

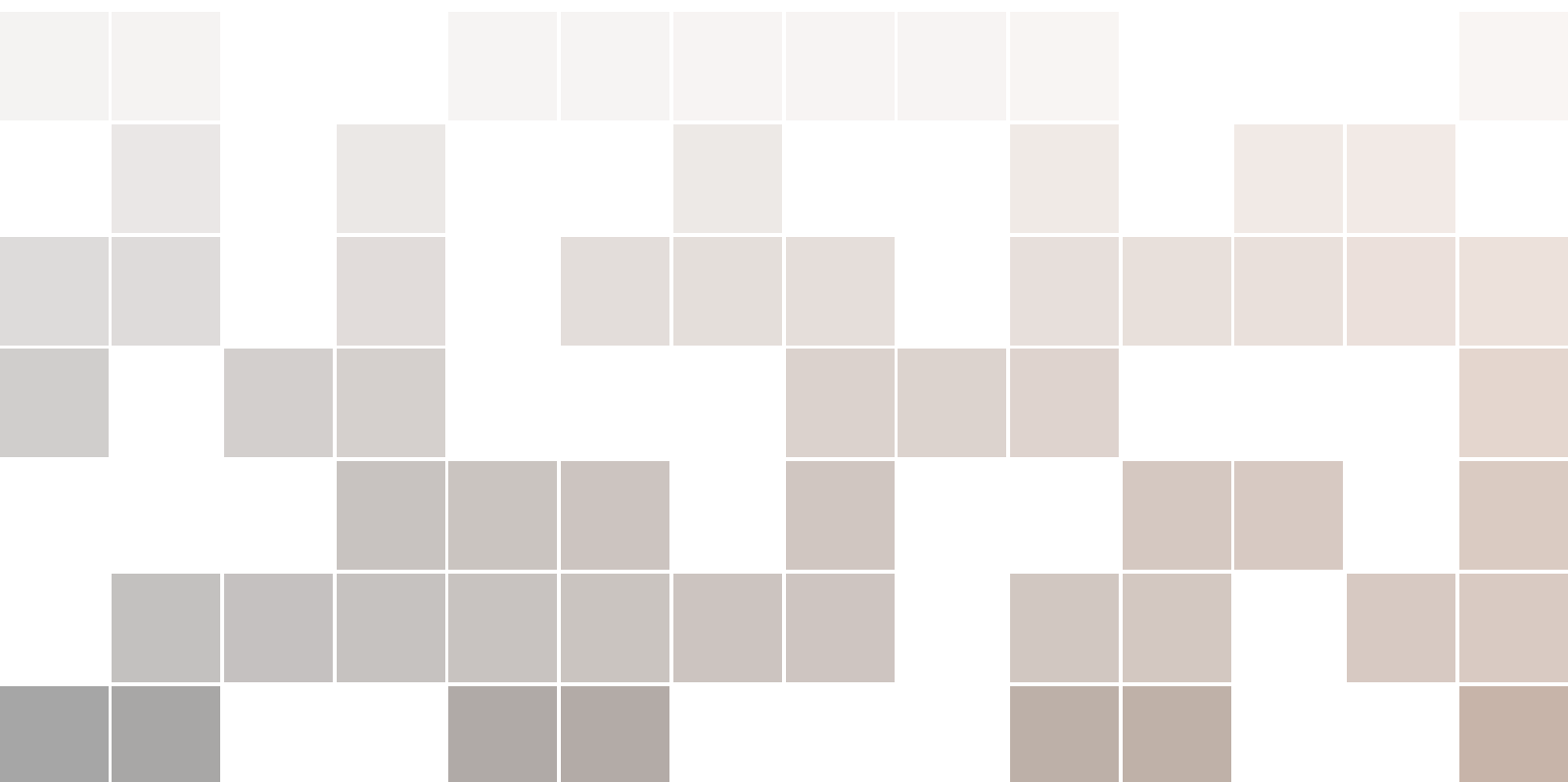


Introduction to Programming with Python 3

Lilian Blot

Based on
“How to think like a computer scientist”

by Allen B. Downey



Copyright © 2018 Lilian Blot

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

I Part I - Basic Programming Concepts

1	The way of the program	2
1.1	The Python programming language	2
1.2	What is a program?	4
1.3	What is debugging?	4
1.3.1	Syntax errors	4
1.3.2	Runtime errors	5
1.3.3	Semantic errors	5
1.3.4	Experimental debugging	5
1.4	Formal and natural languages	6
1.5	The first program	7
1.6	Debugging	9
1.7	Glossary	10
1.8	Exercises	11
2	Variables, expressions and statements	12
2.1	Values and types	12
2.2	Variables	13
2.3	Variable names and keywords	14
2.4	Statements	15
2.5	Operators and operands	16

2.6	Expressions	16
2.7	Order of operations	18
2.8	String operations	19
2.9	Debugging	19
2.10	Glossary	20
2.11	Exercises	21
3	Code Format	22
3.1	A Foolish Consistency is the Hobgoblin of Little Minds	23
3.2	Code Layout	23
3.3	Whitespace in Expressions and Statements	25
3.4	Variable Names	26
4	Conditionals	27
4.1	Modulus operator	27
4.2	Boolean expressions	28
4.3	Logical operators	28
4.4	Conditional execution	29
4.5	Alternative execution	29
4.6	Chained conditionals	31
4.7	Nested conditionals	32
4.8	PEP 8 Recommendations	33
4.8.1	Indentation	33
4.8.2	Compound Statements	33
4.8.3	Comparisons to singletons	34
5	Iteration	35
5.1	Multiple assignment	35
5.2	Updating variables	36
5.3	Simple repetition	37
5.4	The <code>while</code> statement	37
5.5	<code>break</code>	39
5.6	Square roots	41
5.7	Algorithms	42
5.8	PEP 8 Recommendations	43
5.9	Debugging	43
5.10	Glossary	43
5.11	Exercises	44

6	Functions	45
6.1	Function calls	45
6.2	Type conversion functions	46
6.3	Common built-in functions	47
6.4	Keyboard input	47
6.5	Math functions	48
6.6	Composition	49
6.7	Adding new functions	49
6.8	Definitions and uses	51
6.9	Flow of execution	51
6.10	Parameters and arguments	52
6.11	Variables and parameters are local	53
6.12	Stack diagrams	54
6.13	Void functions and non-void functions	55
6.14	Why functions?	56
6.15	PEP 8 Recommendations	56
6.15.1	Function Names	56
6.15.2	Function Arguments	56
6.15.3	Return statement	57
6.15.4	Indentation	57
6.15.5	Blank lines and White Spaces	59
6.16	Debugging	59
6.16.1	Editor	59
6.16.2	Traceback	59
6.17	Glossary	60
6.18	Exercises	62
7	Code Documentation	64
7.1	Introduction	64
7.1.1	Why Documenting Your Code Is So Important	66
7.2	Basics of Commenting Code	66
7.3	Documenting Code via Python Docstring	69
7.3.1	Google Comments and Docstrings Standard	70
7.3.2	NumPy Docstring Standard	71
7.3.3	EpyText Docstring Standard	73
7.3.4	Examples	75
8	Recursion	77
8.1	Stack diagrams for recursive functions	79
8.2	Infinite recursion	80

8.3	Requirements	80
8.4	Glossary	81
8.5	Exercises	81

II

Part II - Built-in Data Structures

9	Strings	85
9.1	A string is a sequence	85
9.2	Function len	86
9.3	Traversal with a for loop	86
9.4	String slices	88
9.5	Strings are immutable	89
9.6	Searching	89
9.7	Looping and counting	90
9.8	string methods	90
9.9	The in operator	91
9.10	String comparison	92
9.11	Debugging	93
9.12	Glossary	94
9.13	Exercises	95
10	Lists	97
10.1	A list is a sequence	97
10.2	Lists are mutable	98
10.3	Traversing a list	99
10.4	List operations	100
10.5	List slices	100
10.6	List methods	101
10.7	Map, filter and reduce	102
10.8	Deleting elements	103
10.9	Lists and strings	104
10.10	Objects and values	105
10.11	Aliasing	106
10.12	List arguments	107
10.13	Debugging	109
10.14	Glossary	110
10.15	Exercises	111

11	Dictionaries	113
11.1	Dictionary as a set of counters	115
11.2	Looping and dictionaries	117
11.3	Reverse lookup	117
11.4	Dictionaries and lists	118
11.5	Debugging	120
11.6	Glossary	121
11.7	Exercises	121
12	Tuples	123
12.1	Tuples are immutable	123
12.2	Tuple assignment	125
12.3	Tuples as return values	125
12.4	Variable-length argument tuples	126
12.5	Lists and tuples	127
12.6	Dictionaries and tuples	129
12.7	Comparing tuples	131
12.8	Sequences of sequences	132
12.9	Glossary	132
12.10	Exercises	133
13	Files	135
13.1	Persistence	135
13.2	Reading from a file	135
13.3	Writing to a file	137
13.4	Format operator	137
13.5	Filenames and paths	138
13.6	Catching exceptions	140
13.6.1	finally clause	141
13.6.2	else clause	141
13.7	Writing modules	142
13.8	Debugging	144
13.9	Glossary	145
13.10	Exercises	145

III

Part III - User Defined Data Structures

14	Classes and objects	149
14.1	User-defined types	149

14.2	Attributes	150
14.3	Rectangles	151
14.4	Instances as return values	152
14.5	Objects are mutable	153
14.6	Copying	153
14.7	Debugging	155
14.8	Glossary	155
15	Classes and functions	157
15.1	Time	157
15.2	Pure functions	158
15.3	Modifiers	159
15.4	Prototyping versus planning	160
15.5	Debugging	161
15.6	Glossary	162
15.7	Exercises	162
16	Classes and methods	164
16.1	Object-oriented features	164
16.2	What is an instance?	165
16.3	Printing objects	165
16.4	The <code>self</code> variable	166
16.5	Another example	167
16.6	A more complicated example	168
16.7	The <code>init</code> method	168
16.8	The <code>__str__</code> method	169
16.9	Operator overloading	170
16.10	Type-based dispatch	171
16.11	Polymorphism	174
16.12	Debugging	175
16.13	Glossary	175
16.14	Exercises	175

IV**Back Matter**

Bibliography	178
Index	179



Part I - Basic Programming Concepts

1	The way of the program	2
2	Variables, expressions and statements .	12
3	Code Format	22
4	Conditionals	27
5	Iteration	35
6	Functions	45
7	Code Documentation	64
8	Recursion	77

1. The way of the program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called, "The way of the program."

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 The Python programming language

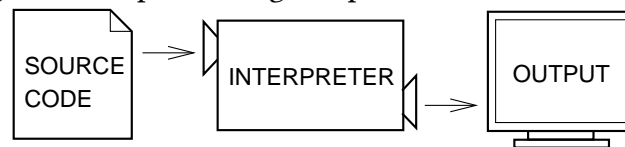
The programming language you will learn is Python. Python is an example of a **high-level language**; other high-level languages you might have heard of are C++, C#, VB, JavaScript, and Java.

There are also **low-level languages**, sometimes referred to as "machine languages" or "assembly languages." Loosely speaking, computers can only execute programs written in low-level languages. So programs written in a high-level language have to be processed

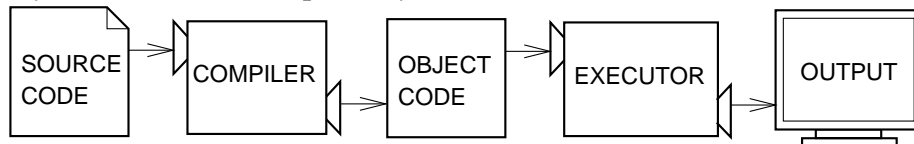
before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

The advantages are enormous. First, it is much easier for a human to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another. Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.



A compiler reads the program and translates it completely before the program starts running. In this context, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.



Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: **interactive mode** and **script mode**. In interactive mode, you type Python programs and the interpreter prints the result:

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

The chevron, `>>>`, is the **prompt** the interpreter uses to indicate that it is ready. If you type `1 + 1`, the interpreter replies `2`.

Alternatively, you can store code in a file and use the interpreter to execute the contents of the file, which is called a **script**. By convention, Python scripts have names that end with `.py`. To execute the script, you have to tell the interpreter the name of the file. In a Windows command shell, you would type:

```
C:\code> python helloworld.py
```

where `helloworld.py` is the script located in the folder `C:\code` you want to execute. In other development environments, the details of executing scripts are different. You can find instructions for your environment at the Python website `python.org`.

Working in interactive mode is convenient for testing small pieces of code because you can type and execute them immediately. But for anything more than a few lines, you should save your code as a script so you can modify and execute it in the future.

1.2 What is a program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

input: Get data from the keyboard, a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

conditional execution: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions. That may be a little vague, but we will come back to this topic when we talk about **algorithms**.

1.3 What is debugging?

Programming is error-prone. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down is called **debugging**. Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

1.3.1 Syntax errors

Python can only execute a program if the syntax is correct; otherwise, the interpreter displays an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so `(1 + 2)` is legal, but `1 + 2)` is a **syntax error**.

In English readers can tolerate most syntax errors, Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

1.3.2 Runtime errors

The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

1.3.3 Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see. Debugging is also like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out [1], “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.”

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, “One of Linus’s earlier projects was a program that would

switch between printing AAAA and BBBB. This later evolved to Linux.” (*The Linux Users’ Guide Beta Version 1*).

Later chapters will make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Definition 1.4.1 — Natural languages. are the languages people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Definition 1.4.2 — Formal languages. are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Formal languages tend to have strict rules about syntax. For example, $3 + 3 = 6$ is a syntactically correct mathematical statement, but $3 + = 6 \times 3$ is not. Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, operators. One of the problems with $3 + = 3 \times 6$ is that \times is not a legal token in mathematics.

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement $3 + = 6 - 3$ is illegal because even though $+$ and $=$ are legal tokens, you can’t have one right after the other.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**.

For example, when you hear the sentence, “The penny dropped,” you understand that “the penny” is the subject and “dropped” is the predicate. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a penny is and what it means to drop, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are some important differences:

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, “The penny dropped,” there is probably no penny and nothing dropping¹. Formal languages mean exactly what they say.

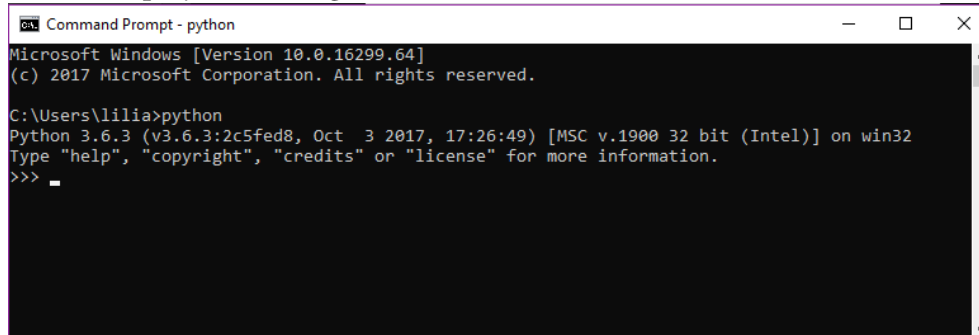
Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally, the first program you write in a new language is called “Hello, World!” because all it does is display the words, “Hello, World!”

