function is unfocused as well. Let's refactor.

First, we'll extract another function to get valid input from the user.

```
import math

def get_num(label):
    """
    This function is responsible for getting a good value
    from the user.
    """
    while True:
        n = input(f"Enter the number of {label} samples: ")
        try:
            n = int(n)
            if n >= 0:
                return n
            print("Value must be >= 0!")
        except ValueError:
            print(f"Cannot convert {n} to an integer")

def compute_entropy():
    positive = get_num('positive')
    negative = get_num('negative')
    total = positive + negative
    if total == 0:
        print("Cannot compute entropy with zero samples!")
        return None
    term_1 = - (positive / total) * math.log2(positive / total)
    term_2 = - (negative / total) * math.log2(negative / total)
    print(f"Total entropy is {term_1 + term_2:.3f}.")
    return term_1 + term_2

compute_entropy()
```

That's a step in the right direction.

Let's continue. `compute_entropy()` is still trying to do too much. It computes a value *and* prints the result *and* returns the result. Computation and display are two very different things, and they should be kept separate. *Separation of concerns* leads to good program design.

Let's separate concerns: computation and display.

```
import math

def get_num(label):
    """
    This function is responsible for getting a good value
    from the user.
    """
    while True:
        n = input(f"Enter the number of {label} samples: ")
        try:
            n = int(n)
            if n >= 0:
                return n
            print("Value must be >= 0!")
        except ValueError:
            print(f"Cannot convert {n} to an integer")

def compute_entropy(pos, neg):
    """
    Compute entropy, given number of positive and negative
    samples.
    """
    total = pos + neg
    if total == 0:
        raise ValueError("Cannot compute entropy with zero samples!")
    p_term = - (pos / total) * math.log2(pos / total)
    n_term = - (neg / total) * math.log2(neg / total)
    return p_term + n_term

positive_samples = get_num('positive')
negative_samples = get_num('negative')
entropy = compute_entropy(positive_samples, negative_samples)
print(f"Total entropy is {entropy:.3f}.")
```

Much better. Now we have `get_num()` which is called twice to get the necessary values from the user, and we have a *pure* function for computing entropy. *Pure functions are easy to test*—so let's do just that (and we'll find our bug).

```
import math

def get_num(label):
    """
    This function is responsible for getting a good value
    from the user.
    """
    while True:
        n = input(f"Enter the number of {label} samples: ")
        try:
            n = int(n)
            if n >= 0:
                return n
            print("Value must be >= 0!")
        except ValueError:
            print(f"Cannot convert {n} to an integer")

def compute_entropy(pos, neg):
    """
    Compute entropy, given number of positive and negative
    samples.
    """
    total = pos + neg
    if total == 0:
        raise ValueError("Cannot compute entropy with zero samples!")
    p_term = - (pos / total) * math.log2(pos / total)
    n_term = - (neg / total) * math.log2(neg / total)
    return abs(p_term + n_term)

def test_compute_entropy():
    assert math.isclose(compute_entropy(4, 5), 0.991076060)
    assert math.isclose(compute_entropy(100, 100), 1.0)
    assert math.isclose(compute_entropy(0, 10), 0.0)


test_compute_entropy()   # will be silent if all OK

positive_samples = get_num('positive')
negative_samples = get_num('negative')
entropy = compute_entropy(positive_samples, negative_samples)
print(f"Total entropy is {entropy:.3f}.")
```

If all is well, tests will pass silently. However, on account of our bug we get the following error.

```
Traceback (most recent call last):
  File "<string>", line 35, in <module>
  File "<string>", line 33, in test_compute_entropy
  File "<string>", line 26, in compute_entropy
ValueError: math domain error
```

On line 33, we have this test:

```
assert math.isclose(compute_entropy(0, 10), 0.0)
```

(The `.isclose()` function provided with the `math` module is used to compare numbers where we aren't concerned with tiny representation or rounding errors. For example, 1000.0000000000000000 and 1000.0000000000000001 are close enough and in most cases we wouldn't want a comparison to fail for a such tiny difference.)

The problem, now exposed, is that we can't take `math.log2(0)`. That's what gives us the math domain error. Logarithms are only defined for positive values. So what can be done? We could do one of three things:

- We can check the value before calling `math.log2()`.
- We can wrap the call to `math.log2()` and supply a default value in the event of an exception.
- We can add a wee tiny epsilon to guarantee that we don't pass zero to `math.log2()`.

Let's roll with the first option. Because this call to `math.log2()` is so intimately connected with the entropy calculation, let's create a helper function inside `compute_entropy()` to implement the fix.

```
import math

def get_num(label):
    """
    This function is responsible for getting a good value
    from the user.
    """
    while True:
        n = input(f"Enter the number of {label} samples: ")
        try:
            n = int(n)
            if n >= 0:
                return n
            print("Value must be >= 0!")
        except ValueError:
            print(f"Cannot convert {n} to an integer")
```

```
def compute_entropy(pos, neg):
    """
    Compute entropy, given number of positive and negative
    samples.
    """
    def get_log(x):
        """Guarded call to math.log2() """
        if x == 0.0:
            return 0.0
        return math.log2(x)

    total = pos + neg
    if total == 0:
        raise ValueError("Cannot compute entropy with zero samples!")
    p_term = - (pos / total) * get_log(pos / total)
    n_term = - (neg / total) * get_log(neg / total)
    return abs(p_term + n_term)

def test_compute_entropy():
    assert math.isclose(compute_entropy(4, 5), 0.991076060)
    assert math.isclose(compute_entropy(100, 100), 1.0)
    assert math.isclose(compute_entropy(0, 10), 0.0)


test_compute_entropy()   # will be silent if all OK

positive_samples = get_num('positive')
negative_samples = get_num('negative')
entropy = compute_entropy(positive_samples, negative_samples)
print(f"Total entropy is {entropy:.3f}.")
```

Now we have nicely structured code that's easy to read and easy to test—and as a result, it's more robust.

Magic numbers

Magic numbers are numbers that… poof!… appear in your code seemingly from nowhere. In the previous example, 0 and 0.0 aren't magic numbers because it's so abundantly clear from context why we need to compare with zero. However, consider this function:

```
def kinetic_energy_per_df(temp):
    temp = temp + 273.15
    return (1 / 2) * 1.380649E-23 * temp
```

Here we have two magic numbers. The first is used to convert degrees Celsius to degrees Kelvin. The second is Boltzmann's constant. Both are constants, and either could be used elsewhere. It's best to define them as Python constants.

```
ZERO_CELSIUS_IN_KELVIN = 273.15
BOLTZMANN_CONSTANT = 1.380649E-23

def kinetic_energy_per_df(temp):
    temp = temp + ZERO_CELSIUS_IN_KELVIN
    return (1 / 2) * BOLTZMANN_CONSTANT * temp
```

This makes the code more readable. We can see what's happening and why at each step. Also, when we write this way, it becomes clear that this function is trying to do two things at once. Since average kinetic energy per degree of freedom is always expressed with respect to degrees Kelvin, it's best if we factor out the first conversion, and leave handling temperature units to some other portion of the code.

```
ABSOLUTE_ZERO_C = -273.15
BOLTZMANN_CONSTANT = 1.380649E-23

def celsius_to_kelvin(deg_c):
    if deg_c < ABSOLUTE_ZERO_C:
        raise ValueError(f"{deg_c} is below absolute zero!")
    return deg_c - ABSOLUTE_ZERO_C

def kinetic_energy_per_df(deg_k):
    return (1 / 2) * BOLTZMANN_CONSTANT * deg_k
```

Better.

Sometimes we see the same constant peppered throughout some code. Defining a constant and then referring to it by name makes code more readable and helps avoid defects introduced when typing. For example, every time we type `8.617333262E-5` (Boltzmann's constant in electron-volts per kelvin), we could introduce a typo. Better to define it once and then use the named constant.

```
BOLTZMANN_CONSTANT_EV = 8.617333262E-5
```

Shotgun surgery

Shotgun surgery isn't something we can always *see* immediately when looking at our code. It's more something that we notice when we start to make changes to our code. If we want to change this little thing *here*, and that breaks something else *over there*, and in order to fix it we need to change code in three other places, that's shotgun surgery—fixes scattered all over the place.

If you find yourself performing shotgun surgery, that's a clear sign your code is in need of refactoring.

```
def send_email(p, message):
    print(f"Sending to {p['email']}: {message}")
    # ...more here

def extract_given_name(p):
    return p["name"].split()[0]

def format_contact(p):
    return f"{p['name']} <{p['email']}>"

def construct_greeting(p):
    return f"Dear {p['name'].split()[0]}:"

person = {
    "name": "Mephista Garply",
    "email": "mxgarply@uvm.edu"
}

# Test
send_email(person, "Hello there...")
print(extract_given_name(person))
print(format_contact(person))
print(construct_greeting(person))
```

When we run this code, everything works as expected.

```
Sending to mxgarply@uvm.edu: Hello there...
Mephista
Mephista Garply <mxgarply@uvm.edu>
Dear Mephista:
```

Now let's say the requirement for structure of the `person` dictionary changes: instead of name as a single string, we want given name and surname instead.

```
def send_email(p, message):
    print(f"Sending to {p['email']}: {message}")
    # ...more here

def extract_given_name(p):
    return p["name"].split()[0]

def format_contact(p):
    return f"{p['name']} <{p['email']}>"

def construct_greeting(p):
    return f"Dear {p['name'].split()[0]}:"
```

```
person = {
    "given_name": "Mephista",
    "surname": "Garply",
    "email": "mxgarply@uvm.edu"
}

# Test
send_email(person, "Hello there...")
print(extract_given_name(person))
print(format_contact(person))
print(construct_greeting(person))
```

If we run this code, the call to `send_email()` works OK, but `extract_given_name()` fails with a `KeyError`. There is no key `name` anymore. So let's fix that.

```
def send_email(p, message):
    print(f"Sending to {p['email']}: {message}")
    # ...more here

def extract_given_name(p):
    return p['given_name']

def format_contact(p):
    return f"{p['name']} <{p['email']}>"

def construct_greeting(p):
    return f"Dear {p['name'].split()[0]}:"

person = {
    "given_name": "Mephista",
    "surname": "Garply",
    "email": "mxgarply@uvm.edu"
}

# Test
send_email(person, "Hello there...")
print(extract_given_name(person))
print(format_contact(person))
print(construct_greeting(person))
```

Now we have another `KeyError` from `format_contact()`!

```
def send_email(p, message):
    print(f"Sending to {p['email']}: {message}")
    # ...more here

def extract_given_name(p):
    return p['given_name']
```

```
def format_contact(p):
    return f"{p['name']} <{p['email']}>"

def construct_greeting(p):
    return f"Dear {p['name'].split()[0]}:"

person = {
    "given_name": "Mephista",
    "surname": "Garply",
    "email": "mxgarply@uvm.edu"
}

# Test
send_email(person, "Hello there...")
print(extract_given_name(person))
print(format_contact(person))
print(construct_greeting(person))
```

Here's a fix.

```
def send_email(p, message):
    print(f"Sending to {p['email']}: {message}")
    # ...more here

def extract_given_name(p):
    return p['given_name']

def extract_full_name(p):
    return f"{p['given_name']} {p['surname']}"

def format_contact(p):
    return f"{extract_full_name(p)} <{p['email']}>"

def construct_greeting(p):
    return f"Dear {p['name'].split()[0]}:"

person = {
    "given_name": "Mephista",
    "surname": "Garply",
    "email": "mxgarply@uvm.edu"
}

# Test
send_email(person, "Hello there...")
print(extract_given_name(person))
print(format_contact(person))
print(construct_greeting(person))
```

Again we have a `KeyError`. This time it's from construct greeting. Here's a fix:

```
def send_email(p, message):
    print(f"Sending to {p['email']}: {message}")
    # ...more here

def extract_given_name(p):
    return p['given_name']

def extract_full_name(p):
    return f"{p['given_name']} {p['surname']}"

def format_contact(p):
    return f"{extract_full_name(p)} <{p['email']}>"

def construct_greeting(p):
    return f"Dear {extract_given_name(p)}:"

person = {
    "given_name": "Mephista",
    "surname": "Garply",
    "email": "mxgarply@uvm.edu"
}

# Test
send_email(person, "Hello there...")
print(extract_given_name(person))
print(format_contact(person))
print(construct_greeting(person))
```