Before we can write our first program we need to launch the Python interpreter. In this book, we assume you have Python 3 installed on the Windows 10 OS. There are two ways to launch Python:

1. Open a command window, and at the prompt type `python`. The command window should display something like:



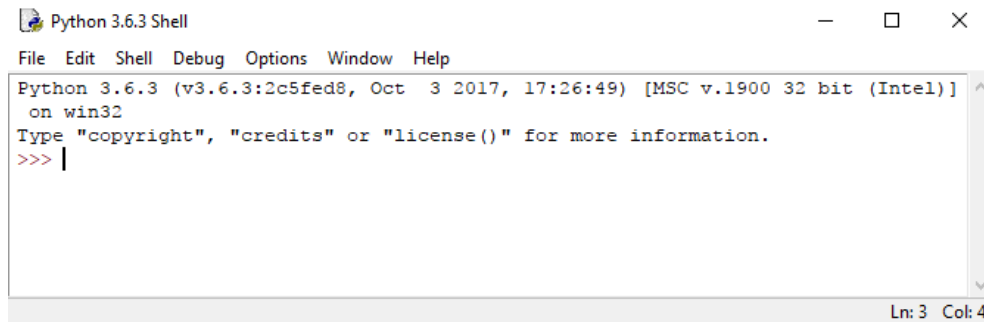
```
Command Prompt - python
Microsoft Windows [Version 10.0.16299.64]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\lilia>python
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

If this is not the case you may want to check that Python is in your PATH environment variable.

¹This idiom means that someone realized something after a period of confusion.

2. Open IDLE(Python 3.6 32 bits) from the Windows start menu. A Python shell opens and is ready to accept your code:



Now that we have a Python shell opened, we can write our program. In Python 3, it looks like this:

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

The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the “Hello, World!” program. By this standard, Python does about as well as possible.

Now that we can start programming, it might be a good time to illustrate the type of errors we may encounter whilst programming. The first one we mentioned was a syntax error like this one:

```
>>> 1 + 2)
SyntaxError: invalid syntax
>>>
```

where the opening bracket is missing.

The second type of error are runtime errors like the division by zero shown here:

```
>>> print(10/0)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(10/0)
ZeroDivisionError: division by zero
>>>
```

Probably the most challenging one is a semantic error. For example, if we try to convert a temperature t_F in Fahrenheit into Celsius the formula to use is:

$$t_C = \frac{5}{9}(t_F - 32)$$

Now if we write the following implementation:

```
>>> print('Fahrenheit 35 in Celsius degree is:', 5/9*35-32)
Fahrenheit 35 in Celsius degree is: -12.555555555555554
>>>
```

the program runs and does not create any error. Does that mean that the program is correct? Unfortunately, NO. The result should have been 1.66 not -12.55 .

Exercise 1.1 Write the correct implementation of the conversion from Fahrenheit to Celsius. ■

In conclusion, it is not because your program runs and returns a result that your work is finished. You must ensure that the returned result is correct. It is essential when implementing a code to define a series of tests (i.e. a series of known outputs given certain inputs) that can validate your code.

1.6 Debugging

It is a good idea to read this book in front of a computer so you can try out the examples as you go. You can run most of the examples in interactive mode, but if you put the code into a script, it is easier to try out variations. Whenever you are experimenting with a new feature, you should try to make mistakes. For example, in the “Hello, world!” program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `print` wrong?

R This kind of experiment helps you remember what you read; it also helps with debugging, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later and accidentally.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent or embarrassed. There is evidence that people naturally respond to computers as if they were people². When they work well, we think of them as teammates, and when they are obstinate or rude, we respond to them the same way we respond to rude, obstinate people. Preparing for these reactions might help you deal with them. One approach is to think of the computer as an employee with certain strengths, like speed and precision, and particular weaknesses, like lack of empathy and inability to grasp the big picture. Your job is to be a good manager: find ways to take advantage of the strengths and mitigate the weaknesses. And find ways to use your emotions to engage with the problem, without letting your reactions interfere with your ability to work effectively.

²See Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*.

Learning to debug can be frustrating, but it is a valuable skill that is useful for many activities beyond programming. At the end of each chapter there is a debugging section, like this one, with my thoughts about debugging. I hope they help!

1.7 Glossary

- **1 – problem solving:** The process of formulating a problem, finding a solution, and expressing the solution.
- **2 – high-level language:** A programming language like Python that is designed to be easy for humans to read and write.
- **3 – low-level language:** A programming language that is designed to be easy for a computer to execute; also called “machine language” or “assembly language.”
- **4 – portability:** A property of a program that can run on more than one kind of computer.
- **5 – interpret:** To execute a program in a high-level language by translating it one line at a time.
- **6 – compile:** To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.
- **7 – source code:** A program in a high-level language before being compiled.
- **8 – object code:** The output of the compiler after it translates the program.
- **9 – executable:** Another name for object code that is ready to be executed.
- **10 – prompt:** Characters displayed by the interpreter to indicate that it is ready to take input from the user.
- **11 – script:** A program stored in a file (usually one that will be interpreted).
- **12 – interactive mode:** A way of using the Python interpreter by typing commands and expressions at the prompt.
- **13 – script mode:** A way of using the Python interpreter to read and execute statements in a script.
- **14 – program:** A set of instructions that specifies a computation.
- **15 – algorithm:** A general process for solving a category of problems.
- **16 – bug:** An error in a program.
- **17 – debugging:** The process of finding and removing any of the three kinds of programming errors.
- **18 – syntax:** The structure of a program.
- **19 – syntax error:** An error in a program that makes it impossible to parse (and therefore impossible to interpret).
- **20 – exception:** An error that is detected while the program is running.
- **21 – semantics:** The meaning of a program.
- **22 – semantic error:** An error in a program that makes it do something other than what the programmer intended.
- **23 – natural language:** Any one of the languages that people speak that evolved naturally.
- **24 – formal language:** Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

- 25 — **token**:. One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.
- 26 — **parse**:. To examine a program and analyze the syntactic structure.
- 27 — **print statement**:. An instruction that causes the Python interpreter to display a value on the screen.

1.8 Exercises

Exercise 1.2 Use a web browser to go to the Python website `python.org`. This page contains information about Python and links to Python-related pages, and it gives you the ability to search the Python documentation.

For example, if you enter `print` in the search window, the first link that appears is the documentation of the `print` statement. At this point, not all of it will make sense to you, but it is good to know where it is.

Exercise 1.3 Start the Python interpreter and type `help()` to start the online help utility. Or you can type `help('print')` to get information about the `print` statement.

If this example doesn't work, you may need to install additional Python documentation or set an environment variable; the details depend on your operating system and version of Python.

Exercise 1.4 Start the Python interpreter and use it as a calculator. Python's syntax for math operations is almost the same as standard mathematical notation. For example, the symbols `+`, `-` and `/` denote addition, subtraction and division, as you would expect. The symbol for multiplication is `*`.

If you run a 10 kilometer race in 43 minutes 30 seconds, what is your average time per mile? What is your average speed in miles per hour? (Hint: there are 1.61 kilometers in a mile).

2. Variables, expressions and statements

In Chapter 1 we wrote our first (albeit very basic) program. In this chapter we will learn to write more complex programs using fundamental programming techniques such as the use of variables. We will learn about operators and operands, the building blocks of a Python expressions. We will also look at how to build more complex expression and assign the result of the computation to a variable.

2.1 Values and types

A **value** is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World! '.

These values belong to different **types** also known as **classes** in Python 3. For example, 2 is an integer, and 'Hello, World! ' is a **string**, so-called because it contains a “string” of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for integers.

```
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Not surprisingly, strings belong to the type (class) `str` and integers belong to the type (class) `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called **floating-point**.

```
>>> type(3.2)
<class 'float'>
```

What about values like `'17'` and `'3.2'`? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

They are strings.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in `1,000,000`. This is not a legal integer in Python, but it is legal:

```
>>> print(1,000,000)
1 0 0
```

Well, that's not what we expected at all! Python interprets `1,000,000` as a comma-separated sequence of integers, which it prints with spaces between. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

2.2 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

An **assignment statement** creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second gives the integer 17 to `n`; the third assigns the (approximate) value of π to `pi`.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the previous example:

```
message —> 'And now for something completely different'
n —> 17
pi —> 3.1415926535897931
```

To display the value of a variable, you can use a `print` statement:

```
>>> print(n)
17
>>> print(pi)
3.14159265359
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

R If you type an integer with a leading zero, you might get a confusing error:

```
>>> zipcode = 02492
                ^
SyntaxError: invalid token
```

2.3 Variable names and keywords

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for. Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later). The underscore character (`_`) can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones is illegal because it does not begin with a letter. more@ is illegal because it contains an illegal character, @. But what's wrong with class? It turns out that class is one of Python's **keywords**. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names. Python 3 has 33 keywords has shown in Table 2.1 on page 15.

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Table 2.1: Keywords in Python 3 programming language.

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

- R** There are two naming conventions used in the Python language. We have already seen the **snake_case** notation where variables name consisting of several words have the words in lowercase separated by an underscore, like `circle_area`. The other convention is **mixedCase**; if the name consists of several words, we concatenate them into one, making the first word lowercase and capitalising the first letter of each subsequent word, like `circleArea`.

2.4 Statements

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print and assignment. When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one. A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
1 print(1)
2 x = 2
3 print(x)
```

produces the output

```
1
2
```

The assignment statement produces no output.

2.5 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called **operands**. The operators `+`, `-`, `*`, `/` and `**` perform addition, subtraction, multiplication, division and exponentiation, as in the following examples:

```
| 20+32    hour-1    hour*60+minute    minute/60    5**2    (5+9)*(15-7)
```

Python 3, as well as other languages have an **integer division** `//`. The integer division operate a **floor division**, that is the operator returns the whole part of the division. The result of the integer division is an integer. If both operands are integer the value returned is a integer. Otherwise, if at least one of the operand is a float, the value returned is a float.

```
>>> minute = 59
>>> minute // 60
0
>>> minute = 61
>>> minute // 60
1
>>> minute // 60.0
1.0
>>> minute / 60.0
1.0166666666666666
```

2.6 Expressions

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
1 | 17
2 | x
3 | x + 17
```


If you type an expression in interactive mode, the interpreter **evaluates** it and displays the result:

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

But in a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Exercise 2.1 Type the following statements in the Python interpreter to see what they do:

```
5
x = 5
x + 1
```

Now put the same statements into a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again. ■

You can use a variable in an expression, and you can assign the result of an expression to a variable. In addition, a variable can be used on both side of the assignment operator = like in `x = x + 1`. In this assignment statement the result of `x + 1` is assigned to `x`. For example, if `x` has the value 1 before the assignment statement is executed, then `x` becomes 2 after the statement is executed.

R In mathematics, $x = 2 * x + 1$ denotes an equation (where $x = -1$ is the solution). In python, it is an assignment statement, and if x has the value 1 before the statement, executing the statement would change x to 3.

2.7 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

- **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, $2 * (3 - 1)$ is 4, and $(1 + 1) ** (5 - 2)$ is 8. You can also use parentheses to make an expression easier to read, as in $(\text{minute} * 100) / 60$, even if it doesn't change the result.
- **E**xponentiation has the next highest precedence, so $2 ** 1 + 1$ is 3, not 4, and $3 * 1 ** 3$ is 3, not 27.
- **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So $2 * 3 - 1$ is 5, not 4, and $6 + 4 / 2$ is 8, not 5.
- Operators with the same precedence are evaluated from left to right. So in the expression $\text{degrees} / 2 * \text{pi}$, the division happens first and the result is multiplied by pi , therefore the expression is equal to $\frac{\pi}{2} \text{degrees}$. To divide by 2π , you can use parentheses (e.g. $\text{degrees} / (2 * \text{pi})$) or write $\text{degrees} / 2 / \text{pi}$.

Operator	Description
<code>**</code>	Exponentiation (raise to the power)
<code>+, -</code>	Unary plus and minus (method names for the last two are <code>+</code> @ and <code>-</code> @)
<code>*, /, %, //</code>	Multiply, divide, modulo and floor division
<code>+, -</code>	Addition and subtraction
<code><=, <, >, >=</code>	Comparison operators
<code><>, ==, !=</code>	Equality operators
<code>is, is not</code>	Identity operators
<code>in, not in</code>	Membership operators
<code>not</code>	Logical operators
<code>and</code>	Logical operators
<code>or</code>	Logical operators
<code>=, %=, /=, //=, -=, +=, *=, **=</code>	Assignment operators

Table 2.2: Operators precedence in Python from highest precedence (top) to lowest (bottom).

2.8 String operations

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

```
| '2' - '1'      'eggs' / 'easy'      'third' * 'a charm'
```

The `+` operator works with strings, but it might not do what you expect: it performs **concatenation**, which means joining the strings by linking them end-to-end. For example:

```
1 | first = 'throat'
2 | second = 'warbler'
3 | print(first + second)
```

The output of this program is `throatwarbler`.

The `*` operator also works on strings; it performs repetition. For example, `'Spam' * 3` is `'SpamSpamSpam'`. If one of the operands is a string, the other has to be an integer. This use of `+` and `*` makes sense by analogy with addition and multiplication. Just as $4 * 3$ is equivalent to $4 + 4 + 4$, we expect `'Spam' * 3` to be the same as `'Spam' + 'Spam' + 'Spam'`, and it is.

On the other hand, there is a significant way in which string concatenation is different from integer addition. Concatenation is not commutative, which means that `'spam' + 'sandwich'` is not the same as `'sandwich' + 'spam'`, whereas $3 + 4$ is the same as $4 + 3$.

2.9 Debugging

At this point the syntax error you are most likely to make is an illegal variable name, like `class` and `yield`, which are keywords, or `odd~job` and `GBP£`, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

```
| >>> bad name = 5
| SyntaxError: invalid syntax
```

For syntax errors, the error messages don't help much. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

The runtime error you are most likely to make is a `NameError` that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    interest = principle * rate
NameError: name 'principle' is not defined
```

Variables names are case sensitive, so `Principal` is not the same as `principal`.

At this point the most likely cause of a semantic error is the order of operations. For example, to evaluate $\frac{1}{2\pi}$, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

But the division happens first, so you would get $\pi/2$, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

2.10 Glossary

- **28 — value:** One of the basic units of data, like a number or string, that a program manipulates.
- **29 — type:** A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).
- **30 — integer:** A type that represents whole numbers.
- **31 — floating-point:** A type that represents numbers with fractional parts.
- **32 — string:** A type that represents sequences of characters.
- **33 — variable:** A name that refers to a value.
- **34 — statement:** A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.
- **35 — assignment:** A statement that assigns a value to a variable.
- **36 — state diagram:** A graphical representation of a set of variables and the values they refer to.
- **37 — keyword:** A reserved word that is used by the compiler to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.
- **38 — operator:** A special symbol that represents a simple computation like addition, multiplication, or string concatenation.
- **39 — operand:** One of the values on which an operator operates.

floor division: The operation that divides two numbers and chops off the fraction part.

- **40 — expression:** A combination of variables, operators, and values that represents a single result value.
- **41 — evaluate:** To simplify an expression by performing the operations in order to yield a single value.
- **42 — rules of precedence:** The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

- 43 — **concatenate:** To join two operands end-to-end.