If we run this code, all is well again:

```
Sending to mxgarply@uvm.edu: Hello there...
Mephista
Mephista Garply <mxgarply@uvm.edu>
Dear Mephista:
```

Many would say that the best approach would be to use *object-oriented programming* (OOP) and to construct a person class (new kind of object). For this, I might be inclined to use Python's `dataclass` to facilitate.

```
from dataclasses import dataclass

@dataclass
class Person:
    given_name: str
    surname: str
    email: str

    @property
```

```
    def full_name(self):
        return f"{self.given_name} {self.surname}"

    @property
    def greeting(self):
        return f"Dear {self.given_name}:"

    @property
    def contact(self):
        return f"{self.full_name} <{self.email}>"

def send_email(p, message):
    print(f"Sending to {p.contact}: {message}")

data = {
    "given_name": "Mephista",
    "surname": "Garply",
    "email": "mxgarply@uvm.edu"
}

person = Person(**data)

# Test
send_email(person, "Hello there...")
print(person.given_name)
print(person.contact)
print(person.greeting)
```

The lines starting with `@` are *decorators* which "wrap" a function to give it special features or functionality. When we run this, everything works OK.

```
Sending to Mephista Garply <mxgarply@uvm.edu>: Hello there...
Mephista
Mephista Garply <mxgarply@uvm.edu>
Dear Mephista
```

Constructing classes like this is one way of encapsulating information and providing a uniform interface for getting information out. The class itself is responsible for its own data. For more on classes and Python's `dataclass`, see:

- Classes[1]
- Dataclasses[2]

[1] https://docs.python.org/3/tutorial/classes.html
[2] https://docs.python.org/3/library/dataclasses.html

Appendix H

The call stack

Exploring the call stack

We introduced the concept of a stack in Chapter 11: Loops, iteration, and iterables. Here, we'll learn about the *call stack* and how calls to functions are handled.

The call stack

Python uses a stack to keep track of function calls. We've seen stacks before. They are LIFO (last-in, first-out) data structures.

Let's consider a single function call. Say we have this function:

```
def h(z):
    return z + 1
```

When we call `z()`, say with `z(1)`, Python creates a *stack frame* and pushes this frame onto the stack. This frame includes some crucial information:

- the frame's address,
- the address of the code that called the function (technically, the address of the previous stack frame),
- the *code object* currently being executed,
- a dictionary of local variables used,
- a dictionary of the global namespace, and
- other information useful for tracing and debugging.

The code object includes:

- its address,
- the raw bytecode produced when executing the body of the function called,
- names and values of arguments, and
- other information useful for tracing and debugging.

The top level of a module has its own stack frame. This stack frame has no previous caller and so it does not include the address of the calling code (there isn't any).

So if we have code like this:

```
def h(z):
    return z + 1

if __name__ == '__main__':
    result = h(1)
    print(result)  # prints 2
```

then we start with the top-level stack frame. Python pushes the frame for the function call to `h()` onto the call stack. Then `h()` does its work, and when it returns, the address of the calling code tells Python where to return to, and the frame for the call to `h()` is popped off the stack. (We'll ignore the call to `print()`.)

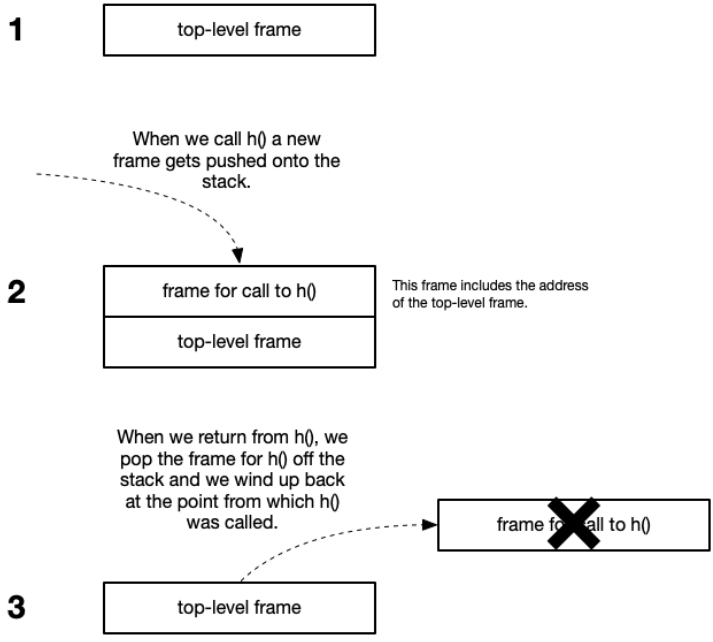


Figure H.1: Stack operations with a single function call

Now let's consider another case, where we have two functions, one which calls the other.

```
def g(y):
    return h(y) * 2

def h(z):
    return z + 1

if __name__ == '__main__':
    result = g(1)
    print(result)  # prints 4
```

We start with the top-level stack frame, then Python pushes the frame for the function call to `g()` onto the call stack. The function `g()` calls `h()` and uses the result of that call in another calculation. When `g()` calls `h()`, Python pushes a frame for the call to `h(x)` onto the stack. This includes the address of the frame for `g()` so Python knows where to return to. Then `h()` does its work, and when it returns, Python pops the frame for the call to `h()` off the stack and returns to the point at which it was called. Then `g()` does its work, and returns, and the stack frame for `g()` is popped off the stack, and we're back at the top level. (We'll ignore the call to `print()`.)

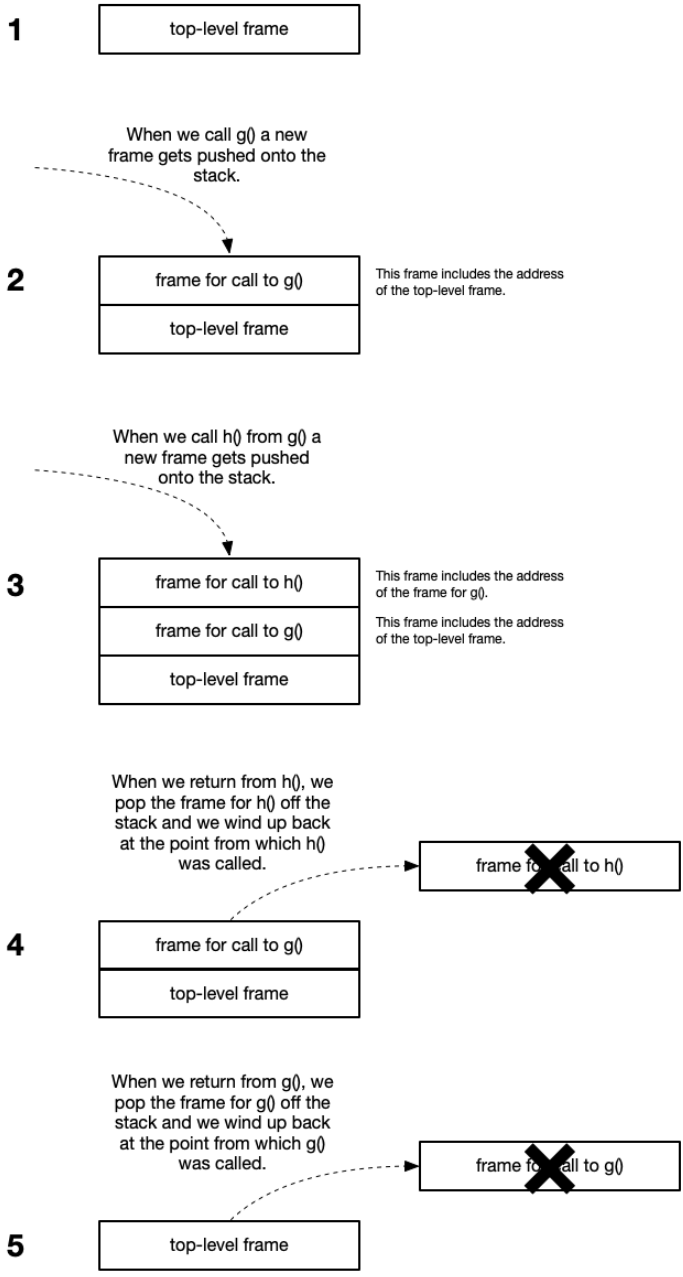


Figure H.2: Stack operations with nested function calls

If we had yet another function `f()` that's called at the top level, and `f()` calls `g()`, then when we get to the call to `h()` the depth of the stack would be four:

- frame for `h()` at top,
- frame for `g()`,
- frame for `f()`, and
- top-level frame.

As the function calls return, the frames are successively popped off the stack—pop, pop, pop—and we wind up back at the top level.

This is how Python keeps track of what's going on when a function is called.

Python provides us with the `inspect` module, which allows us to get information about the call stack, stack frames, and code objects (among other things).

Here's some code we can use to inspect and document nested function calls.

```
"""
Demonstration of call stack with nested function calls
"""
import inspect
import os


EXCLUSIONS = ('EXCLUSIONS', 'TAB', 'filter_vars',
              'pretty_code', 'pretty_frame', 'depth',
              'result')

TAB = '  '


def filter_vars(coll):
    """
    Because we're being intrusive---adding code to functions
    to demonstrate behavior of the stack---and because we
    have additional variables to support this, we need to
    filter variables reported by the inspect module. This
    allows us to focus on what gets passed to and used by
    f(), g(), and h(), and not have to face unnecessary
    clutter. """
    if isinstance(coll, dict):
        # Filter out dunders
        coll = {k: v for k, v in coll.items()
                if not (k.startswith('__')
                and k.endswith('__'))}
        coll = {k: v for k, v in coll.items()
                if '<module' not in repr(v)}
        coll = {k: v for k, v in coll.items()
                if k not in EXCLUSIONS}
```

```
        return coll
    elif isinstance(coll, tuple):
        # Filter out dunders
        coll = [name for name in coll
                if not (name.startswith('__')
                        and name.endswith('__'))]
        coll = [name for name in coll
                if not name.startswith('<module')]
        coll = [name for name in coll
                if name not in EXCLUSIONS]
        return coll
    else:
        raise TypeError(f'Expected dict or tuple, '
                        f'got {type(coll)}')


def pretty_code(code_obj):
    """
    From the Python documentation:

    https://docs.python.org/3/reference/datamodel.html

    Selected read-only attributes
    -----------------------------------------------------
    co_name:          The function name

    co_argcount:      The total number of positional parameters
                      (including positional-only parameters
                      and parameters with default values)
                      that the function has

    co_varnames:      A tuple containing the names of the
                      local variables in the function
                      (starting with the parameter names)

    co_code:          A string representing the sequence of
                      bytecode instructions in the function

    co_firstlineno:   The line number of the first line of
                      the function

    """
    if code_obj is None:
        raise TypeError("Expected a code object; got None.")

    depth = len(inspect.stack())
    print(f"{TAB * (depth - 1)}{'-' * 19} "
          f"CODE OBJECT {'-' * 19}")
    print(f"{TAB * (depth - 1)}Address: {hex(id(code_obj))}")
    print(f"{TAB * (depth - 1)}Function name: "
```

```
            f"{code_obj.co_name}")

    print(f"{TAB * (depth - 1)}Number of arguments: "
          f"{code_obj.co_argcount}")
    varnames = filter_vars(code_obj.co_varnames)
    # Don't use code_obj.co_nlocals; need to filter!
    print(f"{TAB * (depth - 1)}Number of local variables "
          f"used by the function: {len(varnames)}")
    if varnames:
        print(f"{TAB * (depth - 1)}Names of local variables: "
              f"{', '.join(str(x) for x in varnames)}")

    print(f"{TAB * (depth - 1)}Line number of the first "
          f"line of the function: "
          f"{code_obj.co_firstlineno}")
    print(f"{TAB * (depth - 1)}{'-' * 17} "
          f"END CODE OBJECT {'-' * 17}")


def pretty_frame(frm):
    """
    From the Python documentation:

    https://docs.python.org/3/reference/datamodel.html

    Selected read-only attributes
    -----------------------------------------------------
    f_back:   Points to the previous stack frame (towards
              the caller), or None

    f_code:   The code object being executed in this frame.

    f_locals: The mapping used by the frame to look up
              local variables.
    """
    if frm is None:
        raise TypeError("Expected a frame object; got None.")

    depth = len(inspect.stack())
    print(f"{TAB * (depth - 1)}{'-' * 22} FRAME {'-' * 22}")
    addr = hex(id(frm))
    print(f"{TAB * (depth - 1)}Address of this stack frame: "
          f"{addr}")
    addr = hex(id(frm.f_back)) if frm.f_back else 'None'
    print(f"{TAB * (depth - 1)}Address of calling frame: "
          f"{addr}")

    ls = filter_vars(frm.f_locals)
    if ls:
        print(f"{TAB * (depth - 1)}Dictionary of local "
```

```
                f"variables:")
        for k, v in ls.items():
            print(f"{TAB * depth}{k}: {v}")

    print(f"{TAB * (depth - 1)}Code object being executed "
          f"in this frame...")
    pretty_code(frm.f_code)
    print(f"{TAB * (depth - 1)}{'-' * 20} "
          f"END FRAME {'-' * 20}")
    print()


def f(x):
    depth = len(inspect.stack())
    print(f"{TAB * (depth - 1)}Size of stack: {depth}")
    pretty_frame(inspect.currentframe())
    print(f"{TAB * (depth - 1)}Call g()...")
    result = g(x) - 1
    print(f"{TAB * (depth - 1)}Returning result from "
          f"f({x}): {result}")
    print(f"{TAB * (depth - 1)}Size of stack: "
          f"{len(inspect.stack())}")
    return result


def g(y):
    depth = len(inspect.stack())
    print(f"{TAB * (depth - 1)}Size of stack: "
          f"{len(inspect.stack())}")
    pretty_frame(inspect.currentframe())
    print(f"{TAB * (depth - 1)}Call h()...")
    result = h(y) * 2
    print(f"{TAB * (depth - 1)}Returning result from "
          f"g({y}): {result}")
    print(f"{TAB * (depth - 1)}Size of stack: "
          f"{len(inspect.stack())}")
    return result


def h(z):
    depth = len(inspect.stack())
    print(f"{TAB * (depth - 1)}Size of stack: "
          f"{len(inspect.stack())}")
    pretty_frame(inspect.currentframe())
    result = z + 1
    print(f"{TAB * (depth - 1)}Returning result from "
          f"h({z}): {result}")
    print(f"{TAB * (depth - 1)}Size of stack: "
          f"{len(inspect.stack())}")
    return result
```

```
if __name__ == '__main__':
    print("Demonstration of stack frames with nested "
          "function calls...")
    print("We'll make a call to f(), but f() calls g(), "
          "and g() calls h().")
    print(f"Size of stack: {len(inspect.stack())}")
    pretty_frame(inspect.currentframe())
    print(f"Call f()...")
    print(f"result = {f(1)}")
    print(f"Size of stack: {len(inspect.stack())}")
    print("Done!")
```

When we run this code here's the result:

```
Demonstration of stack frames with nested function calls...
We'll make a call to f(), but f() calls g(), and g() calls h().
Size of stack: 1
  ---------------------- FRAME ----------------------
  Address of this stack frame: 0x104169a40
  Address of calling frame: None
  Dictionary of local variables:
    f: <function f at 0x117a5d900>
    g: <function g at 0x117a5d990>
    h: <function h at 0x117a5da20>
  Code object being executed in this frame...
    ------------------- CODE OBJECT -------------------
    Address: 0x117cd10b0
    Function name: <module>
    Number of arguments: 0
    Number of local variables used by the function: 0
    Line number of the first line of the function: 1
    ----------------- END CODE OBJECT -----------------
  -------------------- END FRAME --------------------

Call f()...
  Size of stack: 2
    ---------------------- FRAME ----------------------
    Address of this stack frame: 0x117a78580
    Address of calling frame: 0x104169a40
    Dictionary of local variables:
      x: 1
    Code object being executed in this frame...
      ------------------- CODE OBJECT -------------------
      Address: 0x117cd0df0
      Function name: f
      Number of arguments: 1
      Number of local variables used by the function: 1
      Names of local variables: x
```

```
      Line number of the first line of the function: 145
      ----------------- END CODE OBJECT -----------------
    -------------------- END FRAME --------------------

  Call g()...
    Size of stack: 3
      ---------------------- FRAME ----------------------
      Address of this stack frame: 0x117a78e40
      Address of calling frame: 0x117a78580
      Dictionary of local variables:
        y: 1
      Code object being executed in this frame...
        ------------------- CODE OBJECT -------------------
        Address: 0x117cd0ea0
        Function name: g
        Number of arguments: 1
        Number of local variables used by the function: 1
        Names of local variables: y
        Line number of the first line of the function: 158
        ----------------- END CODE OBJECT -----------------
      -------------------- END FRAME --------------------

    Call h()...
      Size of stack: 4
        ---------------------- FRAME ----------------------
        Address of this stack frame: 0x117a79000
        Address of calling frame: 0x117a78e40
        Dictionary of local variables:
          z: 1
        Code object being executed in this frame...
          ------------------- CODE OBJECT -------------------
          Address: 0x117cd1000
          Function name: h
          Number of arguments: 1
          Number of local variables used by the function: 1
          Names of local variables: z
          Line number of the first line of the function: 172
          ----------------- END CODE OBJECT -----------------
        -------------------- END FRAME --------------------

      Returning result from h(1): 2
      Size of stack: 4
    Returning result from g(1): 4
    Size of stack: 3
  Returning result from f(1): 3
  Size of stack: 2
result = 3
Size of stack: 1
Done!
```

Notice that the top-level stack frame has the address `0x104169a40`

(hexadecimal). When we call `f()`, the address of stack frame for `f()` is `0x117a78580`, but the address of the calling frame is `0x104169a40`—that's precisely the address of the top-level stack, from which `f()` was called.

When `f()` calls `g()`, another frame is pushed onto the stack. The address of this frame is `0x117a78e40`, and the address of the calling frame is `0x117a78580`—that's precisely the address of the stack frame for `f()`, from which `g()` was called.

When `g()` calls `h()`, another frame is pushed onto the stack. The address of this frame is `0x117a79000`, and the address of the calling frame is `0x117a78e40`—that's precisely the address of the stack frame for `g()`, from which `h()` was called.

Notice too that the value of the argument passed to `h()` is one. That's the original value that was passed to `f()` in the very first call. That value is passed all the way to this point, then `h()` does its work and returns the result to `g()`, `g()` does its work and returns the result to `f()`. Then `f()` does its work, and the result is returned to the top level. This process is referred to as *stack unwinding*.

So the stack serves to keep track of all the nested calls and provides context for each call. The "back" addresses included in each stack frame are like a trail of breadcrumbs, letting Python know where to go as the stack unwinds.

This mechanism is used in Python and other languages, so (I hope) you see how useful a stack can be!

Appendix I

The joy of Unicode

Here we explain a little bit about Unicode and why we may encounter `UnicodeDecodeError` or `UnicodeEncodeError` exceptions.

There are many different ways to encode letters and symbols. For example, let's go back to Chapter 2, where we saw how the letter `'A'` is encoded (see: Figure 3.1). There we saw that the letter `'A'` is encoded in binary as `01000001` which is numerically equivalent to decimal 65. We can confirm this in the shell:

```
>>> ord('A')
65
```

Here's what wasn't said in Chapter 2.

Python uses Unicode (UTF-8) encoding by default

Python, by default since 2008, uses what's called UTF-8 encoding. That's short for *Unicode Transformation Format—8-bit*, which is a mouthful. UTF-8 provides 1,112,064 different symbols. With Unicode we can write, display, and typeset symbols in different writing systems: alphabets, pictograms, mathematical symbols, musical notes, and even emoji. Unicode supports English, Chinese, Hindi, Arabic, Russian, and hundreds of other languages and writing systems. It even supports ancient writing systems with cuneiform and hieroglyphs. (For more, see: https://home.unicode.org/)

There's more than one encoding system

While much of the world runs on UTF-8 these days, it's not the only encoding system out there. Other systems in use include ISO-8859-1, Windows-1252, UTF-16, UTF-32, GB2312, Shift_JIS, GBK, EUC-KR, and Big5. Another term for encoding system is *code page* (think of a sheet of paper with all the characters represented in a given system along with their encoding in that system). Yikes!

So when I said the letter `'A'` is encoded as binary representation of decimal 65, that's true but more specifically `'A'` *is encoded as 65*

in the UTF-8 encoding system. Now, many encoding systems have the same values for the letters of the English alphabet. The reason for this is historical—designers of these encoding systems wanted to maintain backward compatibility with the ASCII (American Standard Code for Information Interchange) standard, first published in 1963. But in the case of many other symbols, there may not be agreement between systems in terms of how a given symbol can be encoded. Indeed, there may not even be an encoding for a particular symbol in one system which can be represented just fine in another.

Here's an example. Consider the string `"Henderson's Café"`. This contains a "curly" apostrophe, `'''`, and an accented symbol, the `'é'`. In UTF-8 these are encoded as 8217 and 233 (decimal), respectively We refer to these values as a *code points*. So how could we encode these in ASCII? We can't! There is no symbol `'''` in the ASCII code page. There is no symbol `'é'` in the ASCII code page. In fact, ASCII has only 128 code points in total whereas UTF-8 has 1,112,064. So the vast majority of symbols in UTF-8 can't be encoded in ASCII at all.

If we try to encode a symbol using a code page that does not support that symbol we get an encoding error.

```
>>> s = "Henderson's Café"
>>> s.encode('ascii')
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
    s.encode('ascii')
    ~~~~~~~~^^^^^^^^^^
UnicodeEncodeError: 'ascii' codec can't encode character
     '\u2019' in position 9: ordinal not in range(128)
```

Here it's choking on the curly quote (notice it says position 9 and Python starts counting at zero).

Hey, wait a minute, you might think: You said that the curly apostrophe is at code point 8217 in UTF-8 but the error message says `\u2019`. What's up with that? Let's try printing that:

```
>>> print('\u2019')
'
```

So this prints a curly apostrophe. But what is `\u2019`? Strings prefixed by `\u` are called Unicode strings, and the number that follows is the Unicode code point in *hexadecimal* notation. *Hexadecimal* is a base-16 number format, and it is quite common. So 2019 (hexadecimal) is equivalent to 8217 (decimal). Here's confirmation:

```
>>> int("2019", 16)   # interpret as base-16
8217
```

(By default, the `int` constructor interprets strings as base-10. Here we provide an additional argument, specifying the radix (base) to be used.)

Now let's take a look at a relatively modern encoding system Windows-1252 (which dates from the 1990s like UTF-8). While it's not as

prevalent as UTF-8 (for example, under 1.5% of all web pages worldwide are encoded in this system), it is still in use.

Here we'll encode the string `"Henderson's Café"` using two different encodings: UTF-8 and Windows-1252.

```
>>> s = "Henderson's Café"
>>> s.encode("utf-8")
b'Henderson\xe2\x80\x99s Caf\xc3\xa9'
>>> s.encode("cp1252")
b'Henderson\x92s Caf\xe9'
```

Notice the results—the *encodings*—are different. But, oh gosh, what now? What's up with the `b` and the `\x`?

Let's start with the `b`. Strings prefixed with `b` indicates that this string is to be interpreted as bytes. (This is a new type we've not seen before: `bytes` type.) What's a byte? Typically eight bits (binary digits). So if we have eight bits, we can represent numbers up to 255 because $2^8 = 256$, and we start at zero.

`\x` indicates that the following two digits are to be interpreted as hexadecimal. However, when encoding a symbol like `'’'` which has a Unicode code point of 2019 (hex) or equivalently 8217 (decimal) we need more than one byte (because a byte only goes up to 255). So we need a way of telling how many bytes we need and how to continue reading bytes as needed. (I know, I know, you're thinking "All this for a curly apostrophe?" Alas, yes.) For example, let's unpack `\xe2\x80\x99` within that bytestring. Each of these is a hexidecimal number: e2 80 99. Now let's look at these as binary 8-bit bytes:[1]

```
>>> f"{ord(b'\xe2'):08b}"  # format specifier for 8-bit binary
'11100010'
>>> f"{ord(b'\x80'):08b}"
'10000000'
>>> f"{ord(b'\x99'):08b}"
'10011001'
```

But not all of these bits are character bits. When representing multi-byte symbols there are prefixes that indicate how to interpret them.