2.11 Exercises

Exercise 2.2 Assume that we execute the following assignment statements:

```
width = 17
height = 12.0
delimiter = '.'
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. `width/2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`
5. `delimiter * 5`

Use the Python interpreter to check your answers. ■

Exercise 2.3 Practice using the Python interpreter as a calculator:

1. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5? Hint: 392.6 is wrong!
2. Suppose the cover price of a book is \$24.95, but bookstores get a 40% discount. Shipping costs \$3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
3. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast? ■

3. Code Format

Perhaps you thought that "getting it working" was the most important part of coding. This is not the case, the most important aspect of your code is about communication. When working as a professional developer, the code that you create today has a good chance of changing in subsequent releases, but the readability of your code will have a profound effect on all the changes that will ever be made. The coding style and readability set precedents that continue to affect maintainability and extensibility long after the original code has been changed beyond recognition. Your style and discipline survives, even though your code does not.

“ Code formatting is important. It is too important to ignore and it is too important to treat religiously. ”

Robert C. Martin

The guidelines provided in this chapter are specific to the Python language, and will vary depending on the language used. If you are implementing a project using a different language, you should first look for the conventions specific to that language in addition to conventions set for the given project.

Throughout this book, we will be adding to the conventions described in this chapter. To find these conventions easily, they will be grouped into a single section within a chapter. The title of such sections will contain the keyword PEP. We will be exploring part of the PEP 8 [3] and PEP 257 [2] (Docstring Conventions) where it applies. PEP 8 and PEP 257 were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide [2].

The Python style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

Note that many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides should take precedence for that project. It is essential that conventions within a project remain consistent.

3.1 A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided throughout the booklet are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 [13] says, "Readability counts".

“ Code is more often read than written.

”

Guido Van Rossum

A style guide is about consistency. Consistency with the PEP 8 style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

R However, know when to be inconsistent – sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgement. Note, this is beyond the scope of what you will be learning in your first year at University.

3.2 Code Layout

Indentation

Use 4 spaces per indentation level.

Tabs or Spaces?

Spaces are the preferred indentation method. Tabs should be used solely to remain consistent with code that is already indented with tabs. Python 3 disallows mixing the use of tabs and spaces for indentation. Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.

When invoking the Python 2 command line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When using `-tt` these warnings become errors. These options are highly recommended!

Maximum Line Length

Limit all lines to a maximum of 79 characters. For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters. Limiting the required editor window width makes it possible to have several

files open side-by-side, and works well when using code review tools that present the two versions in adjacent columns.

The default wrapping in most tools disrupts the visual structure of the code, making it more difficult to understand. The limits are chosen to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final column when wrapping lines. Some web based tools may not offer dynamic line wrapping at all.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Should a Line Break Before or After a Binary Operator?

For decades the recommended style was to break after binary operators. But this can hurt readability in two ways: the operators tend to get scattered across different columns on the screen, and each operator is moved away from its operand and onto the previous line. Here, the eye has to do extra work to tell which items are added and which are subtracted:

```
1  # No: operators sit far away from their operands
2  income = (gross_wages +
3             taxable_interest +
4             (dividends - qualified_dividends) -
5             ira_deduction -
6             student_loan_interest)
```

To solve this readability problem, mathematicians and their publishers follow the opposite convention. Donald Knuth explains the traditional rule in his Computers and Typesetting series: "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations" [3].

Following the tradition from mathematics usually results in more readable code:

```
1  # Yes: easy to match operators with operands
2  income = (gross_wages
3            + taxable_interest
4            + (dividends - qualified_dividends)
5            - ira_deduction
6            - student_loan_interest)
```

In Python code, it is permissible to break before or after a binary operator, as long as the convention is consistent locally. For new code Knuth's style is suggested.

3.3 Whitespace in Expressions and Statements

- Avoid extraneous whitespace. For example, more than one space around an assignment (or other) operator to align it with another.

```

1  #Yes:
2  x = 1
3  y = 2
4  long_variable = 3
5
6  #No:
7  x           = 1
8  y           = 2
9  long_variable = 3

```

- Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker. Some editors don't preserve it and many projects (like CPython itself) have pre-commit hooks that reject it.
- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

```


1  #Yes:
2  i = i + 1
3  submitted += 1
4  x = x*2 - 1
5  hypot2 = x*x + y*y
6  c = (a+b) * (a-b)
7
8  #No:
9  i=i+1
10 submitted +=1
11 x = x * 2 - 1
12 hypot2 = x * x + y * y
13 c = (a + b) * (a - b)

```

3.4 Variable Names

Variable names should be lowercase, with words separated by underscores as necessary to improve readability.

mixedCase is allowed only in contexts where that's already the prevailing style (e.g. `threading.py`), to retain backwards compatibility.

-  When using acronyms in mixedCase, capitalize all the letters of the acronym. Thus `HTTPServerError` is better than `httpServerError`.

4. Conditionals

4.1 Modulus operator

The **modulus operator** works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if $x \% y$ is zero, then x is divisible by y . Also, you can extract the right-most digit or digits from a number. For example, $x \% 10$ yields the right-most digit of x (in base 10). Similarly $x \% 100$ yields the last two digits.

4.2 Boolean expressions

A **boolean expression** is an expression that is either true or false. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` and `False` are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

The `==` operator is one of the **relational operators**; the others are:

```
1  x != y           # x is not equal to y
2  x > y           # x is greater than y
3  x < y           # x is less than y
4  x >= y          # x is greater than or equal to y
5  x <= y          # x is less than or equal to y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a relational operator. There is no such thing as `=<` or `=>`.

4.3 Logical operators

There are three **logical operators**: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. If you are not familiar with the logical operators, their **truth table** are given in Table ?? on page ?. The tables read as follow, the operands value are given in the first row and first column. The operator is given in the first cell (top-left) of the table. Looking at the `and` operator, the result of the expression `True and True` is `True`, whereas the result of the expression `True and False` is `False`.

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

not	True	False
	False	True

Table 4.1: Truth table for the three logical operators `and`, `or`, and `not`.

For example, `x > 0 and x < 10` is true only if `x` is greater than 0 *and* less than 10. Another example, `n%2 == 0 or n%3 == 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3. Finally, the `not` operator negates a Boolean expression, so `not (x > y)` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

R Strictly speaking, the operands of the logical operators should be Boolean expressions, but Python is not very strict. Any nonzero number is interpreted as “True”, whereas 0 is interpreted as “False”.

```
>>> 17 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to (shall I say **MUST**) avoid it, even if you know what you are doing. Another developer maintaining your code may not be familiar with the subtleties.

4.4 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
1 | if x > 0:
2 |     print('x is positive')
```

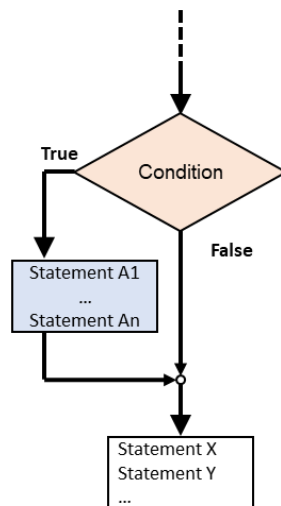
The Boolean expression after the `if` statement is called the **condition**. If it is true, then the indented statement gets executed. If not, nothing happens. This is illustrated in the flow diagram shown in Figure 4.1.

`if` statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called **compound statements**. There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven’t written yet). In that case, you can use the `pass` statement, which does nothing.

```
1 | if x < 0:
2 |     pass                # need to handle negative values!
```

4.5 Alternative execution

A second form of the `if` statement is **alternative execution** (aka `if-else` statement), in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

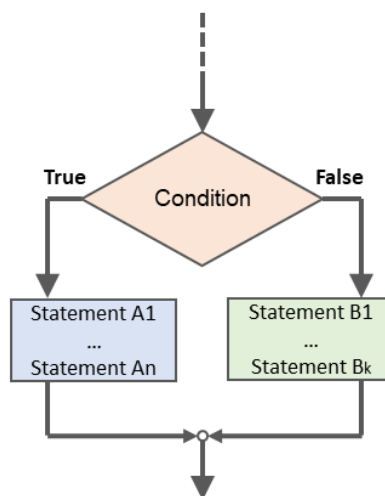
Figure 4.1: Conditional `if` statement control flow diagram.

```

1  if x%2 == 0:
2      print('x is even')
3  else:
4      print('x is odd')

```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution. This is clearly illustrated in the flow diagram shown in Figure 4.2.

Figure 4.2: Conditional `if-else` statement control flow diagram.

4.6 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional** (aka `if-elif-else` statement):

```
1  if x < y:
2      print('x is less than y')
3  elif x > y:
4      print('x is greater than y')
5  else:
6      print('x and y are equal')
```

`elif` is an abbreviation of “else if.” Again, exactly one branch will be executed. There is no limit on the number of `elif` statements. If there is an `else` clause, it has to be at the end, but there doesn’t have to be one.

```
1  if choice == 'a':
2      draw_a()
3  elif choice == 'b':
4      draw_b()
5  elif choice == 'c':
6      draw_c()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes. The flow diagram of a chained conditional is shown in Figure 4.3.

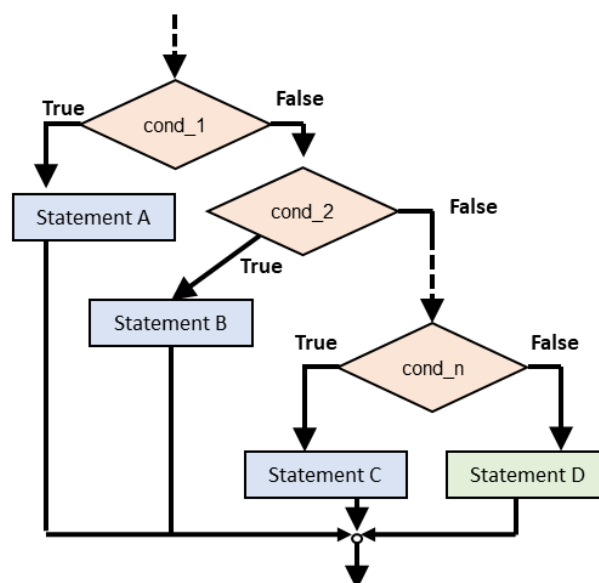


Figure 4.3: Conditional `if-elif-else` statement control flow diagram.

4.7 Nested conditionals

One conditional can also be nested within another. We could have written the trichotomy example like this:

```

1  if x == y:
2      print('x and y are equal')
3  else:
4      if x < y:
5          print('x is less than y')
6      else:
7          print('x is greater than y')

```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well. Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Nested conditionals structures should be used when several statements are common to more than one sub-branch as illustrated in Figure 4.4. Here, statements B1 and B2 have been moved out of the nested `if` statement as if is common to branches C and D. The same flow of execution could have been done using chained conditionals, however statements B1 and B2 would have to be duplicate in each sub-branch.

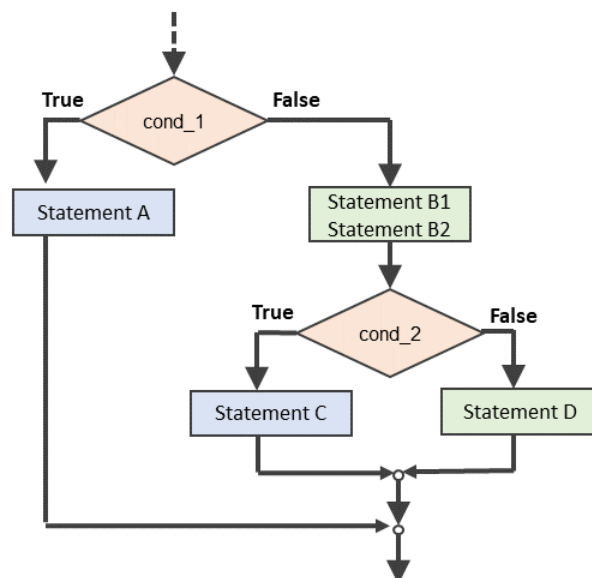


Figure 4.4: Nested conditionals `if` statements control flow diagram.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:


```
1 | if 0 < x:  
2 |     if x < 10:  
3 |         print('x is a positive single-digit number.')
```

The `print` statement is executed only if we make it past both conditionals, so we can get the same effect with the `and` operator:

```
1 | if 0 < x and x < 10:  
2 |     print('x is a positive single-digit number.')
```

4.8 PEP 8 Recommendations

4.8.1 Indentation

When the conditional part of an if-statement is long enough to require that it be written across multiple lines, it's worth noting that the combination of a two character keyword (i.e. `if`), plus a single space, plus an opening parenthesis creates a natural 4-space indent for the subsequent lines of the multiline conditional. This can produce a visual conflict with the indented suite of code nested inside the if-statement, which would also naturally be indented to 4 spaces. The PEP 8 takes no explicit position on how (or whether) to further visually distinguish such conditional lines from the nested suite inside the if-statement. Acceptable options in this situation include, but are not limited to:

```
1 | # No extra indentation.  
2 | if (this_is_one_thing and  
3 |     that_is_another_thing):  
4 |     do_something()  
5 |  
6 | # Add a comment, which will provide some distinction in editors  
7 | # supporting syntax highlighting.  
8 | if (this_is_one_thing and  
9 |     that_is_another_thing):  
10 |     # Since both conditions are true, we can frobnicate.  
11 |     do_something()  
12 |  
13 | # Add some extra indentation on the conditional continuation line.  
14 | if (this_is_one_thing  
15 |     and that_is_another_thing):  
16 |     do_something()
```

4.8.2 Compound Statements

Compound statements (multiple statements on the same line) are generally discouraged.

```
1 | # Yes:  
2 | if foo == 'blah':  
3 |     do_blah_thing()
```

```
4 | do_one()
5 | do_two()
6 | do_three()
7 |
8 |
9 | # Rather not:
10 | if foo == 'blah': do_blah_thing()
11 | do_one(); do_two(); do_three()
```

While sometimes it's okay to put an `if` with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

```
1 | # Rather not:
2 | if foo == 'blah': do_blah_thing()
3 |
4 | # Definitely not:
5 | if foo == 'blah': do_blah_thing()
6 | else: do_non_blah_thing()
7 |
8 | if foo == 'blah': one(); two(); three()
```

4.8.3 Comparisons to singletons

- Comparisons to singletons like `None` should always be done with `is` or `is not`, never the equality operators.
- Also, beware of writing `if x` when you really mean `if x is not None` – e.g. when testing whether a variable or argument that defaults to `None` was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!
- Use `is not` operator rather than `not ... is`. While both expressions are functionally identical, the former is more readable and preferred.

```
1 | # Yes:
2 | if foo is not None:
3 |
4 | # No:
5 | if not foo is None:
```

5. Iteration

5.1 Multiple assignment

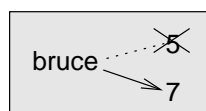
As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
1 | bruce = 5
2 | print(bruce)
3 | bruce = 7
4 | print(bruce)
```

The output of this program is:

```
5
7
```

because the first time `bruce` is printed, its value is 5, and the second time, its value is 7. Here is what **multiple assignment** looks like in a state diagram:



With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because Python uses the equal sign (`=`) for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First, equality is a symmetric relation and assignment is not. For example, in mathematics, if $a = 7$ then $7 = a$. But in Python, the statement `a = 7` is legal and `7 = a` is not. Furthermore, in mathematics, a statement of equality is either true or false, for all time. If $a = b$ now, then a will always equal b . In Python, an assignment statement can make two variables equal, but they don't have to stay that way:

```
1 | a = 5
2 | b = a      # a and b are now equal
3 | a = 3      # a and b are no longer equal
```

The third line changes the value of `a` but does not change the value of `b`, so they are no longer equal. Although multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

5.2 Updating variables

One of the most common forms of multiple assignment is an **update**, where the new value of the variable depends on the old.

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

This means “get the current value of `x`, add one, and then update `x` with the new value.” If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to `x`:

```
>>> y = y+1
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    y = y+1
NameError: name 'y' is not defined
```

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> y = 0
>>> y = y+1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

5.3 Simple repetition

The following code repeats the same task twice:

```
1 word = 'Lilian'
2 print(word)
3 print(word)
```

If we wanted to write a script that print the same thing a thousand times, this approach would be impractical. Thankfully we can do the same thing more concisely with a `for` statement as shown below.

```
1 word = 'Lilian'
2 for i in range(1000):
3     print(word)
```

This is the simplest use of the `for` statement; we will see more later. The syntax of a `for` statement is similar to a function definition. It has a header that ends with a colon and an indented body. The body can contain any number of statements. A `for` statement is sometimes called a **loop** because the flow of execution runs through the body and then loops back to the top. In this case, it runs the body a thousand times.

5.4 The while statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Because iteration is so common, Python provides several language features to make it easier. One is the `for` statement we saw earlier. Another is the `while` statement. Here is a version of the previous script that uses a `while` statement:

```
1 word = 'Lilian'
2 n = 0
3 while n < 1000:
4     print(word)
5     n = n+1
```

Lets consider a script that takes a positive number as input and print a countdown to zero, and then print 'Blastoff.'. The code for the function is:

```
1 n = int(input('Enter a positive integer:'))
2 while n > 0:
3     print(n)
4     n = n-1
5 print('Blastoff!')
```

You can almost read the `while` statement as if it were English. It means, “While `n` is greater than 0, display the value of `n` and then reduce the value of `n` by 1. When you get to 0, display the word `Blastoff!`”

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `True` or `False`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. A control flow diagram for a loop is shown in Figure 5.1.

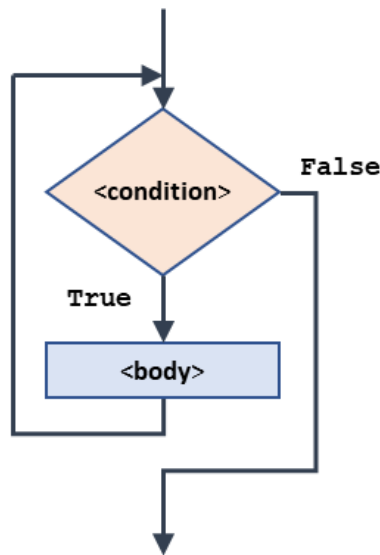


Figure 5.1: Control flow diagram of a simple while loop.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop.

In the previous case, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of n gets smaller each time through the loop, so eventually we have to get to 0. In other cases, it is not so easy to tell:

```
1  n = int(input('Enter a positive integer:'))
2  while n != 1:
3      print(n)
4      if n % 2 == 0:          # n is even
5          n = n / 2
6      else:                  # n is odd
7          n = n * 3 + 1
```

The condition for this loop is $n \neq 1$, so the loop will continue until n is 1, which makes the condition false. Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, n is divided by 2. If it is odd, the value of n is replaced with $n * 3 + 1$. For example, if the input is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1, or that the program terminates. For some particular values of n , we can prove termination. For example, if the starting value is a power of two, then the value of n will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

The hard question is whether we can prove that this program terminates for *all positive values* of n . So far¹, no one has been able to prove it *or* disprove it!

5.5 break

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can use the `break` statement to jump out of the loop.

For example, suppose you want to take input from the user until they type done. You could write:

```
1  while True:
2      line = input('> ')
3      if line == 'done':
4          break
5      print('You wrote:', line)
6
7  print('Done!')
```

The loop condition is `True`, which is always true, so the loop runs until it hits the `break` statement.

¹See wikipedia.org/wiki/Collatz_conjecture.

Each time through, it prompts the user with an angle bracket. If the user types `done`, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> not done
You wrote: not done
> done
Done!
```

A more general flow diagram using a `break` statement is shown in Figure 5.2. In the middle of the body we have a `if` statement with a `testA` condition. If `testA` is true, the `break` statement is called and the program exits the loop immediately without executing statements `M`. Otherwise, the program skips over the `break` and continues the loop as normal.

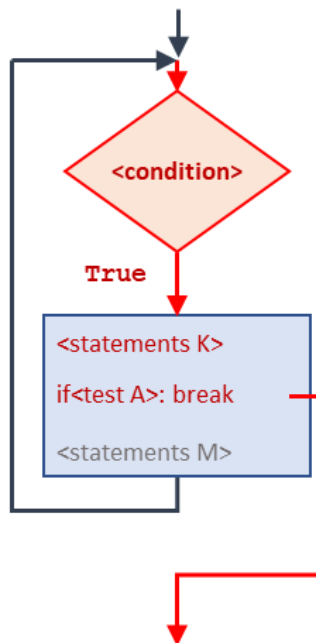


Figure 5.2: Control flow diagram of a simple while loop with a `break` statement. If `testA` is true, the loop will terminate immediately.

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively (“stop when this happens”) rather than negatively (“keep going until that happens.”).

5.6 Square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it. For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a . If you start with almost any estimate, x , you can compute a better estimate with the following formula:

$$y = \frac{x + a/x}{2} \quad (5.1)$$

For example, if a is 4 and x is 3:

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print(y)
2.16666666667
```

Which is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print(y)
2.00641025641
```

After a few more updates, the estimate is almost exact:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print(y)
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> print(y)
2.00000000003
```

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print(y)
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> print(y)
2.0
```

When `y == x`, we can stop. Here is a loop that starts with an initial estimate, `x`, and improves it until it stops changing:

```
1 while True:
2     print(x)
3     y = (x + a/x) / 2
4     if y == x:
5         break
6     x = y
```

For most values of `a` this works fine, but in general it is dangerous to test `float` equality. Floating-point values are only approximately right: most rational numbers, like $1/3$, and irrational numbers, like $\sqrt{2}$, can't be represented exactly with a `float`. Rather than checking whether `x` and `y` are exactly equal, it is safer to use the built-in function `abs` to compute the absolute value, or magnitude, of the difference between them:

```
1 if abs(y-x) < epsilon:
2     break
```

Where `epsilon` has a value like `0.0000001` that determines how close is close enough.

5.7 Algorithms

Newton's method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots). It is not easy to define an algorithm. It might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic. But if you were "lazy," you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming. Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

5.8 PEP 8 Recommendations

The recommendations given for conditional statement in Section 4.8 apply to the iterations statements.

5.9 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more place for bugs to hide.

One way to cut your debugging time is “debugging by bisection.” For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps. Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a `print` statement (or something else that has a verifiable effect) and run the program. If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half. Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

5.10 Glossary

- 44 — **multiple assignment**:: Making more than one assignment to the same variable during the execution of a program.
- 45 — **update**:: An assignment where the new value of the variable depends on the old.
- 46 — **initialization**:: An assignment that gives an initial value to a variable that will be updated.
- 47 — **increment**:: An update that increases the value of a variable (often by one).
- 48 — **decrement**:: An update that decreases the value of a variable.
- 49 — **iteration**:: Repeated execution of a set of statements using either a recursive function call or a loop.
- 50 — **infinite loop**:: A loop in which the terminating condition is never satisfied.

5.11 Exercises

Exercise 5.1 To test the square root algorithm in this chapter, you could compare it with `math.sqrt`. Write a script that prints a table like this:

1.0	1.0	1.0	0.0
2.0	1.41421356237	1.41421356237	2.22044604925e-16
3.0	1.73205080757	1.73205080757	0.0
4.0	2.0	2.0	0.0
5.0	2.2360679775	2.2360679775	0.0
6.0	2.44948974278	2.44948974278	0.0
7.0	2.64575131106	2.64575131106	0.0
8.0	2.82842712475	2.82842712475	4.4408920985e-16
9.0	3.0	3.0	0.0

The first column is a number, a ; the second column is the square root of a computed with the algorithm from Section 5.7; the third column is the square root computed by `math.sqrt`; the fourth column is the absolute value of the difference between the two estimates. ■

Exercise 5.2 The brilliant mathematician Srinivasa Ramanujan found an infinite series^a that can be used to generate a numerical approximation of π :

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}} \quad (5.2)$$

Write a function called `estimate_pi` that uses this formula to compute and return an estimate of π . It should use a `while` loop to compute terms of the summation until the last term is smaller than $1\text{e-}15$ (which is Python notation for 10^{-15}). You can check the result by comparing it to `math.pi`.

You can see my solution at thinkpython.com/code/pi.py. ■

^aSee wikipedia.org/wiki/Pi.

6. Functions

In Chapter 3 we have seen some of the fundamental programming techniques to write simple program. This chapter introduces Python functions to perform common operations. First, we will study the *built-in* functions provided with the Python language. Secondly, we will be looking at common mathematical operations provided by the `math` module. Finally, we will learn how to create our own custom functions.

6.1 Function calls

In the context of programming, a **function** is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name. We have already seen one example of a **function call**:

```
>>> type(32)
<type 'int'>
```

The name of the function is `type`. The expression in parentheses is called the **argument** of the function. The result, for this function, is the type of the argument.

It is common to say that a function “takes” an argument and “returns” a result. The result is called the **return value**.

6.2 Type conversion functions

Python provides built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    int('hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finally, `str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

6.3 Common built-in functions

Python provides many useful function for common programming tasks. We have already seen one, the `print` function. A subset of the built-in function is given in Table 6.1 on page 47.

Function	Description	Example
<code>abs(x)</code>	return the absolute value for <code>x</code>	<code>abs(-2)</code> is 2
<code>max(x1, x2, ...)</code>	Returns the largest among <code>x1</code> , <code>x2</code> , ...	<code>max(1, 5, 2)</code> is 5
<code>min(x1, x2, ...)</code>	Returns the smallest among <code>x1</code> , <code>x2</code> , ...	<code>max(1, 5, -2)</code> is -2
<code>pow(a, b)</code>	Returns a^b . Same as <code>a ** b</code>	<code>pow(2, 3)</code> is 8
<code>round(x)</code>	Returns an integer nearest to <code>x</code> . If <code>x</code> is equally close to two integers, the even one is returned.	<code>round(5.4)</code> is 5

Table 6.1: Simple Python built-in functions

6.4 Keyboard input

The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

Python provides a built-in function called `input` that gets input from the keyboard¹. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and `input` returns what the user typed as a string.

```
>>> input = input()
What are you waiting for?
>>> print(input)
What are you waiting for?
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. `input` can take a prompt as an argument:

```
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> print(name)
Arthur, King of the Britons!
```

The sequence `\n` at the end of the prompt represents a **newline**, which is a special character that causes a line break. That's why the user's input appears below the prompt.

¹In Python 2.7, this function is named `raw_input`.

If you expect the user to type an integer, you can try to convert the return value to `int`:

```
>>> prompt = 'What is the velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What is the velocity of an unladen swallow?
17
>>> int(speed)
17
```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What is the velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    int(speed)
ValueError: invalid literal for int() with base 10:
'What do you mean, an African or a European swallow?'
```

We will see how to handle this kind of error later.

6.5 Math functions

Python has a `math` module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions.

Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a **module object** named `math`. The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called **dot notation**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The `math` module also provides a function called `log` that computes logarithms base e .

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π :


```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression `math.pi` gets the variable `pi` from the `math` module. The value of this variable is an approximation of π , accurate to about 15 digits.

6.6 Composition

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, the argument of a function can be any kind of expression, including arithmetic operators:

```
| x = math.sin(degrees / 360.0 * 2 * math.pi)
```

And even function calls:

```
| x = math.exp(math.log(x+1))
```

Almost anywhere you can put a value, you can put an arbitrary expression, with one exception: the left side of an assignment statement has to be a variable name. Any other expression on the left side is a syntax error².

```
>>> hours = 2                # correct
>>> minutes = hours * 60     # correct
>>> hours * 2 = minutes      # wrong
SyntaxError: can't assign to operator
```

6.7 Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A **function definition** specifies the name of a new function and the sequence of statements that execute when the function is called.

Here is an example:

²We will see exceptions to this rule later.

```
1 | def print_lyrics():
2 |     print("I'm a lumberjack, and I'm okay.")
3 |     print("I sleep all night and I work all day.")
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should (**must** would be a better word) avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments.

The first line of the function definition is called the **header**; the rest is called the **body**. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces (see Section 6.16.1). The body can contain any number of statements.

The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics at 0x03092C00>
>>> type(print_lyrics)
<class 'function'>
```

The value of `print_lyrics` is a **function object**, which has type `'function'`.

The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
1 | def repeat_lyrics():
2 |     print_lyrics()
3 |     print_lyrics()
```

And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

6.8 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
1 def print_lyrics():
2     print("I'm a lumberjack, and I'm okay.")
3     print("I sleep all night and I work all day.")
4
5 def repeat_lyrics():
6     print_lyrics()
7     print_lyrics()
8
9 repeat_lyrics()
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Exercise 6.1 Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get. ■

Exercise 6.2 Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program? ■

6.9 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**. Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off. That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates. What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

6.10 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called **parameters**. Here is an example of a user-defined function that takes an argument:

```
1 | def print_twice(word):  
2 |     print(word)  
3 |     print(word)
```

This function assigns the argument to a parameter named `word`. When the function is called, it prints the value of the parameter (whatever it is) twice. This function works with any value that can be printed.

```
>>> print_twice('Spam')  
Spam  
Spam  
>>> print_twice(17)  
17  
17  
>>> print_twice(math.pi)  
3.14159265359  
3.14159265359
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>>> print_twice('Spam ' * 4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions `'Spam ' * 4` and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`word`).

6.11 Variables and parameters are local

When you create a variable inside a function, it is **local**, which means that it only exists inside the function. For example:

```
1 | def cat_twice(part1, part2):
2 |     cat = part1 + part2
3 |     print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

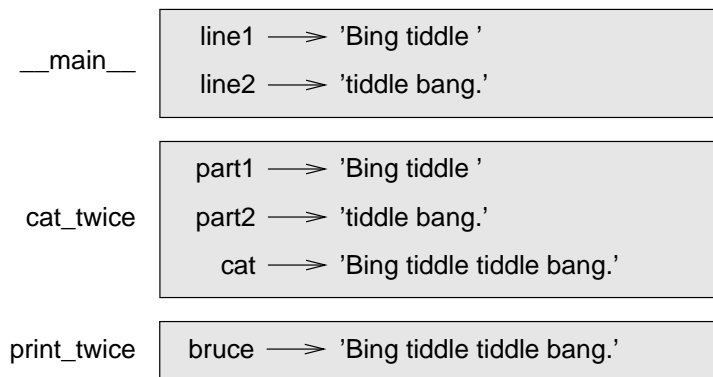
When `cat_twice` terminates, the variable `cat` is destroyed. If we try to print it, we get an exception:

```
>>> print(cat)
NameError: name 'cat' is not defined
```

Parameters are also local. For example, outside `print_twice`, there is no such thing as `word`.

6.12 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to. Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:



The frames are arranged in a stack that indicates which function called which, and so on. In this example, `print_twice` was called by `cat_twice`, and `cat_twice` was called by `__main__`, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to `__main__`. Such variable is said to be **global** as opposed to local.

Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `word` has the same value as `cat`. If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called *that*, all the way back to `__main__`.

For example, if you try to access `cat` from within `print_twice` – for example by adding the statement `print(cat)` in the definition of `print_twice` – you get a `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error. The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

6.13 Void functions and non-void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, we call them **non-void functions**. Other functions, like `print_twice`, perform an action but don't return a value. They are called **void functions**.

When you call a non-void function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
1 | x = math.cos(radians)
2 | golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.2360679774997898
```

But in a script, if you call a non-void function all by itself, the return value is lost forever!

```
| math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful. Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called `None`. Note the capital letter N in `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value `None` is not the same as the string `'None'`. It is a special value that has its own type:

```
>>> print(type(None))
<type 'NoneType'>
```

The functions we have written so far are all void. We will start writing non-void functions in a few chapters.

6.14 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place. This makes your code easier to maintain.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it. The aim is to write code once and use it many times.

6.15 PEP 8 Recommendations

6.15.1 Function Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

`mixedCase` is allowed only in contexts where that's already the prevailing style (e.g. `threading.py`), to retain backwards compatibility.

6.15.2 Function Arguments

If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus `class_` is better than `class`. (Perhaps better is to avoid such clashes by using a synonym.)

Don't use spaces around the = sign when used to indicate a keyword argument or a default parameter value.