Prefix	Meaning
0xxxxxxx	One-byte (7-bit) characters (ASCII)
110xxxxx	Start of 2-byte sequence
1110xxxx	Start of 3-byte sequence
11110xxx	Start of 4-byte sequence
10xxxxxx	Continuation of multi-byte sequence

[1] You may ask how is 'e2' a hexadecimal number? Hexidecimal adds the digits: a = 10, b = 11, c = 12, d = 13, e = 14, and f = 15. So 'e2' is $14 \times 16^1 + 2 \times 16^0 = 224 + 2 = 226$.

(I know, I know, you're thinking "Seriously? All this for a curly apostrophe?" Alas, yes.)

Let's go back to those three bitstrings. `\xe2\x80\x99` is equivalent to `11100010 10000000 10011001`. Notice that the first begins with `1110` that's the prefix indicating the start of a 3-byte sequence. The remaining character bits (sometimes called "payload bits") in that byte are `0010`. The second byte, `10000000`, starts with `10` indicating the continuation of a multi-byte sequence, and the remaining character bits are `000000`. The third byte, `10011001`, starts with `10`—another continuation, with remaining character bits `011001`. Now we take all those character bits and concatenate them to get `0010000000011001`. What is that in decimal?

```
>>> int('0010000000011001', 2)   # interpret as base-2
8217
```

Voilà! 8217 as expected. Now what is this in hexadecimal?

```
>>> hex(8217)   # hex() is the hex constructor
'0x2019'
```

Boom! There's the "2019" in `'\u2019`. That's the UTF-8 code point for `''`.

Now let's look at the same symbol with Windows-1252 encoding: just `\x92` which is decimal 146.

```
>>> int("92", 16)
146
```

So in the case of UTF-8, the curly apostrophe, `''`, is encoded with three bytes, whereas Windows-1252 encodes it with a single byte, with a very different value (at this point, you should see where this is heading).

```
>>> s = "Henderson's Café"
>>> utf_encoded = s.encode("utf-8")
>>> win1252_encoded = s.encode("cp1252")
```

Now let's try to decode the latter without specifying an encoding.

```
>>> win1252_encoded.decode()
Traceback (most recent call last):
  File "<python-input-26>", line 1, in <module>
    win1252_encoded.decode()
    ~~~~~~~~~~~~~~~~~~~~~~~^^
UnicodeDecodeError: 'utf-8' codec can't decode byte
    0x92 in position 9: invalid start byte
```

Why? Because the default encoding in Python is UTF-8, and the bytestring that we got from encoding in Windows-1252 isn't valid UTF-8. Now, you may ask, "Surely code point 146 is a valid code point in UTF-8. Why does this fail?" Great question. Let's look at hexadecimal

92 (binary 146) in binary: `10010010`. Now compare that with the prefixes in the table above. This starts with `10` which indicates continuation of a multi-byte string in UTF-8, but there was no start indicator for any multi-byte string! That's why this is invalid, and that's why we get a `UnicodeDecodeError`!

Seriously? Does it really work this way? Yup.

When do we encounter this kind of error? A common case is when trying to read from a file that was saved with a Windows-1252 without specifying the correct encoding. Here's proof:

```
with open("test.txt", 'w', encoding='cp1252') as fh:
    fh.write("Henderson's Café")

with open("test.txt") as fh:
    s = fh.read()
```

This fails with

```
Traceback (most recent call last):
  File "<string>", line 5, in <module>
  ...
UnicodeDecodeError: 'utf-8' codec can't decode byte
     0x92 in position 9: invalid start byte
```

That's one example.

How do we fix this? We specify the encoding used when opening the file for reading.

```
with open("test.txt", 'w', encoding='cp1252') as fh:
    fh.write("Henderson's Café")

with open("test.txt", encoding='cp1252') as fh:
    s = fh.read()

print(s)
```

This prints "Henderson's Café" with curly apostrophe and accented character as expected.

Appendix J

A brief introduction to databases with SQLite

We have seen how to read data from a file, either in unstructured text or CSV format, and how to write data in these formats. We've also seen how to read and write JSON. In this chapter we'll see how to use a database with Python.

There is a bewildering variety of databases, and a great many special-purpose databases, but the most widely used are *relational databases* and "NoSQL" databases. Common relational databases include Oracle, MySQL, PostgreSQL, MariaDB, Percona, and Microsoft SQL. Common NoSQL databases include MongoDB, Cassandra, Redis, and DynamoDB. In this appendix, we'll learn about a simple relational database that has native support built in to Python: SQLite.

SQLite (usually pronounced "sequel-ite") was first released in 2000 and has had native support in Python since 2006 through Python's `sqlite3` module.

The "SQL" you see in many of these names is short for *structured query language*. There are many dialects of SQL but they all share many common characteristics. SQLite is one such dialect, and true to its name, SQLite is a lightweight, minimal database solution.

This is not intended as a complete introduction to relational databases. We will not cover topics such as referential integrity, triggers and cascades, relational calculus, varieties of joins, database normalization, object-relational mappers (ORMs), database administration, or security. This will be a minimal introduction.

What is a "database"?

What exactly is a database? Usually what we mean by this (and certainly what we mean here) is a structured repository of data, organized into tables, managed by a *database management system* (DBMS). We don't edit the files of a database directly (and in some systems there can be a great many). Instead we let the DBMS do that for us, and we interact with the database through *queries* and commands specific to the particular system. Here we will use SQLite.

SQLite with Python

A relational database consists of one or more *tables*. Like a CSV table, a SQLite table has rows and columns. Unlike a CSV table, a SQLite table includes specification of data type, and (perhaps) other constraints. The structure of a database is called a *schema*.

When we use SQLite with Python, it's SQLite that's doing all the work. That's the database engine. Python provides us with a nice interface allowing us to write code in Python that "talks to" SQLite. This affords us many conveniences, but it doesn't eliminate the need to write our queries in SQL.[1] So this will take some getting used to. SQL is an entirely different language—unlike Python in almost every respect. To keep things straight: SQL code will appear as string arguments passed to functions provided by Python's SQLite interface, `sqlite3`. The surrounding code is Python. So you'll see SQL queries as arguments in Python function calls, but never the other way around.

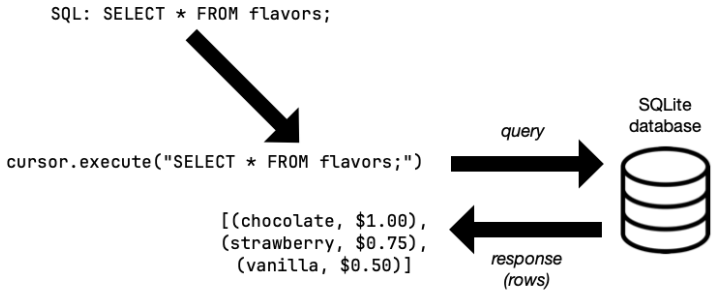


Figure J.1: SQL in Python function call to SQLite interface

Let's say we were creating a database of student clubs on campus. Here are a few (from UVM's clubs directory):

- African Student Association
- Agriculture Club
- Alianza de Latines
- American Sign Language Club
- Anthropology Club
- Archery Club
- Art Club
- Asian Student Union

Let's say we wanted to store the name of the club, the year the current by-laws were approved, and the current president. (I'm going to make up some names and numbers here.)

[1] There are object-relational mappers that abstract most if not all of database operations, like SQLAlchemy, and many others. Here we'll be writing good old-fashioned SQL.

Name	By-laws	President
African Student Association	2020	Amy Adeoye
Agriculture Club	2019	Hortense Greenthumb
Alianza de Latines	2021	Ernesto Ortiz
American Sign Language Club	2017	Marie Austyn
Anthropology Club	2022	Ken Korey
Archery Club	2023	Bill Tell
Art Club	2020	Julia Fish
Asian Student Union	2021	Michiko Itatani

If we wish to make a database table to represent such data, we'll need to specify the column names and the data type for each column. Clearly name and president are strings, and year is an integer.

So how do we create a database table? While a SQLite database consists of files on your computer's disk or solid-state drive, we don't create or edit these files directly—the *database engine* does this for us. So instead, we create a connection to a database, and then write a query to create the table for us. What we supply is a definition of the table, called a *table schema*.

We'll need two things to start, a *connection* to the database, and a *database cursor*.

```
>>> import sqlite3
>>> cnx = sqlite3.connect("clubs.db")
>>> cursor = cnx.cursor()
```

First we import `sqlite3`. This is a standard Python module, included with your Python distribution—there's nothing to install.

Then we create a connection, giving the name of the database `"clubs.db"`. The `.connect()` function opens a connection to the specified database. If the specified database does not exist, it will create the database for you. The `.db` file extension isn't strictly necessary but it's good practice to include it. Notice that we assigned the database connection to a variable named `cnx`. (`con`, `conn`, and `connection` are also commonly-used names for this.)

Then we get a *cursor* into the database. A database cursor is like a pointer into your database which is used to traverse the database, and to retrieve and manipulate records. Notice that we assigned the cursor object to a variable named `cursor`.

Now we're ready to create our table. Here's where we'll write in SQL. We'll write SQL to construct our table, and pass this to the cursor for execution. Notice that the script is a multi-line string. If we have multiple SQL statements, we separate them with semicolons. If we have a single statement, termination with a semicolon is optional. Like Python, SQL isn't particularly fussy about single- or double-quotes for strings. SQL keywords appear in UPPER CASE. Tokens in lower case (without quotation) are identifiers. Identifiers here are names of tables and names of columns within tables.

```
>>> cursor.execute("""
...     CREATE TABLE club (
...         name      TEXT     PRIMARY KEY,
...         year      INTEGER,
...         president TEXT
...     );
...     """
... )
...
<sqlite3.Cursor object at 0x1025b6f40>
>>> cnx.commit()
```

Let's walk through this. We'd previously created a cursor with `cursor = cnx.cursor()`. Here we passed this SQL to the cursor for execution:[2] The last bit is what's emitted at the prompt. When we call `cursor.execute()` it returns itself (that's what we see at the prompt; you may ignore this). The reason it returns itself, is to facilitate *method chaining*, but we won't futz with that here.

```
CREATE TABLE club (
    name       TEXT      PRIMARY KEY,
    year       INTEGER,
    president  TEXT
);
```

This was wrapped in triple double-quotes (for a multi-line string). This tells SQLite to create a table named `club` (it's considered good style to name tables with singular nouns). `name TEXT PRIMARY KEY` tells SQLite to add a column in the table, called `name`, and that this column will hold text (string) data. `PRIMARY KEY` tells SQLite that entries in this column must be unique (can't have two clubs with the same name). `KEY` has a different meaning in SQL than it does with Python dictionaries. `cnx.commit()` tells SQLite to commit the change to disk storage.

After this has been executed, we can check the resulting change in the database. Here we'll ask for information about the newly-created table.

```
>>> result = cursor.execute("PRAGMA table_info(club);")
>>> for column in result.fetchall():
...     print(column)
...
(0, 'name', 'TEXT', 0, None, 1)
(1, 'year', 'INTEGER', 0, None, 0)
(2, 'president', 'TEXT', 0, None, 0)
```

Again, there's a lot going on here, so let's take it in chunks.

[2] You'll notice this is formatted differently than Python code. That's what you'd expect. SQL is a different language, and the conventions of Python and PEP 8 do not apply to SQL. Examples here will follow good style for SQL. Spacing is intentional, and SQL keywords appear in ALL CAPS.

```
PRAGMA table_info(club);
```

`PRAGMA` commands (from "pragmatic") are SQLite-specific commands used to get information about the structure of the database or tables, or to inspect or alter configuration information. (They're generally not used in production.) Here we're using the `PRAGMA` command `table_info()` to get information about a particular table (`club`).

Notice that we've assigned the result to a variable named `result` (which is a cursor object). Then we call `.fetchall()` on this object to get records containing information about individual columns within the table, `club`. When we iterate over this, we get three tuples, each describing one of the three columns in the table. Each tuple has six elements:

- the column number (zero-indexed),
- the column name,
- the datatype of the column,
- whether or not a null (empty) value is permitted for the column,[3]
- a default value if any (in this case all `None`), and
- whether the column is a primary key (1 if a primary key, 0 otherwise).

Consider why each row is represented with a tuple. It should make sense to say that in a very real sense, a row in a database table *is* a tuple. Not a Python tuple, of course, but think about what makes a tuple a tuple, in the abstract sense. *A tuple is an ordered collection of objects.* You can't rearrange the elements of a tuple. You can't remove an element; you can't add an element. Database rows are like this.

Now that we've confirmed the table has been created, let's add some records. For this we use an `INSERT` statement and `cnx.commit()`.

```
>>> cursor.execute("""
...     INSERT INTO club (name, year, president)
...     VALUES ('African Student Association', 2020, 'Amy Adeoye'),
...            ('Agriculture Club', 2019, 'Hortense Greenthumb'),
...            ('Alianza de Latines', 2021, 'Ernesto Ortiz'),
...            ('American Sign Language Club', 2017, 'Marie Austyn'),
...            ('Anthropology Club', 2022, 'Ken Korey'),
...            ('Archery Club', 2023, 'Bill Tell'),
...            ('Art Club', 2020, 'Julia Fish'),
...            ('Asian Student Union', 2021, 'Michiko Itatani')
... """)
<sqlite3.Cursor object at 0x1025b6f40>
>>> cnx.commit()
```

[3]The astute reader might have trouble with the fourth element here—whether or not null values are permitted. Why? You'd be right to assume that a column that serves as a primary key cannot be null. You'd also be right to assume that SQLite would not permit us to add a row to this table with a null (empty) value for name. However, this reports the structure of the table as it was declared, and doesn't report anything implicit.

Again, there's a lot going on here, so let's take it step-by-step. `INSERT` and certain other operations involve *transactions*. Transactions give us **ACID**. ACID is an acronym for *atomicity*, *consistency*, *isolation*, and *durability*.

Atomicity is a guarantee that either an entire transaction succeeds or it doesn't. It's never the case that *part* of a transaction succeeds but another part does not. Imagine a bank transfer, for example a transfer from your Venmo account to your bank. This transaction has two parts: a debit (deduction) from your Venmo account, and a credit (deposit) to your bank account. *Somebody* would be unhappy if only one part of this transaction succeeded and the other failed. With an assurance of *atomicity*, this can't happen. Either the transfer succeeds, or it doesn't.

Consistency is a guarantee that before and after a transaction has taken place, the database is in a valid state. For example, if a certain constraint is required by our database, consistency ensures that this constraint is satisfied before and after each transaction. If we try to insert a record without the required values, the transaction fails.

Isolation is a guarantee that one transaction cannot affect another.

Durability is a guarantee that once a transaction has been committed, the data will persist in the database, no matter what—even if at some point the system crashes.

When we issue an `INSERT`, a transaction is opened automatically. To close the transaction, we call `.commit()`. SQLite has an auto-commit mode (and you can check this with the attribute `cnx.autocommit`), but it's good practice to call `.commit()` (but I may not explicitly use this in every example).

If there was even a single error in the `INSERT` statement above, the entire transaction would fail, and the database would be left unchanged in a valid state. If the transaction succeeds, we're guaranteed that all eight records representing clubs are saved.

Now, let's check the database. Databases wouldn't be very useful if we couldn't retrieve data from them. Here's a simple query to retrieve all records in the `club` table.

```
>>> result = cursor.execute("SELECT * FROM club")
```

In SQL, `*` means *all*, so `"SELECT * FROM club"` means select *all* rows in the table `club`. To access these records, we can use `.fetchall()`.

```
>>> result.fetchall()
[('African Student Association', 2020, 'Amy Adeoye'),
 ('Agriculture Club', 2019, 'Hortense Greenthumb'),
 ('Alianza de Latines', 2021, 'Ernesto Ortiz'),
 ('American Sign Language Club', 2017, 'Marie Austyn'),
 ('Anthropology Club', 2022, 'Ken Korey'),
 ('Archery Club', 2023, 'Bill Tell'),
 ('Art Club', 2020, 'Julia Fish'),
 ('Asian Student Union', 2021, 'Michiko Itatani')]
```

Whenever we get a row of data from SQLite, the default result is a tuple, as you see in the example above. Sometimes a tuple is fine, but

there are other options. Why consider other options? Well, for one, with a simple tuple, we can't refer to columns within a row by name, we have to use indexed reads to select values in specific columns.

Python's `sqlite3` allows us to specify a *row factory* that is used to construct objects retrieved by a query. Again, by default, rows are returned as tuples. Other options include `sqlite3.Row` objects, named tuples, dictionaries, Python dataclasses, or custom classes. We won't get deep into object-oriented programming here (OOP), but let's explore the other options so we can evaluate the pros and cons, and give you some idea of how we'd choose the right tool for a given job.

Generally, to choose a different representation of a row, we supply a *factory method* to the SQLite connection. SQLite then uses this factory method to construct output in the desired form. Factory methods are a common approach to constructing objects—they're used in all kinds of programming. A factory method constructs objects according to some type or specification. The first we'll investigate is `sqlite3.Row`.

`sqlite3.Row` will give us something akin to a dictionary, in which we can refer to columns in a row by their name. We can tell SQLite to use this factory thus:

```
>>> cnx.row_factory = sqlite3.Row
```

Notice what we're doing here. `sqlite3.Row` is the name of a function and by assigning this to `cnx.row_factory` we're telling the database connection which function to call when constructing rows. If we want our cursor object to use this, we must instantiate it *after* setting the row factory.

```
>>> cnx.row_factory = sqlite3.Row
>>> cursor = cnx.cursor()  # NOW cursor knows to use the factory
```

Now, when we get results of a query, they're of type `sqlite3.Row`, not plain-vanilla tuples—and these objects have keys for their fields.

```
>>> result = cursor.execute("SELECT * FROM club")
>>> for row in result.fetchall():
...     print(f"{row['president']} is the president of the "
...           f"{row['name']}.")
...
Amy Adeoye is the president of the African Student Association.
Hortense Greenthumb is the president of the Agriculture Club.
Ernesto Ortiz is the president of the Alianza de Latines.
Marie Austyn is the president of the American Sign Language Club.
Ken Korey is the president of the Anthropology Club.
Bill Tell is the president of the Archery Club.
Julia Fish is the president of the Art Club.
Michiko Itatani is the president of the Asian Student Union.
```

That's nicer than having to refer to the president column as `row[2]` and the name column as `row[0]`.

Again, there are other options. For example, this is a great case for *named tuples*, so let's see how we can used a named tuple here. For this, we'll need to supply our own factory method to the database connection and then instantiate a new cursor object. First, let's define a named tuple class for a club.

```
>>> from collections import named tuple
>>> Club = namedtuple('club', ['name', 'bylaw_year',
...                            'president'])
```

Now we need to define a factory method. Think about what would be needed: given a row as a tuple, construct a named tuple object from the row tuple. There's a specific interface we must use so that SQLite can use the factory. Our method *must* take two arguments: a `Cursor` object and the tuple of row values. Our method should return an object which represents the given row.

Now, you might think something as simple as this would be OK:

```
>>> def club_factory(cursor, row):
...     return Club(*row)
>>> cnx.row_factory = club_factory
```

and indeed this works as you'd expect.

```
>>> cursor = cnx.cursor()
>>> result = cursor.execute("SELECT * FROM club")
>>> for row in result.fetchall():
...    print(row)
...
club(name='African Student Association',
     bylaw_year=2020, president='Amy Adeoye')
club(name='Agriculture Club',
     bylaw_year=2019, president='Hortense Greenthumb')
club(name='Alianza de Latines',
     bylaw_year=2021, president='Ernesto Ortiz')
club(name='American Sign Language Club',
     bylaw_year=2017, president='Marie Austyn')
club(name='Anthropology Club',
     bylaw_year=2022, president='Ken Korey')
club(name='Archery Club',
     bylaw_year=2023, president='Bill Tell')
club(name='Art Club',
     bylaw_year=2020, president='Julia Fish')
club(name='Asian Student Union',
     bylaw_year=2021, president='Michiko Itatani')
```

Boom! How easy was that? Ah, but there's a "gotcha" here. *This solution only works for the* `club` *table.* If we were to query the database regarding table structure as we did with `PRAGMA table_info(club)`; this would fail! Why? Because this doesn't return rows of club data. Few

databases consist of a single table, and we'll get to adding another table to our database soon. We need a more general approach—one that will work with any table, any query result. Here we'll adapt a solution from the Python documentation (see: SQLite row factory[4]).

When we issue a query like: `cursor.execute("SELECT * FROM club")`, we get the selected rows, but we can also inspect the cursor to get the row names. For this we use `cursor.description`. For each field in the query `cursor.description` will hold a tuple, the first element of which is the field name (even if no rows are returned).

```
>>> result = cursor.execute("SELECT * FROM club")
>>> for item in cursor.description:
...     print(item[0])
...
name
year
president
```

This allows us to dynamically retrieve column names returned by any query. We can use this information to define a dynamic factory method for named tuples.

```
>>> def namedtuple_factory(cursor, row):
...     fields = []   # to hold the field names
...     for column in cursor.description:
...         fields.append(column[0])    # get the field names
...     # dynamically construct a named tuple class
...     cls = namedtuple("Row", fields)
...     # instantiate an object of this class
...     return cls(*row)  # use a splat, but see footnote
```

Now we can use this factory and it will work with *any* table or query result![5]

```
>>> cnx.row_factory = namedtuple_factory
>>> cursor = cnx.cursor()   # get a new cursor to use factory
```

Now instead of referring to fields like this `row['president']` we can use `row.president`.

```
>>> result = cursor.execute("SELECT * FROM club")
>>> for row in result.fetchall():
...     print(f"{row.president} is the president of the "
...           f"{row.name}.")
...
```

[4] https://docs.python.org/3/library/sqlite3.html#sqlite3-howto-row-factory

[5] There's a little better approach to this, which is slightly more robust, but this will suffice for the present purpose. For more, see the Python documentation on row factories.

```
Amy Adeoye is the president of the African Student Association.
Hortense Greenthumb is the president of the Agriculture Club.
Ernesto Ortiz is the president of the Alianza de Latines.
Marie Austyn is the president of the American Sign Language Club.
Ken Korey is the president of the Anthropology Club.
Bill Tell is the president of the Archery Club.
Julia Fish is the president of the Art Club.
Michiko Itatani is the president of the Asian Student Union.
```

If we wanted results as a dictionary we could construct a dictionary factory method. However, since named tuples have `._asdict()`, we can always construct a dictionary if needed from a given named tuple. We might do that if we wanted to update the values of fields (say, there's a new president or the bylaws have been updated). We'll leave construction of a dictionary factory method as an exercise for the reader.

Here's another relatively lightweight option that doesn't involve full-fledged OOP: Python `dataclass`. The `dataclass` type is available via the `dataclasses` module, which is part of the standard Python distribution (no installation needed).

```
>>> from dataclasses import dataclass
>>> @dataclass  # tell Python this is a dataclass definition
... class Club:
...     name: str
...     year: int
...     president: str
```

This creates a dataclass `Club`, with three fields: `name`, `year`, and `president`. Notice also that we specify the type of each field.

Unfortunately, creating a dynamic factory method that respects existing `dataclass` definitions would require more machinery than would be appropriate here (and isn't more performant), so when we want to work with `dataclass` objects, we'll construct what we need while iterating over query results. (We'll see more on `dataclass` soon.)