```
1  #Yes:
2  def complex(real, imag=0.0):
3      return magic(r=real, i=imag)
```

```
1  #No:
2  def complex(real, imag = 0.0):
3      return magic(r = real, i = imag)
```

6.15.3 Return statement

Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return statement should be present at the end of the function (if reachable).

```
1  #Yes:
2  def foo(x):
3      if x >= 0:
4          return math.sqrt(x)
5      else:
6          return None
7
8  def bar(x):
9      if x < 0:
10         return None
11         return math.sqrt(x)
```

```
1  #No:
2  def foo(x):
3      if x >= 0:
4          return math.sqrt(x)
5
6  def bar(x):
7      if x < 0:
8          return
9      return math.sqrt(x)
```

6.15.4 Indentation

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent [7]. When using a hanging indent the following should be considered; there should be no arguments

on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

```
1  #Yes:  
2  # Aligned with opening delimiter.  
3  foo = long_function_name(var_one, var_two,  
4                             var_three, var_four)  
5  
6  # More indentation included to distinguish this from the rest.  
7  def long_function_name(  
8      var_one, var_two, var_three,  
9      var_four):  
10     print(var_one)  
11  
12 # Hanging indents should add a level.  
13 foo = long_function_name(  
14     var_one, var_two,  
15     var_three, var_four)
```

```
1  #No:  
2  # Arguments on first line forbidden when not using vertical alignment.  
3  foo = long_function_name(var_one, var_two,  
4      var_three, var_four)  
5  
6  # Further indentation required as indentation is not distinguishable.  
7  def long_function_name(  
8      var_one, var_two, var_three,  
9      var_four):  
10     print(var_one)
```

 The 4-space rule is optional for continuation lines.

6.15.5 Blank lines and White Spaces

- Avoid extraneous white spaces immediately before the open parenthesis that starts the argument list of a function call:

```
1 | #Yes:  
2 | spam(1)
```

```
1 | #No:  
2 | spam (1)
```

- Surround top-level function and class definitions with two blank lines.
- Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).
- Use blank lines in functions, sparingly, to indicate logical sections.

6.16 Debugging

6.16.1 Editor

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don't. Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you. Also, don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case the program you are looking at in the text editor is not the same as the program you are running. Debugging can take a long time if you keep running the same, incorrect, program over and over! Make sure that the code you are looking at is the code you are running. If you're not sure, put something like `print('hello')` at the beginning of the program and run it again. If you don't see `hello`, you're not running the right program!

6.16.2 Traceback

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming, especially when there are many frames on the stack. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>> y = 6
      File "<stdin>", line 1
        y = 6
        ^
SyntaxError: invalid syntax
```

In this example, the problem is that the second line is indented by one space. But the error message points to `y`, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$. In Python, you might write something like this:

```
1 import math
2 signal_power = 9
3 noise_power = 10
4 ratio = signal_power / noise_power
5 decibels = 10 * math.log10(ratio)
6 print(decibels)
```

But when you run it, you get an error message³:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
OverflowError: math range error
```

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, because dividing two integers does floor division. The solution is to represent signal power and noise power with floating-point values.

In general, error messages tell you where the problem was discovered, but that is often not where it was caused.

6.17 Glossary

- 51 — **function**:. A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.
- 52 — **function definition**:. A statement that creates a new function, specifying its name, parameters, and the statements it executes.

³In Python 3.0, you no longer get an error message; the division operator performs floating-point division even with integer operands.

- **53 — function object:**. A value created by a function definition. The name of the function is a variable that refers to a function object.
- **54 — header:**. The first line of a function definition.
- **55 — body:**. The sequence of statements inside a function definition.
- **56 — parameter:**. A name used inside a function to refer to the value passed as an argument.
- **57 — function call:**. A statement that executes a function. It consists of the function name followed by an argument list.
- **58 — argument:**. A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.
- **59 — local variable:**. A variable defined inside a function. A local variable can only be used inside its function.
- **60 — return value:**. The result of a function. If a function call is used as an expression, the return value is the value of the expression.
- **61 — non-void function:**. A function that returns a value.
- **62 — void function:**. A function that doesn't return a value.
- **63 — module:**. A file that contains a collection of related functions and other definitions.

- **64 — import statement:**. A statement that reads a module file and creates a module object.
- **65 — module object:**. A value created by an `import` statement that provides access to the values defined in a module.
- **66 — dot notation:**. The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.
- **67 — composition:**. Using an expression as part of a larger expression, or a statement as part of a larger statement.
- **68 — flow of execution:**. The order in which statements are executed during a program run.
- **69 — stack diagram:**. A graphical representation of a stack of functions, their variables, and the values they refer to.
- **70 — frame:**. A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.
- **71 — traceback:**. A list of the functions that are executing, printed when an exception occurs.

6.18 Exercises

Exercise 6.3 Python provides a built-in function called `len` that returns the length of a string, so the value of `len('allen')` is 5.

Write a function named `right_justify` that takes a string named `s` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

```
>>> right_justify('allen')  
allen
```

Exercise 6.4 A function object is a value you can assign to a variable or pass as an argument. For example, `do_twice` is a function that takes a function object as an argument and calls it twice:

```
def do_twice(f):  
    f()  
    f()
```

Here's an example that uses `do_twice` to call a function named `print_spam` twice.

```
def print_spam():  
    print('spam')  
  
do_twice(print_spam)
```

1. Type this example into a script and test it.
2. Modify `do_twice` so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.
3. Write a more general version of `print_spam`, called `print_twice`, that takes a string as a parameter and prints it twice.
4. Use the modified version of `do_twice` to call `print_twice` twice, passing 'spam' as an argument.
5. Define a new function called `do_four` that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.

Exercise 6.5 This exercise^a can be done using only the statements and other features we have learned so far.

1. Write a function that draws a grid like the following:

```
+ - + - +
|   |   |
+ - + - +
|   |   |
+ - + - +
```

Hint: to print more than one value on a line, you can print a comma-separated sequence:

```
print('+', '-')
```

In order to build a string spanning several lines, we can use the string character '`\n`' which represents the newline character. The newline character can be embedded in a string, or can be concatenated using the `+` operator as shown in the following examples.

```
>>> print('line 1 \nline 2')
line 1
line 2
>>> print('line 1' + '\n' + 'line 2')
line 1
line 2
```

A `print` statement all by itself ends the current line and goes to the next line.

2. Use the previous function to draw a similar grid with four rows and four columns.

^aBased on an exercise in Oualline, *Practical C Programming, Third Edition*, O'Reilly (1997)

7. Code Documentation

7.1 Introduction

When you write code, you write it for two primary audiences: your end-users and developers (including yourself) in your project team. Both audiences are equally important.

There is a surprising amount of passionate debates within the developer community regarding the usefulness or desirability of comments in source code. Should source code include significant comments or none at all or something in-between?

To date, I haven't been able to find a study exploring the benefit (or lack of) of commenting code, and its economic impact (positive or negative). Similarly, I could not find any study exploring the benefits and drawbacks of different standards and approaches to documenting code. In the remainder of the chapter, I will present some best practices as well as several standards used in the Python community.

But first of all, we should have a look at the debate on the use of comments in code, and explore both side of the argument. In his article, Kunk [6] summarise his thoughts on both side of the argument. Kunk lists three points in favour of using comments

- **It's important to communicate what the code SHOULD be doing.**

Comments are required to communicate the intent of the source apart from its functionality. When reading source code, it is extremely difficult to be certain as to what the original author intended. Comments should clearly communicate the developer's intent and if needed, an explanation as to how the source code algorithm accomplishes that intent.

- **Comments show respect for the next developer to read your code.**

Source code should be written in the way that you yourself would most like to encounter unfamiliar code. It should be well-structured with a clear logic flow, and

be both efficient and effective. Comments enhance the readability of the code, so that its intent and approach are immediately apparent with a clean appearance.

- **Comments indicate potential problem areas to avoid.**

Comments should mention factors that may be of concern for the developer or tester. Boundary conditions, valid argument ranges, and corner cases are all important factors to mention in comments for the benefit of future developers and testers.

In his column, he also lists three point against the use of comments in source code:

- **Comments take time and have cost but do not influence runtime behaviour.**

Clearly, comments are not going to improve runtime performance, they are solely for the benefit of the developers. The creation of comment do take time, and may require additional maintenance when code is changed or re-factored. This obviously add an additional cost to the maintenance. Therefore, the comment must justify its cost by providing a benefit greater than its cost. Bad comments are an unnecessary cost.

- **Comments are not properly maintained.**

Under time pressure, comments are generally the last thing to be addressed (if at all) and may become out of sync with the current code.

- **Comments may simply be incorrect.**

Sometimes comments are written by developers that misunderstand the code and reinforce that misunderstanding through incorrect or misleading text. The feeling is that bad comments are worse than no comments.

Kunk concluded that he was generally in the camp in favour of documenting code. He added that comments have values and justifies the cost. However, he also warn that comments need to be maintained with the same diligence as executable source code.

In is column, Vogel [15] agrees on the first point from Kunk, that is comments should explain the "why" of the code and stop there. He extended the usefulness of comments to includes description of how inputs and outputs of a method/function are related, and a list of expected "side effects". What does he mean by "side effect"? A "side effect" is any result from executing the code that isn't reflected in the values that the code returns. For example, deleting a record, sorting a mutable object such as a `list` passed in the parameters.

Vogel also argues that comments (apart from end-user documentation) should be replaced by a ROC (Really Obvious Code) approach, supported by Test Driven Development (TDD) or at least a set of automated test suits, something that will be discussed later in the book.

So where do I stand on the issue of comments. I believe that comments are necessary, but one must strive to write good/useful comments. There are several reasons for this:

- Not all programmers can write really obvious code. In addition, judging whether code is obvious is a subjective call. What is obvious to one person might be cryptic for another. This is especially true if a novice programmer reads the code of an experienced developer. Mertz [11] illustrates this point brilliantly. "If you're like me, you've probably opened up old codebases and wondered to yourself, "What in the world was I thinking?" If you're having a problem reading your own code, imagine what your users or other developers are experiencing when they're trying

to use or contribute to your code.”

- Comments are not just for code. They can document important program information such as author, date, license, and copyright details. Although it should be noted this might be a mute point since version control systems are readily available.
- Comments can be place holders for future work. This is a useful way to create an outline for a large program. Some Integrated Development Environment (IDE) uses tag within comments to create to-do lists.

In summary, both authors agree that good comments are useful and bad comments should be avoided at all costs. So what are good and bad comments?

Finally, a key message to get from this introduction is perfectly summarised by a quote from "The Elements of Programming Style" [5]:

“ Don’t comment bad code – rewrite it. ”

Brian W. Kernighan and P. J. Plaugher

7.1.1 Why Documenting Your Code Is So Important

Conversely, I’m sure you’ve run into a situation where you wanted to do something in Python and found what looks like a great library that can get the job done. However, when you start using the library, you look for examples, write-ups, or even official documentation on how to do something specific and can’t immediately find the solution.

After searching, you come to realize that the documentation is lacking or even worse, missing entirely. This is a frustrating feeling that deters you from using the library, no matter how great or efficient the code is. Daniele Procida summarized this situation best:

“ It doesn’t matter how good your software is, because if the documentation is not good enough, people will not use it. ”

Daniele Procida

7.2 Basics of Commenting Code

In general, commenting is describing your code to/for developers. The intended main audience is the maintainers and developers of the code. In conjunction with well-written code, comments help to guide the reader to better understand your code and its purpose and design:

“ Code tells you how; Comments tell you why. ”

Jeff Atwood

The following section describes how and when to comment your code. Comments are created in Python using the hash sign (#) and should be brief statements no longer than a few sentences. Here’s a simple example:

```
1 | def hello_world():  
2 |     # A simple comment preceding a simple print statement  
3 |     print("Hello World")
```

According to PEP 8 [3], comments should have a maximum length of 72 characters. This is true even if your project changes the max line length to be greater than the recommended 80 characters. You should use two spaces after a sentence-ending period in multi- sentence comments, except after the final sentence. Paragraphs inside a block comment are separated by a line containing a single #.

In addition, comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!). Furthermore, when writing English, you should follow Strunk and White [14].

If a comment is going to be greater than the comment char limit, using multiple lines for the comment is appropriate:

```
1 | def hello_long_world():  
2 |     # A very long statement that just goes on and on and  
3 |     # on and on and never ends until after it's reached  
4 |     # the 80 char limit.  
5 |     print("Hellooooooooooooooooooooooooooooooooooooo World")
```

Commenting your code serves multiple purposes, including:

- Planning and Reviewing: When you are developing new portions of your code, it may be appropriate to first use comments as a way of planning or outlining that section of code. **Remember to remove these comments** once the actual coding has been implemented and reviewed/tested:

```
1 | # First step  
2 | # Second step  
3 | # Third step
```

This especially true if you are new at coding. This will help you formalise and organise your thoughts before you start coding.

- Code Description: Comments can be used to explain the intent of specific sections of code:

```
1 | # Attempt a connection based on previous settings. If  
2 | # unsuccessful, prompt user for new settings.
```

- Algorithmic Description: When algorithms are used, especially complicated ones, it can be useful to explain how the algorithm works or how it's implemented within your code. It may also be appropriate to describe why a specific algorithm was selected over another.

```
1 | # Using quick sort for performance gains
```

- Tagging: The use of tagging can be used to label specific sections of code where known issues or areas of improvement are located. Some examples are: BUG, FIXME, and TODO.