Now let's compare the possible ways we could construct objects to represent rows of data:

feature	`tuple`	`sqlite3.Row`	`namedtuple`	`datatype`
default	yes	no	no	no
index access	yes	yes	yes	no
key access	no	yes	no	no
named fields	no	no	yes	yes
immutable	yes	yes	yes	no
enforces type	no	no	no	yes
full control	no	no	no	yes

Here "index access" means we can access elements by index like `row[2]`; "key access" means we can access elements by key like this: `row['foo']`; "named fields" means we can access by field name like this: `row.foo`; and

"full control" means we can customize the behavior of objects and add custom methods (we'll see a little more about this later). If we want something fast and easy, tuples are fine, but access by index isn't ideal—especially if we have lots of columns. `sqlite3.Row` is also fast and easy, and gives us dictionary-like key access. `sqlite3.Row` is also robust to changes in schema or column orders. If we don't mind defining our own `namedtuple` class or `datatype` class and corresponding factory methods, then these are good choices.

Selecting records by criteria

It's often the case that we want to select specific records or even a single record based on certain criteria, and SQL allows us to do this. Let's say we wanted to select all the clubs with by-laws adopted before 2020. We use a `SELECT` query with a `WHERE` clause.

```
>>> result = cursor.execute("""
...     SELECT *
...       FROM club
...      WHERE year < 2020
... """)
>>> result.fetchall()
[Club(name='Agriculture Club', year=2019,
      president='Hortense Greenthumb'),
 Club(name='American Sign Language Club', year=2017,
      president='Marie Austyn')]
```

Notice something about SQL queries. We don't have to tell SQLite *how* to find the records, apart from specifying the table and criteria. We don't have to tell it to iterate over records in the table and to look in a particular column and perform a comparison, or anything like that. Instead, we tell SQLite what we want, and let SQLite figure out how to retrieve the appropriate records. This is what we call a *declarative* style of programming. In contrast, Python is an *imperative* language. To get the results we want we have to specify step-by-step instructions as to how to accomplish our goal. These are two very different *programming paradigms*.

Let's write a query that selects a single record. Since `name` is a primary key, we know that the value for `name` uniquely identifies a single row in our table.

```
>>> result = cursor.execute("""
...     SELECT *
...       FROM club
...      WHERE name = 'Anthropology Club'
... """)
>>> result.fetchone()
Club(name='Anthropology Club', year=2022, president='Ken Korey')
```

Again, let's unpack this. Here we have a `SELECT` query. The `WHERE` clause targets the single record with name `'Anthropology Club'`. Notice

that in the query = serves as a comparison operator (unlike Python's ==). Also, notice that since we know we'll have only one record, we can use `.fetchone()`.

You might ask: What happens if there is no matching record or records for some criteria? Good question. The query succeeds, but we get an empty result.

```
>>> result = cursor.execute("""
...     SELECT *
...       FROM club
...      WHERE name = 'Bongo Drumming Circle'
... """)
>>> result.fetchone()
>>>
```

or

```
>>> result = cursor.execute("""
...     SELECT *
...       FROM club
...      WHERE year > 3000
... """)
>>> result.fetchall()
[]
```

Adding a new table

Now let's create a table for club members and see how to join tables.

It wouldn't make sense for us to include all the members of a club in the `club` table, and it's not entirely clear how we'd do it anyway. A new table makes sense, because members of clubs (people) have different properties than clubs themselves. Let's create a table with some club members.

```
>>> cursor.execute("""
...     CREATE TABLE member (
...         club_name  TEXT,
...         given_name TEXT,
...         surname    TEXT,
...         email      TEXT,
...     );
... """)
...
>>> cnx.commit()
```

Now let's insert some records.

```
>>> cursor.execute("""
...     INSERT INTO member (club_name, given_name,
...                         surname, email)
...         VALUES ('Agriculture Club', 'Hortense',
...                 'Greenthumb', 'hgreenthumb@uvm.edu'),
...                ('Agriculture Club', 'Travis', 'Wilbury',
...                 'tqwilbury@uvm.edu'),
...                ('Agriculture Club', 'Richard', 'Mason',
...                 'rymason21@uvm.edu'),
...                ('Agriculture Club', 'Flora', 'Landis',
...                 'ftlandis@uvm.edu'),
...                ('Agriculture Club', 'Lillian', 'Valley',
...                 'lgvalley@uvm.edu')
... """)
... cnx.commit()
```

Let's check that all these records are now present.

```
>>> result = cursor.execute("SELECT * FROM member")
>>> for row in result.fetchall()
...    print(row)
...
Row(club_name='Agriculture Club', given_name='Hortense',
    surname='Greenthumb', email='hgreenthumb@uvm.edu')
Row(club_name='Agriculture Club', given_name='Travis',
    surname='Wilbury', email='tqwilbury@uvm.edu')
Row(club_name='Agriculture Club', given_name='Richard',
    surname='Mason', email='rymason21@uvm.edu')
Row(club_name='Agriculture Club', given_name='Flora',
    surname='Landis', email='ftlandis@uvm.edu')
Row(club_name='Agriculture Club', given_name='Lillian',
    surname='Valley', email='lgvalley@uvm.edu')
```

This looks OK, but notice that the president of the Agriculture Club appears in both tables. That's redundant and brittle. What should be done?

We could use some unique identifier (for example, email address) for the president of the club in the `club` table, or we could add a field to the member table to use as an indicator that a particular member is the president. Let's go with the second option.

To implement this, we need to *alter* both tables. We're going to remove the `president` column from the `club` table, and we're going to add a new column to the `member` table to hold the indicator of presidency.

For the first, we'll use `ALTER TABLE` and `DROP COLUMN` (drop means "delete").

```
>>> cursor.execute("""
...     ALTER TABLE club
...     DROP COLUMN president
... """)
...
>>> cnx.commit()
>>> result = cursor.execute("SELECT * FROM club")
>>> for row in result.fetchall():
...     print(row)
...
Row(name='African Student Association', year=2020)
Row(name='Agriculture Club', year=2019)
Row(name='Alianza de Latines', year=2021)
Row(name='American Sign Language Club', year=2017)
Row(name='Anthropology Club', year=2022)
Row(name='Archery Club', year=2023)
Row(name='Art Club', year=2020)
Row(name='Asian Student Union', year=2021)
```

Where did the president data go? It's gone forever! Accordingly, you should proceed with caution when dropping columns—this is a destructive operation.

Notice something else about this result. We're still using named tuples here, and despite having changed the table schema, our row factory method is working just fine. That's because we designed it to dynamically construct a (local) `namedtuple` class directly from query results. Neat!

Now let's use `ALTER TABLE` with `ADD COLUMN` to modify the `member` table. But first, let's ask, what datatype should we use for this column? Remember, the goal is to have an indicator as to which member of a given club is its president. You might think a Boolean would be appropriate, but SQLite doesn't have a Boolean datatype (other dialects of SQL do). So we could use a letter, *e.g.*, `'Y'`, `'N'`, or we could use an integer, `1` or `0`. Since the latter is closer in spirit to a Boolean let's go with that. However, we might want to constrain the values to `1` or `0` to prevent inserting a record with `5` (what on earth could that mean?). To emulate a Boolean we'll use SQLite's `CHECK` to add a column constraint. We'll also specify that this column should not be null and that it should have a default value of `0`.

```
>>> cursor.execute("""
...     ALTER TABLE member
...     ADD COLUMN president
...     INTEGER NOT NULL DEFAULT 0
...     CHECK(president IN (0,1))
... """)
...
>>> cnx.commit()
```

Now let's check to see what a record looks like.

```
>>> cursor.execute("""
...     SELECT *
...       FROM member
...      WHERE club_name = 'Agriculture Club'
... """)
...
>>> result.fetchone()
Row(club_name='Agriculture Club', given_name='Hortense',
    surname='Greenthumb', email='hgreenthumb@uvm.edu',
    president=0)
```

This looks OK, and notice that again, our dynamic named tuple factory handles the change automatically.

However, now we need to update Hortense Greenthumb's record to indicate she's the club president. For this we'll use an `UPDATE` query. For this kind of query, we need to specify the table we wish to update, the columns we wish to update with their new value(s), and the criteria used to select rows we wish to update. We want to update Hortense's record, setting `president = 1`, and leave other records unchanged.

```
>>> cursor.execute("""
...     UPDATE member
...        SET president = 1
...      WHERE email = 'hgreenthumb@uvm.edu'
... """)
>>> cnx.commit()
>>> result.fetchone()
Row(club_name='Agriculture Club', given_name='Hortense',
    surname='Greenthumb', email='hgreenthumb@uvm.edu',
    president=1)
```

This looks good.

You may notice we used Hortense's email to identify her record. You may also realize that email address could serve as a unique identifier for a person. We could have two people named Tim Smith at the university, but they'd each have their own email address. Why don't we set email address to be a unique key field in the member table? Because a person might be a member of more than one club! (We'll revisit this a little later.)

Let's add the other presidents for now, to restore information we deleted when dropping a column from the `club` table. Let's see a little different approach, assuming we have all the necessary tuples ready to insert:

```
>>> members = [
...     ('African Student Association', 'Amy', 'Adeoye',
...      'aaadeoye@uvm.edu', 1),
...     ('Alianza de Latines', 'Ernesto', 'Ortiz',
...      'eeortiz@uvm.edu', 1),
```

```
...     ('American Sign Language Club', 'Marie', 'Austyn',
...      'mmaustyn@uvm.edu', 1),
...     ('Anthropology Club', 'Ken', 'Korey',
...      'kakorey@uvm.edu', 1),
...     ('Archery Club', 'Bill', 'Tell',
...      'wmqtell@uvm.edu', 1),
...     ('Art Club', 'Julia', 'Fish',
...      'jafish@uvm.edu', 1),
...     ('Asian Student Union', 'Michiko', 'Itatani',
...      'michiko.itatani@uvm.edu', 1)
... ]
>>> cursor.executemany("""
...     INSERT INTO member (club_name, given_name,
...                         surname, email, president)
...         VALUES (?, ?, ?, ?, ?)
... """, members)
... cnx.commit()
```

This is a convenient way to insert multiple records. The question marks following `VALUES` are placeholders (called bindings) that are filled in by unpacking the values in each tuple. Notice that this uses `cursor.executemany()` and that we supply two arguments: the query with the appropriate number of bindings, and the data (here `members`).

Querying multiple tables with JOIN

Let's say we wanted to produce a report about clubs on campus. Something that looks like this:

Club	By-law year	Members	President
Agriculture Club	2019	5	Hortense Greenthumb
…	…	…	…

We have the by-law year in the `club` table, and the membership information is in the `member` table. So for something like this, we'd need to use a *join*. Joins are a way of connecting tables, so we can extract information in interesting and useful ways.

For this report, we'll need a join *and* a calculated column. The number of members in a club isn't explicitly stored in the database, but we can calculate this in a query.

Let's start by excluding the name of the president, then we'll add that later.

To retrieve the club name, by-law year, and count of members, we use `JOIN` to connect the two tables, `COUNT` to count rows in the member table, and `GROUP BY` to tell SQLite how to aggregate the count data.

```
>>> result = cursor.execute("""
...    SELECT club.name,
...           club.year,
...           COUNT(member.email) AS member_count
...      FROM club
...     JOIN member
...       ON club.name = member.club_name
... GROUP BY club.name
... """)
...
>>> for row in result.fetchall():
...     print(row)
...
Row(name='African Student Association', year=2020, member_count=1)
Row(name='Agriculture Club', year=2019, member_count=5)
Row(name='Alianza de Latines', year=2021, member_count=1)
Row(name='American Sign Language Club', year=2017, member_count=1)
Row(name='Anthropology Club', year=2022, member_count=1)
Row(name='Archery Club', year=2023, member_count=1)
Row(name='Art Club', year=2020, member_count=1)
Row(name='Asian Student Union', year=2021, member_count=1)
```

Of course, we haven't fully populated our database so the member count for every club other than the Agriculture Club is one. Nevertheless, you can see we've retrieved the club name, by-law year, and the count of members by club. Let's unpack that query, because there's a lot there.

First, notice that in our `SELECT` statement, we're specifying the columns we wish to retrieve from the `club` table. Then we're using `COUNT` to get the count of unique email addresses in the `member` table. There's no corresponding column for this, and every column in the result must have a name, so we give this the name `member_count` using `AS`.[6] But how do we connect these two tables, and how does SQLite know how to aggregate the member count by club? That's where `JOIN ON` and `GROUP BY` come in.

```
JOIN member
  ON club.name = member.club_name
```

`club` has a column, `name`, which holds the unique name for each club. Every `member` record has column `club_name` which indicates in which club a person is a member. `JOIN member ON club.name = member.club_name` tells SQLite to use common values in `club.name` and `member.club_name` to join the tables. For this to work, the club names in each table have to agree (but we've set it up that way). The last bit

```
GROUP BY club.name
```

tells SQLite how we want to aggregate records for purposes of counting—that we want to aggregate by club name rather than by some other

[6] This is not unlike giving a name to a file handle in Python.

column (we could aggregate by by-law year, for example, but that's not what we want).

Now let's see how to add the club president to the report.

```
>>> result = cursor.execute("""
...     SELECT club.name,
...            club.year,
...            COUNT(member.email) AS member_count,
...            p.given_name || ' ' || p.surname AS president
...       FROM club
...       JOIN member
...         ON club.name = member.club_name
...       JOIN member
...         AS p
...         ON club.name = p.club_name
...        AND p.president = 1
...      GROUP BY club.name
...      ORDER BY club.name ASC
... """)
...
```

Here we're joining twice: once to get the count of members, and once to get the president's name. Notice that the second join is conditioned on equivalence of club name and the correct value (`1`) for the president indicator. Like Python, SQLite has `AND` and `OR` Boolean connectives.

The last clause

```
ORDER BY club.name ASC
```

sorts the resulting rows alphabetically by club name. `ASC` here means *in ascending order*.

The strange looking bit is this:

```
p.given_name || ' ' || p.surname AS president
```

The `||` is SQLite's string concatenation operator, and it works just like Python's `+` would if both operands were strings. So this bit constructs the president's full name by concatenating their given name with their surname (with a space in-between).

Then, if we wanted to pretty-print a table we could do it like this:

```
print(f"{'Club':<28} {'Year':>5} "
      f"{'Members':>8} {'President':<20}")
print(f"{'-' * 28:<28} {'-' * 5:>5} "
      f"{'-' * 8:>8} {'-' * 20:<20}")
for row in result.fetchall():
    print(f"{row.name:<28} {row.year:>5} "
          f"{row.member_count:>8} {row.president:<20}")
```

and we'd get a nice table like this:

```
Club                          Year  Members President
----------------------------- ----- -------- --------------------
African Student Association   2020        1 Amy Adeoye
Agriculture Club              2019        5 Hortense Greenthumb
Alianza de Latines            2021        1 Ernesto Ortiz
American Sign Language Club   2017        1 Marie Austyn
Anthropology Club             2022        1 Ken Korey
Archery Club                  2023        1 Bill Tell
Art Club                      2020        1 Julia Fish
Asian Student Union           2021        1 Michiko Itatani
```

There's something important to notice in all this. The named tuple factory method continues to work (without alteration!) for this query as well as the queries made against a single table. But wait, you say: the named tuple factory gets column names from a table, but there is no table with these particular columns in this particular order. Well, actually *there is a table*. The SQLite database engine constructs this table in memory based on the query, so there is indeed a table here, just not one that persists (is saved to disk). *All* query results are a table.

There's still a little more we could do to tighten things up a bit.

Consider this: We have a unique key in our `club` table. This prevents us from creating a duplicate record for a given club. The name of the club must be unique and that's enforced by SQLite. However, there is no such key in the members table, and this would allow us to create two records for the same student if they were a member in more than one club. This is wasteful and error-prone. But what would be a good unique key for a student? As mentioned earlier, while two students could have the same name, two students cannot have the same email address. However, if we were to make email address a unique key in the `member` table, this would preclude our being able to have one student be a member of more than one club. What should we do?

We'll take a common approach. We'll alter the `member` table, to become a general-purpose `student` table with a unique key—email address. If we do this, we'll need a new way to represent membership—one which allows us to have a single student be in more than one club. The way we'll accomplish this is with an *association table*. When we're done, we'll have the `club` table (unchanged), a `student` table made by altering the `member` table, and an *association table*, `membership`, which links the two. This is a *very* common pattern in database design.

Let's create the `membership` table first. We can do this without a whole lot of work, because we can construct the data we need from the current tables. First, we'll create the table:

```
>>> cursor.execute("""
...     CREATE TABLE membership (
...         club_name  TEXT       NOT NULL,
...         email      TEXT       NOT NULL,
...         joined     INTEGER,
...         president  INTEGER    NOT NULL   DEFAULT 0
...         CHECK(president IN (0,1))
...     );
... """)
>>> cnx.commit()
```

Now, let's populate this with the appropriate data. We already have all the data in question, so we'll write one query to fetch what we need to populate this table, and then another to insert the data into the new table.

```
>>> result = cursor.execute("""
...     SELECT club_name, email, null AS joined, president
...       FROM member
... """)
...
>>> data = result.fetchall()
>>> for row in data:
...    print(row)
...
```

This is what we get.

```
Row(club_name='Agriculture Club',
    email='hgreenthumb@uvm.edu',
    joined=None, president=1)
Row(club_name='Agriculture Club',
    email='tqwilbury@uvm.edu',
    joined=None, president=0)
Row(club_name='Agriculture Club',
    email='rymason21@uvm.edu',
    joined=None, president=0)
Row(club_name='Agriculture Club',
    email='ftlandis@uvm.edu',
    joined=None, president=0)
Row(club_name='Agriculture Club',
    email='lgvalley@uvm.edu',
    joined=None, president=0)
```

…and so on.

Notice that we've saved this result, assigning it to the variable name `data`. We're going to use this in our insert query.

```
>>> cursor.executemany("""
...     INSERT INTO membership (club_name, email,
...                             joined, president)
...     VALUES (?, ?, ?, ?)
... """, data)
...
>>> cnx.commit()
```

That's it! Now our membership table is populated! (We could have written this as a single query, but I think it's easier to read if we break it into two queries.)

Now that we've constructed our `membership` table, we can alter the `member` table without losing any data.

```
>>> cursor.execute("ALTER TABLE member RENAME TO student")
>>> cnx.commit()
>>> cursor.execute("ALTER TABLE student DROP COLUMN club_name")
>>> cnx.commit()
>>> cursor.execute("ALTER TABLE student DROP COLUMN president")
>>> cnx.commit()
```

That completes our schema migration. Notice that `club` no longer stores email addresses, and that `student` (formerly `member`) no longer stores club information. This is good *separation of concerns* in database design.

All that's left is to rewrite the report query.

```
>>> result = cursor.execute("""
...     SELECT club.name,
...            club.year,
...            COUNT(student.email) AS member_count,
...            s.given_name || ' ' || s.surname AS president
...       FROM club
...       JOIN membership
...         ON club.name = membership.club_name
...       JOIN student
...         ON membership.email = student.email
...       JOIN membership AS mm
...         ON club.name = mm.club_name AND mm.president = 1
...       JOIN student AS s
...         ON mm.email = s.email
...   GROUP BY club.name
...   ORDER BY club.name ASC
""")
```

If we issue this query and print out the result as before we get the same result.

```
Club                          Year  Members President
----------------------------- ----- -------- ------------------------
African Student Association   2020        1 Amy Adeoye
Agriculture Club              2019        5 Hortense Greenthumb
Alianza de Latines            2021        1 Ernesto Ortiz
American Sign Language Club   2017        1 Marie Austyn
Anthropology Club             2022        1 Ken Korey
Archery Club                  2023        1 Bill Tell
Art Club                      2020        2 Julia Fish
Asian Student Union           2021        1 Michiko Itatani
```

Because we've nicely separated concerns, we have no duplication of data if we have a student or students who are members of more than one club. We can also add columns to `club` that are specific to clubs, *e.g*, website URLs, meeting pattern, *etc.*, and columns that are specific to students to the `student` table, *e.g.*, class (graduation year), home address, *etc.* We could also add queries to get a roster of members for a given club, or to display all the clubs of which a given student is a member. These are left as exercises for the reader.

A quick look at `dataclass`

Python's `dataclass` is widely used with databases, and it sits quite comfortably between named tuples and full-fledged OOP in complexity. Python makes it easy to define classes for your data without a whole lot of coding overhead (called "boilerplate").

Let's make a dataclass for student data.

```
>>> from dataclasses import dataclass
>>> @dataclass  # tell Python this is a dataclass definition
... class Student:
...     given_name: str
...     surname: str
...     email: str
```

That's a minimal example, and this would work with rows returned from our `student` table. Let's expand on this just a little so that we can get a feel for things we can do with such classes.

```
>>> from dataclasses import dataclass
>>> @dataclass  # tell Python this is a dataclass definition
... class Student:
...     given_name: str
...     surname: str
...     email: str

...     @property
...     def full_name(self):
...         return f"{self.given_name} {self.surname}"
```

```
...     @property
...     def contact(self):
...         return f"{self.full_name} <{self.email}>"
```

`@property` is a *decorator* which turns a method into a property field. In doing so, we make each property `full_name` and `contact` accessible without having to provide an empty parenthesised parameter list. So if we have an object of this type `student` we can access the student's full name (given name and surname) with `student.full_name` rather than the awkward `student.full_name()`.

Let's take it out for a test drive.

```
>>> result = cursor.execute("""
...     SELECT * FROM student
...     ORDER BY surname ASC")
>>> students = []
>>> for student in result.fetchall():
...     s = Student(*student)
...     print(s.contact)
...     students.append(s)
...
Amy Adeoye <aaadeoye@uvm.edu>
Marie Austyn <mmaustyn@uvm.edu>
Julia Fish <jafish@uvm.edu>
Hortense Greenthumb <hgreenthumb@uvm.edu>
Michiko Itatani <michiko.itatani@uvm.edu>
Ken Korey <kakorey@uvm.edu>
Flora Landis <ftlandis@uvm.edu>
Richard Mason <rymason21@uvm.edu>
Ernesto Ortiz <eeortiz@uvm.edu>
Bill Tell <wmqtell@uvm.edu>
Roxie Tremonto <rrtremonto@uvm.edu>
Lillian Valley <lgvalley@uvm.edu>
Travis Wilbury <tqwilbury@uvm.edu>
```

How cool is that? Notice also that we saved the objects created from retrieved records in a list called `students` for future use. We did it this way because we did not use this `datatype`-based class in the row factory—we constructed the objects after the query was returned.

This only hints at the full capabilities of dataclasses.

We won't get into it here, but you should be aware that SQLite has some support for JSON. This comes in handy, for example, when you have an application that exchanges data with a web API (application program interface).

By no means is this a complete introduction to SQLite with Python, but it should serve to get you started. We've covered:

- database connections and cursors,
- table creation,
- altering tables,
- inserting data,

- queries on single tables,
- queries on joined tables,
- row factory methods, including
  - `sqlite3.Row`, and
  - named tuples,
- `dataclass` objects and properties, and
- association tables.