```
1 | # TODO: Add condition for when val is None
```

Comments to your code should be kept brief and focused. Avoid using long comments when possible. Additionally, you should use the following four essential rules as suggested by Jeff Atwood:

1. Keep comments as close to the code being documented as possible. Comments that aren't near their describing code are frustrating to the reader and easily missed when updates are made.
2. Don't use complex formatting (such as tables or ASCII figures). Complex formatting leads to distracting content and can be difficult to maintain over time.
3. Don't include redundant information. Assume the reader of the code has a basic understanding of programming principles and language syntax.
4. Design your code to comment itself. The easiest way to understand code is by reading it. When you design your code using clear, easy-to-understand concepts, the reader will be able to quickly conceptualize your intent.

Remember that comments are designed for the reader, including yourself, to help guide them in understanding the purpose and design of the software. To that end, Google [4] recommends that comments must never describe the code. You should Assume the person reading the code knows the language (though not what you're trying to do) better than you do. The place to have comments is in tricky parts of the code. Complicated operations get a few lines of comments before the operations commence. Non-obvious ones get comments at the end of the line. In addition, to improve legibility, these comments should be at least 2 spaces away from the code. Furthermore, comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style. An example is given below:

```
1 | # We use a weighted dictionary search to find out where i  
2 | # is in the array. We extrapolate position based on the  
3 | # largest num in the array and the array size and then do  
4 | # binary search to get the exact number.  
5 |  
6 | if i & (i-1) == 0: # True if i is 0 or a power of 2.
```

In [10], Marting has a rather negative view on comments (not API Documentation). In particular, he list a series of bad habit that should be avoided at all costs. Here is a sub-sample of the thing to be avoided, for the full list please read chapter 4 of [10].

- **Inaccurate comments** are far worse than no comments at all. They delude and mislead.
- **Comments Do Not Make Up for Bad Code.** Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spending time writing comments to explain your code, you should spend that time refactoring your code.
- **Noise comments** state the obvious and provide no new information. For example:

```
1 | output = {} # The output dictionary returned by the
2 |           # function
```

A better comment would have been:

```
1 | # A dictionary containing the time of each athlete
2 | # taking part in the 200 metre race.
3 | # Key: str
4 | #     The athlete's name
5 | # Value: float
6 | #     The athlete's time in seconds during that race
7 |
8 | race_time_200m = {}
```

- **Commented-Out code.** If you do not need that code delete it. Others maintaining that piece of code would not know why it is commented, if it is important, or is it here for historical reasons. Overtime, this type of comments accumulate and clutter the source code.

7.3 Documenting Code via Python Docstring

Documenting code is describing its use and functionality to your users. While it may be helpful in the development process, the main intended audience is the users.

“It doesn’t matter how good your software is, because if the documentation is not good enough, people will not use it.”

— Daniele Procida

Python uses documentation string (docstring) to document code. A docstring is a string that is the first statement in a package, module, class or function. The docstring is a special attribute of the object (`object.__doc__`) and, for consistency, is surrounded by triple double quotes (per PEP 257 [2]). These strings can be extracted automatically by `pydoc` (try running `pydoc` on your module to see how it looks.).

A docstring should be organized as a summary line (one physical line) terminated by a period, question mark, or exclamation point, followed by a blank line, followed by the rest of the docstring starting at the same cursor position as the first quote of the first line (see example below).

```
1  """This is the form of a docstring.  
2  
3  It can be spread over several lines.  
4  
5  """
```

Although it may seem obvious it is worth reiterating the Google Python style guide [4] recommendation to pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

The selection of the docstring format is up to you, but you should stick with the same format throughout your document/project.

7.3.1 Google Comments and Docstrings Standard

This section is directly taken from Google's Python style guide [4]. Some of the elements have been omitted as they were not relevant to the module taught, however the interested reader should have a look at the full documentation.

Functions and Methods

In this section, "function" means a method or a function. A function must have a docstring, unless it meets all of the following criteria:

- not externally visible
- very short
- obvious

A docstring should give enough information to write a call to the function without reading the function's code. The docstring should be descriptive (`"""Fetches rows from a Bigtable."""`) rather than imperative (`"""Fetch rows from a Bigtable."""`). A docstring should describe the function's calling syntax and its semantics, **not its implementation**. For tricky code, comments alongside the code are more appropriate than using docstrings.

Certain aspects of a function should be documented in special sections, listed below. Each section begins with a heading line, which ends with a colon. Sections should be indented two spaces, except for the heading.

- **Args:** List each parameter by name. A description should follow the name, and be separated by a colon and a space. If the description is too long to fit on a single 80-character line, use a hanging indent of 2 or 4 spaces (be consistent with the rest of the file). The description should include required type(s) if the code does not contain a corresponding type annotation. If a function accepts `*foo` (variable length argument lists) and/or `**bar` (arbitrary keyword arguments), they should be listed as `*foo` and `**bar`.
- **Returns:** Describe the type and semantics of the return value. If the function only returns `None`, this section is not required. It may also be omitted if the docstring starts with `Returns` or `Yields` (e.g. `"""Returns row from Bigtable as a tuple of strings."""`) and the opening sentence is sufficient to describe return value.

- **Raises:** List all exceptions that are relevant to the interface.

An example is given below:

```

1  def fetch_bigtable_rows(
2      big_table,
3      keys,
4      other_silly_variable=None):
5      """Fetches rows from a Bigtable.
6
7      Retrieves rows pertaining to the given keys from the
8      Table instance represented by big_table. Silly things
9      may happen if other_silly_variable is not None.
10
11     Args:
12         big_table: An open Bigtable Table instance.
13         keys: A sequence of strings representing the key
14         of each table row to fetch.
15         other_silly_variable: Another optional variable,
16         hat has a much longer name than the other
17         args, and which does nothing.
18
19     Returns:
20         A dict mapping keys to the corresponding table
21         row data fetched. Each row is represented as a
22         tuple of strings. For example:
23
24         {'Serak': ('Rigel VII', 'Preparer'),
25         'Zim': ('Irk', 'Invader'),
26         'Lrrr': ('Omicron Persei 8', 'Emperor')}
27
28         If a key from the keys argument is missing from
29         the dictionary, then that row was not found in
30         the table.
31
32     Raises:
33         IOError: An error occurred accessing the
34         bigtable.Table object.
35     """
```

7.3.2 NumPy Docstring Standard

NumPy, SciPy, and the scikits follow a common convention for docstrings in order to maintain consistency. The guide [12] describes the current community consensus for such a standard.

Numpy docstring standard uses re-structured text (reST) syntax and is rendered using Sphinx (a pre-processor that understands the particular documentation style used).

A selected list of sections is given below, and it will provide a good starting point for documenting your code. A complete list of section can be found in the NumPyDoc style guide [12].

- The length of docstring lines should be kept to 75 characters to facilitate reading the docstrings in text terminals.
- The docstring consists of a number of sections separated by headings (except for the deprecation warning). Each heading should be underlined in hyphens, and the section ordering should be consistent with the description below.

The sections of a function's docstring are:

1. **Short summary:** A one-line summary that does not use variable names or the function name. The function signature is normally found by introspection and displayed by the help function.
2. **Extended Summary:** A few sentences giving an extended description. This section should be used to clarify functionality, not to discuss implementation detail or background theory, which should rather be explored in the Notes section below. You may refer to the parameters and the function name, but parameter descriptions still belong in the Parameters section.
3. **Parameters:** Description of the function arguments, keywords and their respective types. Enclose variables in single backticks. The colon must be preceded by a space, or omitted if the type is absent. For the parameter types, be as precise as possible. Optional keyword parameters have default values, which are displayed as part of the function signature. When a parameter can only assume one of a fixed set of values, those values can be listed in braces, with the default appearing first.

```

1  """
2  Parameters
3  -----
4  x : type
5      Description of parameter 'x'.
6  y
7      Description of parameter 'y' (with type not
8      specified)
9  z : type, optional
10     Description of optional parameter 'z' (the
11     default is -1, which a description of what
12     the default value implies).
13  d : {'Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa', 'Su'}
14     Description of 'd'
15  """

```

4. **Returns:** Explanation of the returned values and their types. Similar to the Parameters section, except the name of each return value is optional. The type of each return value is always required. If both the name and type are specified, the Returns section takes the same form as the Parameters section.

```

1  """
2  Returns
3  -----
4  err_code : int
5      Non-zero value indicates error code, or zero on
6      success.
7  err_msg : str or None

```



```
8         Human readable error message, or None on  
9         success.  
10        """
```

5. **Raises:** An optional section detailing which errors get raised and under what conditions. This section should be used judiciously, that is, only for errors that are non-obvious or have a large chance of getting raised.

7.3.3 EpyText Docstring Standard

Epytext [9] is a lightweight markup language for Python docstrings. The epytext markup language is used by Epydoc [7] to parse docstrings and create structured API documentation. Exploring the complete Epytext language is out of the scope of this book, and in this section we will only cover a small sub-sample of the language properties.

Epytext markup is broken up into the following categories:

- Block Structure divides the docstring into nested blocks of text, such as paragraphs and lists.
 - Basic Blocks are the basic unit of block structure.
 - Hierarchical blocks represent the nesting structure of the docstring.
- Inline Markup marks regions of text within a basic block with properties, such as italics and hyperlinks.

Block Structure

Block structure is encoded using indentation, blank lines, and a handful of special character sequences.

Indentation is used to encode the nesting structure of hierarchical blocks. The indentation of a line is defined as the number of leading spaces on that line; and the indentation of a block is typically the indentation of its first line. Blank lines are used to separate blocks. A blank line is a line that only contains whitespace. Special character sequences are used to mark the beginnings of some blocks. For example, '-' is used as a bullet for unordered list items, and '>' is used to mark doctest blocks. The following sections describe how to use each type of block structure.

Paragraphs

A paragraph is the simplest type of basic block. It consists of one or more lines of text. Paragraphs must be left justified (i.e., every line must have the same indentation). The following example illustrates how paragraphs can be used:

```
1  def example():  
2      """  
3          This is a paragraph. Paragraphs can  
4          span multiple lines, and can contain  
5          {inline markup}.  
6      """
```

```
7 | This is another paragraph. Paragraphs  
8 | are separated by blank lines.  
9 | """
```

Fields

Fields are used to describe specific properties of a documented object. For example, fields can be used to define the parameters and return value of a function; the instance variables of a class; and the author of a module. Each field is marked by a field tag, which consist of an at sign ('@') followed by a field name, optionally followed by a space and a field argument, followed by a colon (':'). For example, '@return:' and '@param x:' are field tags.

Fields can contain paragraphs, lists, literal blocks, and doctest blocks (described in [9]). All of the blocks contained by a field must all have equal indentation, and that indentation must be greater than or equal to the indentation of the field's tag. If the first contained block is a paragraph, it may appear on the same line as the field tag, separated from the field tag by one or more spaces. All other block types must follow on separate lines.

Fields must be placed at the end of the docstring, after the description of the object. Fields may be included in any order.

Fields do not need to be separated from other blocks by a blank line. Any line that begins with a field tag followed by a space or newline is considered a field.

The following example illustrates how fields can be used:

- **@param p:** ...
A description of the parameter p for a function or method.
- **@type p:** ...
The expected type for the parameter p.
- **@return:** ...
The return value for a function or method.
- **@rtype:** ...
The type of the return value for a function or method.
- **@keyword p:** ...
A description of the keyword parameter p.
- **@raise e:** ...
A description of the circumstances under which a function or method raises exception e.

@param fields should be used to document any explicit parameter (including the keyword parameter). @keyword fields should only be used for non-explicit keyword parameters:

```
1 | def plant(seed, *tools, **options):  
2 |     """  
3 |         @param seed: The seed that should be planted.  
4 |         @param tools: Tools that should be used to plant the seed.  
5 |         @param options: Any extra options for the planting.  
6 |
```

```

7 |         @keyword dig_deep: Plant the seed deep under ground.
8 |         @keyword soak: Soak the seed before planting it.
9 |         """

```

For a complete list of fields that are supported by epydoc, see the epydoc fields page [8].

7.3.4 Examples

The following are examples of each type to give you an idea of how each documentation format looks.

Google Docstrings Example

```

1 | """Gets and prints the spreadsheet's header columns
2 |
3 | Parameters:
4 |     file_loc (str): The file location of the spreadsheet
5 |     print_cols (bool): A flag used to print the columns to the
6 |         console (default is False)
7 |
8 | Returns:
9 |     list: a list of strings representing the header columns
10 | """

```

NumPy/SciPy Docstrings Example

```

1 | """Gets and prints the spreadsheet's header columns
2 |
3 | Parameters
4 | -----
5 | file_loc : str
6 |     The file location of the spreadsheet
7 | print_cols : bool, optional
8 |     A flag used to print the columns to the console (default
9 |     is False)
10 |
11 | Returns
12 | -----
13 | list
14 |     a list of strings representing the header columns
15 | """

```

Epytext Example

```

1 | """Gets and prints the spreadsheet's header columns
2 |

```

```
3  @type file_loc: str
4  @param file_loc: The file location of the spreadsheet
5  @type print_cols: bool
6  @param print_cols: A flag used to print the columns to the
7      console (default is False)
8  @rtype: list
9  @returns: a list of strings representing the header columns
10  """
```

8. Recursion

Rather than using `while` or `for` loops, we can implement repetition through recursion. It is legal for one function to call another; it is also legal for a function to call itself, and in that case the function is said to be recursive, and the process is called recursion. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do.

For example, look at the following function:

```
1  def countdown(n):  
2      if n <= 0:  
3          print('Blastoff!')  
4      else:  
5          print(n)  
6          countdown(n-1)
```

If `n` is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs `n` and then calls a function named `countdown`—itself—passing `n - 1` as an argument.

What happens if we call this function like this?

```
| >>> countdown(3)
```

The execution of `countdown` begins with `n=3`, and since `n` is greater than 0, it outputs the value 3, and then calls itself..

The execution of `countdown` begins with `n=2`, and since `n` is greater than 0, it outputs the value 2, and then calls itself..

The execution of `countdown` begins with `n=1`, and since `n` is greater than 0, it outputs the value 1, and then calls itself..

The execution of `countdown` begins with `n=0`, and since `n` is not greater than 0, it outputs the word, “Blastoff!” and then returns.

The `countdown` that got `n=1` returns.


The `countdown` that got `n=2` returns.

The `countdown` that got `n=3` returns.

And then you’re back in `__main__`. So, the total output looks like this:

```
| 3
| 2
| 1
| Blastoff!
```

A recursive function calls itself, usually on a smaller argument. The sequence of calls must eventually terminate in a **base case** that does not generate a new **recursive call**. For the function `countdown`, line 2-3 constitute the base case, whereas line 6 is the recursive call.

 A recursive function may have multiple base cases (at least one), and multiple recursive calls.

As another example, we can write a function that prints a string `n` times.

```
1 | def print_n(s, n):
2 |     if n <= 0:
3 |         return
4 |     print(s)
5 |     print_n(s, n-1)
```

If `n <= 0` the `return` statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

The rest of the function is similar to `countdown`: if `n` is greater than 0, it displays `s` and then calls itself to display `s` $n - 1$ additional times. So the number of lines of output is $1 + (n - 1)$, which adds up to `n`.

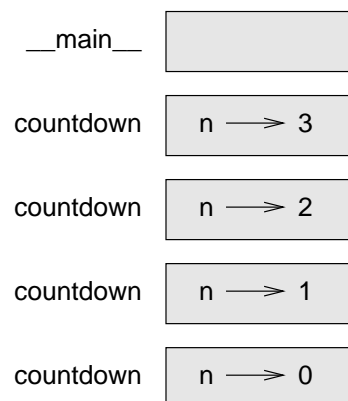
For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

8.1 Stack diagrams for recursive functions

In Section 6.12, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a new function frame, which contains the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

This figure shows a stack diagram for `countdown` called with `n = 3`:



As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it.

The four `countdown` frames have different values for the parameter `n`. The bottom of the stack, where `n=0`, is called the **base case**. It does not make a recursive call, so there are no more frames.

Exercise 8.1 Draw a stack diagram for `print_n` called with `s = 'Hello'` and `n=2`. ■

8.2 Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
1  def recurse():
2      recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
>>> recurse()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    recurse()
  File "chapter_recursion.py", line 19, in recurse
    recurse()
  File "chapter_recursion.py", line 19, in recurse
    recurse()
  File "chapter_recursion.py", line 19, in recurse
    recurse()
[Previous line repeated 990 more times]
```


This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 `recurse` frames on the stack!

8.3 Requirements

There are two general requirements for recursive functions:

1. The base case(s) need to be checked first, before the recursive call.
2. The argument(s) of the recursive call(s) must be smaller (in other word get closer to a base case); otherwise we will end up with an infinite recursion as the base case(s) will never be reached.

These two requirements together ensure that the recursion stops and a value is returned. Otherwise, we will encounter an infinite recursion error seen in Section 8.2.