Resources

- Python's documentation for the `sqlite3` module[7]
- SQLite website (with documentation)[8]
- SQL Style Guide[9]

[7] https://docs.python.org/3/library/sqlite3.html
[8] https://sqlite.org/
[9] https://www.sqlstyle.guide/

Index

__name__, 163

absolute value, 55, 379
accumulator, 254, 256, 379
adjacency list, 372
alternating sum, 256, 379
ambiguity, 20
anonymous variable, 250, 252
anti-pattern, 257
API (application program interface), 289, 334, 485
argument, 77, 79, 380
    keyword, 293, 396
    positional, 293, 380
arithmetic mean, 255, 380
arithmetic sequence, 247, 381
ASCII, 458
assembly language, 12
assertion, 182, 185, 381
assignment, *see* statement, 381
augmented assignment, 54

BDFL (benevolent dictator for life), 8
bell curve, 281, 305
binary arithmetic, 24
binary code, 12, 381
binary numbers, 21
bitstring, 33
Boole, George, 30
Boolean connective, 138, 382
Boolean expressions, *see* expressions
branch, 144
branching, 382
bug, 180, 383
built-in functions
    abs(), 243, 379
    bool(), 147
    chr(), 142
    enumerate(), 257, 258
    exit(), 18
    float(), 119, 385

input(), 116
int(), 119, 385
len(), 220
list(), 207, 231, 385
max(), 220
min(), 220
next(), 305
open(), 290, 291
print(), 18, 80
range(), 247, 269
set(), 345, 346
sorted(), 218, 220
str(), 123
sum(), 220, 254, 380
tuple(), 385
bytecode, 14, 383, 445

call stack, 445, 449
camel case, 383
central tendency, 383
Chomsky, Noam, 20
class, 333, 352, 355
CLI (command line interface), 116, 384
code page, 457
code point, 406, 458
code smell, 431
duplicated code, 431
long function, 431, 433
magic numbers, 431, 438
shotgun surgery, 431, 439
collection, 351
comments, 108, 384
comments as scaffolding, 110
comparable, 384
compilation, 14
compiler, 14, 384
compound statement, 146, 327
concatenation, 121, 384
concatenation, implicit, 111
condition, 144, 385, 406
congruent, 60, 385
conjunction, 139
console, 116, 385
constants, 48, 107
constructor, 247, 352, 385
container datatypes, 351
context manager, 290, 386
CSV (comma separated values), 295, 298, 383
CSV reader, 298
CSV writer, 298

database

ACID, 468
association table, 481
bindings, 478
constraint, 476
cursor, 465, 466
join, 478
management system (DBMS), 463
NoSQL, 463
relational, 463
schema, 464, 473
schema migration, 483
SQL (structured query language), 463
table schema, 465
transaction, 468
dataclass, 443, 444, 484
De Morgan's Laws, 140
decimal system, 22
decoding, 356
decorator, 444, 485
decrement, 54
delimiter, 29, 35, 386
determinism, 282
deterministic, 276
diagrams
decision tree, 154
flow chart, 391, 421
dictionary
key, 386
view objects, 339
dictionary methods
.items(), 340
.keys(), 340
.pop(), 342
.values(), 340
dictionary, see: types, dict, 335
Dijkstra, Edsger, 181
dividend, 54, 387
divisor, 54, 387
docstring, 108, 387
driver code, 163, 387
DRY (don't repeat yourself), 433
duck typing, 408
dunder, 167, 387
dynamic typing, 388

EAFP (easier to ask forgivness…), 329, 330
edge, 388
empty list, 31, 191
empty sequence, 388
empty set, 350
empty string, 30
empty tuple, 30

encapsulation, 333, 444
encoding, 356
entry point, 388
escape sequence, 36, 37, 389
Euclid's algorithm, 244, 245
Euclidean division, 54, 389
evaluation, 46, 389
exception handling, 390
exceptions, 68, 389
    AssertionError, 183, 185
    AttributeError, 323
    FileNotFoundError, 299, 326
    IndentationError, 98, 323
    IndexError, 192, 324
    JSONDecodeError, 327, 359, 366
    KeyError, 336, 345, 364
    ModuleNotFoundError, 99
    NameError, 69, 323
    SyntaxError, 68, 322
    TypeError, 70, 325, 365
    UnboundLocalError, 99, 156, 324
    UnicodeDecodeError, 299, 327, 457
    UnicodeEncodeError, 457
    ValueError, 98, 133, 326
    ZeroDivisionError, 56, 70, 326
expression, 49, 390
    Boolean, 138, 140, 144, 382

f-string, 124, 125, 392
factorial, 262
factory method, 469
Fibonacci sequence, 272, 390
field, 351
FIFO (first-in, first out), 267
file object methods
    .read(), 290
    .write(), 293
floating-point number, 29, 391
floor division, 54, 391
floor function, 58, 391
format specifier, 125, 128, 391
free variable, 95, 97, 392
functional decomposition, 168
functions, 76, 392
    call or invoke, 76, 383, 395
    defining, 77
    formal parameters, 77, 400
    impure function, 394
    local variables, 79
    pass by assignment, 88
    pure functions, 400
    return values, 77

returning None, 80

garbage collection, 224
GCD (greatest common divisor), 243
graph, 371, 392
acyclic, 373
adjacent, 372
breadth-first search, 373
cycle, 373
cyclic, 373
edge, 371
neighbor, 372
vertex, 371
GUI (graphical user interface), 116, 289, 392

hashable, 338, 342, 345, 386, 392
haversine formula, 354
Hello World, 18
heterogeneous, 30, 31, 393
hexadecimal, 458
Hopper, Grace Murray, 180

IDE (integrated development environment), 395
identifier, 44, 393
IDLE (integrated development and learning environment), 395
IEEE 754 (standard for floating-point representation), 39
immutable, 30, 393
import, 393
increment, 54
incremental development, 394
index, 191, 256, 394
information hiding, 87
input validation, 240
input/output (I/O), 394
instance, 394
instantiate, 298, 394
interactive mode, 2, 16, 395
interpretation, 14
interpreter, *see* Python interpreter
ISO 4217 (standard currency codes), 130
ISO-8859-1 (code page), 457
iterable, 233, 246, 395
iteration, 233
iterator, 299

JSON (JavaScript object notation), 333, 356, 361, 485
json functions
.dump(), 359
.dumps(), 357
.load(), 359
.loads(), 358

keyword argument, 298, 364
keywords, 77, 396
    and, 138, 139, 382
    as, 290
    assert, 182, 183, 254, 381
    break, 241, 259
    def, 77
    del, 342
    elif, 382
    else, 382
    except, 327
    False, 30
    for, 234, 245
    if, 382
    import, 393
    in, 339
    is, 149
    None, 30
    not, 138, 382
    or, 138, 139, 382
    return, 77
    True, 30
    try, 327
    while, 234–237, 239–241, 243
    with, 290, 386
kwarg splat, 364

law of the excluded middle (logic), 138
LBYL (look before you leap), 329
LEGB (local, enclosing, global, built-in), 96, 98
lexicographic order, 142, 396
LIFO (last-in, first-out), 265
line continuation, 131
line continuation, implicit, 112
list methods, 194
    .append(), 194, 220
    .copy(), 196, 206, 231
    .extend(), 252
    .pop(), 194, 220
    .sort(), 194, 220
literal, 28, 49, 397
local variable, 88, 397
loop, 233, 397
    break, 241
    for, 233, 246
    nested, 263
    while, 233, 235, 237, 239, 243
loops
    while, 240

math module, 91
    .ceil(), 93

.cos(), 93
.degrees(), 93
.e, 94
.exp(), 94
.floor(), 93
.log() (natural logarithm), 94
.log10(), 94
.log2(), 94
.pi, 94
.radians(), 93
.sin(), 92
.sqrt() (square root), 92
Matplotlib, 398
mean, 303–305, 308, 309
Mersenne twister, 283
method, 398
method chaining, 466
module, 398
modulus, 60, 398
Monte Carlo method, 276, 398
Monty Python's Flying Circus, 8
mutable, 30, 190, 399

named tuple, 333, 351, 355, 362, 470
names, 46
namespace, 399, 445
naming
camel case, 106
snake case, 106
word caps, 106
natural numbers, 38
normal distribution, 281, 305, 308

object, 27, 399
object-oriented programming (OOP), 333, 353
operands, 50
operators, 50, 399
addition, 50
assignment operator, 44, 381
binary operators, 50
comparison operators, 141
concatenation (+), 53
division, 50
exponentiation, 50, 66
floor division, 50, 56, 391
modulo, 50
multiplication, 50
overloading, 399
PEMDAS, 52
precedence, 51, 52
remainder, 56
repeated concatenation, 53

subtraction, 50
unary negation, 52
outer scope, 89

Parnas, David, 87
PEP 8, 104, 400
population, 280
product, 256
programming paradigm
declarative, 473
imperative, 473
proposition (logic), 138
pseudo-random, 276, 282, 400
pseudocode, 110
Python interpreter, 15
Python shell, 16
Python virtual machine, 383
Python-supplied modules
collections, 351
csv, 384
dataclass, 472
dataclasses, 443, 444
json, 357
math, 91
random, 276
sqlite3, 463
statistics, 309

quantile, 304, 309, 400
queue, 267, 268
quotient, 54, 401

random module, 276
.choice(), 277
.gauss(), 281
.randint(), 279
.random(), 278
.sample(), 280
.seed(), 283
.shuffle(), 279
random walk, 277, 401
rational numbers, 38
real numbers, 38
refactoring, 431
remainder, 54, 401
REPL (read-evaluate-print loop), 17, 401
replacement fields, 124
representation error, 37, 39, 402
return value, 402
rubberducking, 402
run time, 402

sample, 280
scope, 78, 88, 95, 96, 224, 402
script mode, 2, 18, 402
seed, 283, 403
semantics, 20, 403
separation of concerns, 434, 483
sequence methods
    .count(), 209, 211
    .index(), 209, 220
set, 333, 345
set intersection, 348
set methods
    .add(), 347
    .difference(), 348
    .discard(), 347
    .intersection(), 348
    .issubset(), 348
    .pop(), 347
    .remove(), 347
    .union(), 348
set symmetric difference, 350
set union, 348
shadowing, 89, 404
side effect, 404
slice, 221, 405
snake case, 405
splat (starred expression), 363
SQL
    add column, 476
    alter table, 475, 476, 483
    and, 480
    as, 480, 482, 483
    asc (ascending), 480, 483, 485
    check, 476
    count, 478, 479
    create table, 466, 474, 481
    default, 476, 481
    drop column, 475, 483
    group by, 478–480, 483
    in, 481
    insert, 468
    insert into, 467, 468, 474, 478, 482
    integer, 466, 476, 481
    not null, 476, 481
    or, 480
    order by, 480, 483, 485
    primary key, 466
    rename to, 483
    select … from, 468, 470, 472, 473, 477–480, 482, 483, 485
    text, 466, 474, 481
    update … set, 477
    values, 467, 474, 478, 482

where, 473, 477
sqlite3 classes
Row, 469, 473
sqlite3 functions
.commit(), 466–468
.connect(), 465
.cursor(), 465, 466, 470
.execute(), 466, 470, 471
.executemany(), 478
.fetchall(), 467, 468, 470
.fetchone(), 473, 474
sqlite3 properties
.autocommit, 468
.description, 471
.row_factory, 469
stack, 265, 267
stack frame, 445, 449
stack unwinding, 455
standard deviation, 303, 306–309, 405
statement, 405
assignment, 44, 45, 406
compound, 146
static typing, 406
statistics functions
.mean(), 309
.pstdev(), 309
.quantiles(), 310
stride, 222, 249, 406
string interpolation, 124, 395, 406
string methods, 150
.capitalize(), 151
.join(), 216, 219, 220
.lower(), 151
.replace(), 217, 220
.split(), 215, 216, 220
.strip(), 291, 295
.upper(), 151
string, see: types, str, 29
style, 104
line length, 107
subscripts as indices, 255
subset, 348
subset, strict, 348
summation, 255, 406
syntax, 19, 407

terminal, 407
top level, 446, 449
top-level code environment, 163, 407
trace (loop), 259
traceback, 68
truth table, 138

truth value, 138, 407
truthy and falsey, 146, 382, 390, 407
try/except, 327
type inference, 408
types, 28, 49, 408
    _csv.reader, 298
    _csv.writer, 298
    _io.TextIOWrapper, 290
    bool, 30
    byte, 459
    dataclass, 443, 444
    dict, 31, 333, 335, 363, 386, 393
    dict_items, 340
    dict_keys, 340
    dict_values, 340
    dynamic typing, 31, 45
    enumerate, 256–258
    Exception, see: Exceptions, 321
    float, 29, 39
    function, 76, 399
    implicit conversion, 51
    int, 29, 38
    iterator, 269
    list, 30, 190, 397
    namedtuple, 351, 362, 363
    NoneType, 30
    range, 247–249, 381
    set, 345–350
    static typing, 31
    str, 29, 35, 406
    tuple, 30, 197, 408
    Unicode string, 458

Unicode, 33, 35, 406, 408, 457
unpacking, 212, 258, 403, 408
UTF-8, 457

vacuously true, 349
variable, 44, 46, 409
vertex, 409

Windows-1252 (code page), 457

A first course in programming and computer science using the Python programming language. Covers fundamentals including types and variables, functions, loops, branching, and exceptions, as well as file I/O, graphs, plotting, pseudo-randomness, and use of structured data. This textbook was originally developed for the University of Vermont's CS 1210 Introduction to Programming.

https://go.uvm.edu/cs1210

ISBN 979-8-9887092-1-3

9 798988 709213