 In fact, there is a limit to what the interpreter can compute recursively, because it needs to keep track of all of the ongoing, partial computations in the call stack as shown in Section 8.1. Despite our function `count down` being correctly implemented, calling `count down(1000)` will result in the same error as an infinite recursion. Other programming languages like 'Scheme', 'Common Lisp', and 'Haskell' are better suited than Python for recursive programming.

8.4 Glossary

- 72 — **temporary variable**:: A variable used to store an intermediate value in a complex calculation.
- 73 — **dead code**:: Part of a program that can never be executed, often because it appears after a `return` statement.
- 74 — **None**:: A special value returned by functions that have no return statement or a return statement without an argument.
- 75 — **incremental development**:: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.
- 76 — **scaffolding**:: Code that is used during program development but is not part of the final version.
- 77 — **guardian**:: A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

8.5 Exercises

Exercise 8.2 Draw a stack diagram for the following program. What does the program print?

```
def b(z):  
    prod = a(z, z)  
    print z, prod  
    return prod  
  
def a(x, y):  
    x = x + 1  
    return x * y  
  
def c(x, y, z):  
    sum = x + y + z  
    pow = b(sum)**2  
    return pow  
  
x = 1  
y = x + 1  
print c(x, y+3, x+y)
```

Exercise 8.3 Write a function called `do_n` that takes a function object and a number, `n`, as arguments, and that calls the given function `n` times.

Exercise 8.4 The Ackermann function, $A(m, n)$, is defined^a:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases} \quad (8.1)$$

Write a function named `ack` that evaluates Ackerman's function. Use your function to evaluate `ack(3, 4)`, which should be 125. What happens for larger values of m and n ?

^aSee wikipedia.org/wiki/Ackermann_function.

Exercise 8.5 A palindrome is a word that is spelled the same backward and forward, like “noon” and “redivider”. Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

The following are functions that take a string argument and return the first, last, and middle letters:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

We'll see how they work in Chapter 9.

1. Type these functions into a file named `palindrome.py` and test them out. What happens if you call `middle` with a string with two letters? One letter? What about the empty string, which is written `''` and contains no letters?
2. Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise. Remember that you can use the built-in function `len` to check the length of a string.

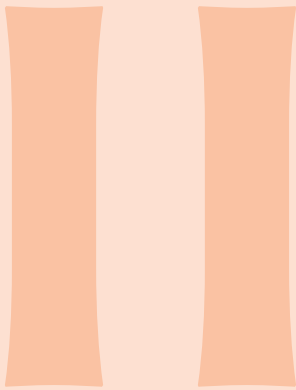
Exercise 8.6 A number, a , is a power of b if it is divisible by b and a/b is a power of b . Write a function called `is_power` that takes parameters a and b and returns `True` if a is a power of b .

Exercise 8.7 The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder^a.

One way to find the GCD of two numbers is Euclid's algorithm, which is based on the observation that if r is the remainder when a is divided by b , then $\text{gcd}(a, b) = \text{gcd}(b, r)$. As a base case, we can consider $\text{gcd}(a, 0) = a$.

Write a function called `gcd` that takes parameters `a` and `b` and returns their greatest common divisor. If you need help, see wikipedia.org/wiki/Euclidean_algorithm.

^aThis exercise is based on an example from Abelson and Sussman's *Structure and Interpretation of Computer Programs*. ■



Part II - Built-in Data Structures

9	Strings	85
10	Lists	97
11	Dictionaries	113
12	Tuples	123
13	Files	135

9. Strings

9.1 A string is a sequence

A string is a **sequence** of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement selects character number 1 from `fruit` and assigns it to `letter`. The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name). But you might not get what you expect:

```
>>> print(letter)
a
```

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th (“two-eth”) letter.

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]  
TypeError: string indices must be integers
```

9.2 Function len

`len` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'  
>>> len(fruit)  
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)  
>>> last = fruit[length]  
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
>>> last = fruit[length-1]  
>>> print(last)  
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

9.3 Traversal with a for loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end.

This pattern of processing is called a **traversal**. One way to write a traversal is with a `while` loop:

```
1 | index = 0
2 | while index < len(fruit):
3 |     letter = fruit[index]
4 |     print(letter)
5 |     index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit) - 1`, which is the last character in the string.

Exercise 9.1 Write a function that takes a string as an argument and displays the letters backward, one per line. ■

Another way to write a traversal is with a `for` loop:

```
1 | for char in fruit:
2 |     print(char)
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a `for` loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
1 | prefixes = 'JKLMNOPQ'
2 | suffix = 'ack'
3 |
4 | for letter in prefixes:
5 |     print(letter + suffix)
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Quack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled.

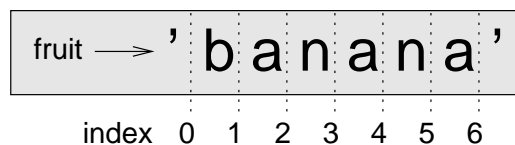
Exercise 9.2 Modify the program to fix this error. ■

9.4 String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

The operator `[n:m]` returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last. This behavior is counter intuitive, but it might help to imagine the indices pointing *between* the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:2]
'bana'
>>> fruit[2:]
'nana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Exercise 9.3 Given that `fruit` is a string, what does `fruit[:]` mean? ■

9.5 Strings are immutable

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence. The reason for the error is that strings are **immutable**, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

9.6 Searching

What does the following function do?

```
1 def find(word, letter):
2     index = 0
3     while index < len(word):
4         if word[index] == letter:
5             return index
6         index = index + 1
7     return -1
```

In a sense, `find` is the opposite of the `[]` operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns `-1`. This is the first example we have seen of a `return` statement inside a loop. If `word[index] == letter`, the function breaks out of the loop and returns immediately. If the character doesn’t appear in the string, the program exits the loop normally and returns `-1`. This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a **search**.

Exercise 9.4 Modify `find` so that it has a third parameter, the index in `word` where it should start looking. ■

9.7 Looping and counting

The following program counts the number of times the letter a appears in a string:

```
1 word = 'banana'
2 count = 0
3 for letter in word:
4     if letter == 'a':
5         count = count + 1
6 print(count)
```

This program demonstrates another pattern of computation called a **counter**. The variable `count` is initialized to 0 and then incremented each time an a is found. When the loop exits, `count` contains the result—the total number of a’s.

Exercise 9.5 Encapsulate this code in a function named `count`, and generalize it so that it accepts the string and the letter as arguments. ■

Exercise 9.6 Rewrite this function so that instead of traversing the string, it uses the three-parameter version of `find` from the previous section. ■

9.8 string methods

A **method** is similar to a function—it takes arguments and may return a value—but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters: Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on the `word`. As it turns out, there is a string method named `find` that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter. Actually, the `find` method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

And as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from 1 to 2 (not including 2).

Exercise 9.7 There is a string method called `count` that is similar to the function in the previous exercise. Read the documentation of this method and write an invocation that counts the number of `a` in `'banana'`. ■

9.9 The in operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'ana' in 'banana'
True
>>> 'rama' in 'banana'
False
```

For example, the following function prints all the letters from `word1` that also appear in `word2`:

```
1 | def in_both(word1, word2):  
2 |     for letter in word1:  
3 |         if letter in word2:  
4 |             print(letter)
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Here’s what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges')  
a  
e  
s
```

9.10 String comparison

The relational operators work on strings. To see if two strings are equal:

```
1 | if word == 'banana':  
2 |     print('All right, bananas.')
```

Other relational operations are useful for putting words in alphabetical order:

```
1 | if word < 'banana':  
2 |     print('Your word,' + word + ', comes before banana.')
```

```
3 | elif word > 'banana':  
4 |     print('Your word,' + word + ', comes after banana.')
```

```
5 | else:  
6 |     print('All right, bananas.')
```

9.11 Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return `True` if one of the words is the reverse of the other, but it contains two errors:

```
1  def is_reverse(word1, word2):
2      if len(word1) != len(word2):
3          return False
4
5      i = 0
6      j = len(word2)
7
8      while j > 0:
9          if word1[i] != word2[j]:
10             return False
11             i = i+1
12             j = j-1
13
14     return True
```

The first `if` statement checks whether the words are the same length. If not, we can return `False` immediately and then, for the rest of the function, we can assume that the words are the same length. This is an example of the guardian pattern in Section ??.

`i` and `j` are indices: `i` traverses `word1` forward while `j` traverses `word2` backward. If we find two letters that don't match, we can return `False` immediately. If we get through the whole loop and all the letters match, we return `True`.

If we test this function with the words “pots” and “stop”, we expect the return value `True`, but we get an `IndexError`:

```
>>> is_reverse('pots', 'stop')
...
File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

```
1      while j > 0:
2          print(i, j)          # print here
3
4          if word1[i] != word2[j]:
5              return False
6          i = i+1
7          j = j-1
```

Now when I run the program again, I get more information:

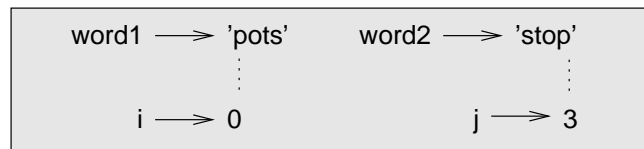
```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

The first time through the loop, the value of `j` is 4, which is out of range for the string `'pots'`. The index of the last character is 3, so the initial value for `j` should be `len(word2) - 1`.

If I fix that error and run the program again, I get:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for `is_reverse` looks like this:



I took a little license by arranging the variables in the frame and adding dotted lines to show that the values of `i` and `j` indicate characters in `word1` and `word2`.

Exercise 9.8 Starting with this diagram, execute the program on paper, changing the values of `i` and `j` during each iteration. Find and fix the second error in this function.

■

9.12 Glossary

- **78 — object:** Something a variable can refer to. For now, you can use “object” and “value” interchangeably.
- **79 — sequence:** An ordered set; that is, a set of values where each value is identified by an integer index.
- **80 — item:** One of the values in a sequence.
- **81 — index:** An integer value used to select an item in a sequence, such as a character in a string.
- **82 — slice:** A part of a string specified by a range of indices.
- **83 — empty string:** A string with no characters and length 0, represented by two quotation marks.
- **84 — immutable:** The property of a sequence whose items cannot be assigned.
- **85 — traverse:** To iterate through the items in a sequence, performing a similar operation on each.

- **86 — search:**. A pattern of traversal that stops when it finds what it is looking for.
- **87 — counter:**. A variable used to count something, usually initialized to zero and then incremented.
- **88 — method:**. A function that is associated with an object and called using dot notation.
- **89 — invocation:**. A statement that calls a method.

9.13 Exercises

Exercise 9.9 A string slice can take a third index that specifies the “step size;” that is, the number of spaces between successive characters. A step size of 2 means every other character; 3 means every third, etc.

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

A step size of -1 goes through the word backwards, so the slice `[::-1]` generates a reversed string. Use this idiom to write a one-line version of `is_palindrome` from Exercise 13.11. ■

Exercise 9.10 ROT13 is a weak form of encryption that involves “rotating” each letter in a word by 13 places^a. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so 'A' shifted by 3 is 'D' and 'Z' shifted by 1 is 'A'.

Write a function called `rotate_word` that takes a string and an integer as parameters, and that returns a new string that contains the letters from the original string “rotated” by the given amount.

For example, “cheer” rotated by 7 is “jolly” and “melon” rotated by -10 is “cubed”. You might want to use the built-in functions `ord`, which converts a character to a numeric code, and `chr`, which converts numeric codes to characters. ■

^aSee wikipedia.org/wiki/ROT13.

Exercise 9.11 Read the documentation of the string methods at `docs.python.org/lib/string-methods.html`. You might want to experiment with some of them to make sure you understand how they work. `strip` and `replace` are particularly useful. The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`, the brackets indicate optional arguments. So `sub` is required, but `start` is optional, and if you include `start`, then `end` is optional. ■

Exercise 9.12 The following functions are all *intended* to check whether a string contains any lowercase letters, but at least some of them are wrong. For each function, describe what the function actually does (assuming that the parameter is a string).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False
```

```
def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag
```

```
def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag
```

```
def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

 ■

10. Lists

10.1 A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
[10, 20, 30, 40]  
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**. A list that contains no elements is called an empty list; you can create one with empty brackets, []. As you might expect, you can assign list values to variables:

```
1 >>> cheeses = ['Cheddar', 'Edam', 'Gouda']  
2 >>> numbers = [17, 123]  
3 >>> empty = []  
4 >>> print(cheeses, numbers, empty)  
5 ['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

10.2 Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

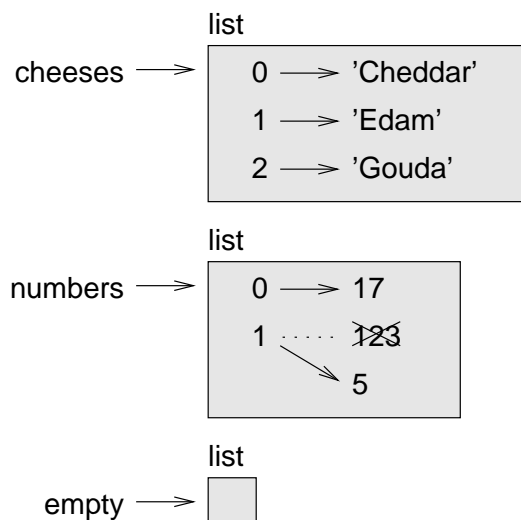
```
1 | >>> print(cheeses[0])
2 | Cheddar
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
1 | >>> numbers = [17, 123]
2 | >>> numbers[1] = 5
3 | >>> print(numbers)
4 | [17, 5]
```

The one-eth element of `numbers`, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a **mapping**; each index “maps to” one of the elements. Here is a state diagram showing `cheeses`, `numbers` and `empty`:



Lists are represented by boxes with the word “list” outside and the elements of the list inside. `cheeses` refers to a list with three elements indexed 0, 1 and 2. `numbers` contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. `empty` refers to a list with no elements. List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The `in` operator also works on lists.

```
1 >>> cheeses = ['Cheddar', 'Edam', 'Gouda']
2 >>> 'Edam' in cheeses
3 True
4 >>> 'Brie' in cheeses
5 False
```

10.3 Traversing a list

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
1 for cheese in cheeses:
2     print(cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
1 for i in range(len(numbers)):
2     numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n - 1$, where n is the length of the list. Each time through the loop `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A `for` loop over an empty list never executes the body:

```
1 for x in []:
2     print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 List operations

The + operator concatenates lists:

```
1 >>> a = [1, 2, 3]
2 >>> b = [4, 5, 6]
3 >>> c = a + b
4 >>> print(c)
5 [1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
1 >>> [0] * 4
2 [0, 0, 0, 0]
3 >>> [1, 2, 3] * 3
4 [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

10.5 List slices

The slice operator also works on lists:

```
1 >>> t = ['a', 'b', 'c', 'd', 'e', 'f']
2 >>> t[1:3]
3 ['b', 'c']
4 >>> t[:4]
5 ['a', 'b', 'c', 'd']
6 >>> t[3:]
7 ['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
1 >>> t[:]
2 ['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
1 >>> t = ['a', 'b', 'c', 'd', 'e', 'f']
2 >>> t[1:3] = ['x', 'y']
3 >>> print(t)
4 ['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 List methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
1 >>> t = ['a', 'b', 'c']
2 >>> t.append('d')
3 >>> print(t)
4 ['a', 'b', 'c', 'd']
```

`extend` takes a list as an argument and appends all of the elements:

```
1 >>> t1 = ['a', 'b', 'c']
2 >>> t2 = ['d', 'e']
3 >>> t1.extend(t2)
4 >>> print(t1)
5 ['a', 'b', 'c', 'd', 'e']
```

This example leaves `t2` unmodified.

`sort` arranges the elements of the list from low to high:

```
1 >>> t = ['d', 'c', 'e', 'b', 'a']
2 >>> t.sort()
3 >>> print(t)
4 ['a', 'b', 'c', 'd', 'e']
```

List methods are all void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

10.7 Map, filter and reduce

To add up all the numbers in a list, you can use a loop like this:

```
1 | def add_all(a_list):
2 |     total = 0
3 |     for val in a_list:
4 |         total += val
5 |     return total
```

`total` is initialized to 0. Each time through the loop, `x` gets one element from the list. The `+=` operator provides a short way to update a variable. This **augmented assignment statement**:

```
1 |     total += val
```

is equivalent to:

```
1 |     total = total + val
```

As the loop executes, `total` accumulates the sum of the elements; a variable used this way is sometimes called an **accumulator**. Adding up the elements of a list is such a common operation that Python provides it as a built-in function, `sum`:

```
1 | >>> t = [1, 2, 3]
2 | >>> sum(t)
3 | 6
```

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce**.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
1 | def capitalize_all(word):
2 |     result = []
3 |     for letter in word:
4 |         result.append(letter.capitalize())
5 |     return result
```

`result` is initialized with an empty list; each time through the loop, we append the next element. So `result` is another kind of accumulator. An operation like `capitalize_all` is sometimes called a **map** because it “maps” a function (in this case the method `capitalize`) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
1  def only_upper(word):
2      result = []
3      for letter in word:
4          if letter.isupper():
5              result.append(letter)
6      return result
```

`isupper` is a string method that returns `True` if the string contains only upper case letters. An operation like `only_upper` is called a **filter** because it selects some of the elements and filters out the others.

Most common list operations can be expressed as a combination of map, filter and reduce. Because these operations are so common, Python provides language features to support them, including the built-in function `map` and an operator called a “list comprehension.”

Exercise 10.1 Write a function that takes a list of numbers and returns the cumulative sum; that is, a new list where the i th element is the sum of the first $i + 1$ elements from the original list. For example, the cumulative sum of `[1, 2, 3]` is `[1, 3, 6]`. ■

10.8 Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use `pop`:

```
1  >>> t = ['a', 'b', 'c']
2  >>> x = t.pop(1)
3  >>> print(t)
4  ['a', 'c']
5  >>> print(x)
6  b
```

`pop` modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the `del` operator:

```
1  >>> t = ['a', 'b', 'c']
2  >>> del t[1]
3  >>> print(t)
4  ['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
1 >>> t = ['a', 'b', 'c']
2 >>> t.remove('b')
3 >>> print(t)
4 ['a', 'c']
```

The return value from `remove` is `None`.

To remove more than one element, you can use `del` with a slice index:

```
1 >>> t = ['a', 'b', 'c', 'd', 'e', 'f']
2 >>> del t[1:5]
3 >>> print(t)
4 ['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

10.9 Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use `list`:

```
1 >>> s = 'spam'
2 >>> t = list(s)
3 >>> print(t)
4 ['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid `l` because it looks too much like `1`. So that's why I use `t`.

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```
1 >>> s = 'pining for the fjords'
2 >>> t = s.split()
3 >>> print(t)
4 ['pining', 'for', 'the', 'fjords']
```


An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
1 >>> s = 'spam-spam-spam'
2 >>> delimiter = '-'
3 >>> s.split(delimiter)
4 ['spam', 'spam', 'spam']
```

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. `join` is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
1 >>> t = ['pining', 'for', 'the', 'fjords']
2 >>> delimiter = '-'
3 >>> delimiter.join(t)
4 'pining-for-the-fjords'
```

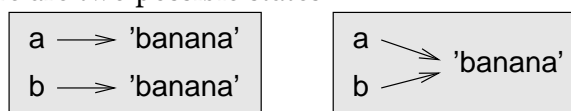
In this case the delimiter is a space character, so `join` puts a ' ' between words. To concatenate strings without delimiters, you can use the empty string, '', as a delimiter.

10.10 Objects and values

If we execute these assignment statements:

```
1 a = 'banana'
2 b = 'banana'
```

We know that `a` and `b` both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:



In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the `is` operator.

```
1 >>> a = 'banana'
2 >>> b = 'banana'
3 >>> a is b
4 True
```

In this example, Python only created one string object, and both `a` and `b` refer to it.

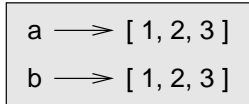
But when you create two lists, you get two objects:

```

1  >>> a = [1, 2, 3]
2  >>> b = [1, 2, 3]
3  >>> a is b
4  False

```

So the state diagram looks like this:



In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using “object” and “value” interchangeably, but it is more precise to say that an object has a value. If you execute `[1, 2, 3]`, you get a list object whose value is a sequence of integers. If another list has the same elements, we say it has the same value, but it is not the same object.

10.11 Aliasing

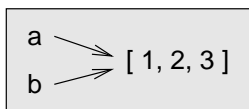
If `a` refers to an object and you assign `b = a`, then both variables refer to the same object:

```

1  >>> a = [1, 2, 3]
2  >>> b = a
3  >>> b is a
4  True

```

The state diagram looks like this:



The association of a variable with an object is called a **reference**. In this example, there are two references to the same object. An object with more than one reference has more than one name, so we say that the object is **aliased**.

If the aliased object is mutable, changes made with one alias affect the other:

```

5  >>> b[0] = 17
6  >>> print(a)
7  [17, 2, 3]

```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
1 | a = 'banana'
2 | b = 'banana'
```

It almost never makes a difference whether `a` and `b` refer to the same string or not.

10.12 List arguments

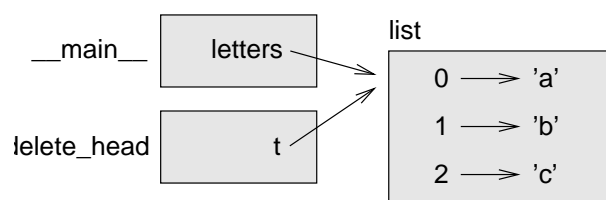
When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
1 | def delete_head(lst):
2 |     del lst[0]
```

Here's how it is used:

```
1 | >>> letters = ['a', 'b', 'c']
2 | >>> delete_head(letters)
3 | >>> print (letters)
4 | ['b', 'c']
```

The parameter `lst` and the variable `letters` are aliases for the same object. The stack diagram looks like this:



Since the list is shared by two frames, I drew it between them.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list:

```
1 >>> t1 = [1, 2]
2 >>> t2 = t1.append(3)
3 >>> print(t1)
4 [1, 2, 3]
5 >>> print(t2)
6 None
7 >>> t1 = [1, 2]
8 >>> t3 = t1 + [3]
9 >>> print(t3)
10 [1, 2, 3]
11 >>> t2 is t3
12 False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
1 def bad_delete_head(t):
2     t = t[1:]           # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
1 def tail(t):
2     return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
1 >>> letters = ['a', 'b', 'c']
2 >>> rest = tail(letters)
3 >>> print(rest)
4 ['b', 'c']
```

Exercise 10.2 Write a function called `chop` that takes a list and modifies it, removing the first and last elements, and returns `None`. Then write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements.

10.13 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

If you are used to writing string code like this:

```
1 | word = word.strip()
```

It is tempting to write list code like this:

```
1 | t = t.sort()           # WRONG!
```

Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at `docs.python.org/lib/typesseq.html`. The methods and operators that only apply to mutable sequences are documented at `docs.python.org/lib/typesseq-mutable.html`.

2. Pick an idiom and stick with it.

Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment. To add an element, you can use the `append` method or the `+` operator. Assuming that `t` is a list and `x` is a list element, these are right:

```
1 | t.append(x)
2 | t = t + [x]
```

And these are wrong:

```
1 | t.append([x])           # WRONG!
2 | t = t.append(x)         # WRONG!
3 | t + [x]                 # WRONG!
4 | t = t + x               # WRONG!
```

Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

```
1 | copied = t.copy()
2 | copied.sort()
```

Alternatively, you can do a copy using slicing:

```
1 | copied = t[:]
2 | copied.sort()
```

In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

10.14 Glossary

- **90 – list:**. A sequence of values.
- **91 – element:**. One of the values in a list (or other sequence), also called items.
- **92 – index:**. An integer value that indicates an element in a list.
- **93 – nested list:**. A list that is an element of another list.
- **94 – list traversal:**. The sequential accessing of each element in a list.
- **95 – mapping:**. A relationship in which each element of one set corresponds to an element of another set. For example, a list is a mapping from indices to elements.
- **96 – accumulator:**. A variable used in a loop to add up or accumulate a result.
- **97 – augmented assignment:**. A statement that updates the value of a variable using an operator like `+=`.
- **98 – reduce:**. A processing pattern that traverses a sequence and accumulates the elements into a single result.
- **99 – map:**. A processing pattern that traverses a sequence and performs an operation on each element.
- **100 – filter:**. A processing pattern that traverses a list and selects the elements that satisfy some criterion.
- **101 – object:**. Something a variable can refer to. An object has a type and a value.
- **102 – equivalent:**. Having the same value.
- **103 – identical:**. Being the same object (which implies equivalence).
- **104 – reference:**. The association between a variable and its value.
- **105 – aliasing:**. A circumstance where two or more variables refer to the same object.
- **106 – delimiter:**. A character or string used to indicate where a string should be split.

10.15 Exercises

Exercise 10.3 Write a function called `is_sorted` that takes a list as a parameter and returns `True` if the list is sorted in ascending order and `False` otherwise. You can assume (as a precondition) that the elements of the list can be compared with the relational operators `<`, `>`, etc.

For example, `is_sorted([1, 2, 2])` should return `True` and `is_sorted(['b', 'a'])` should return `False`. ■

Exercise 10.4 Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called `is_anagram` that takes two strings and returns `True` if they are anagrams. ■

Exercise 10.5 The (so-called) Birthday Paradox:

1. Write a function called `has_duplicates` that takes a list and returns `True` if there is any element that appears more than once. It should not modify the original list.
2. If there are 23 students in your class, what are the chances that two of you have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the `randint` function in the `random` module.

You can read about this problem at wikipedia.org/wiki/Birthday_paradox, and you can see my solution at thinkpython.com/code/birthday.py. ■

Exercise 10.6 Write a function called `remove_duplicates` that takes a list and returns a new list with only the unique elements from the original. Hint: they don't have to be in the same order. ■

Exercise 10.7 Write a function that reads the file `words.txt` and builds a list with one element per word. Write two versions of this function, one using the `append` method and the other using the idiom `t = t + [x]`. Which one takes longer to run? Why?

You can see my solution at thinkpython.com/code/wordlist.py. ■

Exercise 10.8 Two words are a “reverse pair” if each is the reverse of the other. Write a program that finds all the reverse pairs in the word list. ■

Exercise 10.9 To check whether a word is in the word list, you could use the `in` operator, but it would be slow because it searches through the words in order.

Because the words are in alphabetical order, we can speed things up with a bisection search (also known as binary search), which is similar to what you do when you look a word up in the dictionary. You start in the middle and check to see whether the word you are looking for comes before the word in the middle of the list. If so, then you search the first half of the list the same way. Otherwise you search the second half.

Either way, you cut the remaining search space in half. If the word list has 113,809 words, it will take about 17 steps to find the word or conclude that it's not there.

Write a function called `bisect` that takes a sorted list and a target value and returns the index of the value in the list, if it's there, or `None` if it's not.

Or you could read the documentation of the `bisect` module and use that! ■

Exercise 10.10 Two words “interlock” if taking alternating letters from each forms a new word^a. For example, “shoe” and “cold” interlock to form “schooled.”

1. Write a program that finds all pairs of words that interlock. Hint: don't enumerate all pairs!
2. Can you find any words that are three-way interlocked; that is, every third letter forms a word, starting from the first, second or third?

^aThis exercise is inspired by an example at puzzlers.org. ■

11. Dictionaries

A **dictionary** is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called **keys**) and a set of values. Each key maps to a value. The association of a key and a value is called a **key-value pair** or sometimes an **item**.

As an example, we'll build a dictionary that maps from English to French words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2fr = dict()
>>> print(eng2fr)
{}
```

The curly-brackets, `{}`, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2fr['one'] = 'un'
```

This line creates an item that maps from the key `'one'` to the value `'un'`.

If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print(eng2fr)
{'one': 'un'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2fr = {'one': 'un', 'two': 'deux', 'three': 'trois'}
```

But if you print `eng2fr`, you might be surprised:

```
>>> print(eng2fr)
{'one': 'un', 'three': 'trois', 'two': 'deux'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable. But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print(eng2fr['two'])
'deux'
```

The key `'two'` always maps to the value `'deux'` so the order of the items doesn't matter. If the key isn't in the dictionary, you get an exception:

```
>>> print(eng2fr['four'])
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    print(eng2fr['four'])
KeyError: 'four'
```

The `len` function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2fr)
3
```

The `in` operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2fr
True
>>> 'un' in eng2fr
False
```

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns the values as a list, and then use the `in` operator:

```
>>> vals = eng2fr.values()
>>> 'un' in vals
True
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it uses a search algorithm, as in [Section 9.6](#). As the list gets longer, the search time gets longer in direct proportion. For dictionaries, Python uses an algorithm called a **hashtable** that has a remarkable property: the `in` operator takes about the same amount of time no matter how many items there are in a dictionary. I won't explain how that's possible, but you can read more about it at wikipedia.org/wiki/Hash_table.

Exercise 11.1 Write a function that reads the words in `words.txt` and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the `in` operator as a fast way to check whether a string is in the dictionary.

If you did [Exercise ??](#), you can compare the speed of this implementation with the list `in` operator and the bisection search.

11.1 Dictionary as a set of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function `ord`), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way. An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
1  def histogram(text):
2      hist = dict()
3      for character in text:
4          if character not in hist:
5              hist[character] = 1
6          else:
7              hist[character] += 1
8      return hist
```

The name of the function is **histogram**, which is a statistical term for a set of counters (or frequencies).

The first line of the function creates an empty dictionary named `hist`. The `for` loop traverses the string `text`. Each time through the loop, if the character `character` is not in the dictionary, we create a new item with key `character` and the initial value 1 (since we have seen this letter once). If `character` is already in the dictionary we increment `hist[character]`. Here's how it works:

```
>>> h = histogram('brontosaurus')
>>> print(h)
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on.

Exercise 11.2 Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> h = histogram('brontosaurus')
>>> print(h)
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
>>> h.get('a', 0)
1
>>> h.get('z', 0)
0
```

Use `get` to write `histogram` more concisely. You should be able to eliminate the `if` statement. ■

11.2 Looping and dictionaries

If you use a dictionary in a `for` statement, it traverses the keys of the dictionary. For example, `print_histogram` prints each key and the corresponding value:

```
1 | def print_histogram(hist):
2 |     for character in hist:
3 |         print(character, hist[character])
```

Here's what the output looks like:

```
>>> h = histogram('parrot')
>>> print_histogram(h)
a 1
p 1
r 2
t 1
o 1
```

Again, the keys are in no particular order.

Exercise 11.3 Dictionaries have a method called `keys` that returns the keys of the dictionary, in no particular order, as a list. Modify `print_histogram` to print the keys and their values in alphabetical order. ■

11.3 Reverse lookup

Given a dictionary `d` and a key `k`, it is easy to find the corresponding value `v = d[k]`. This operation is called a **lookup**.

But what if you have `v` and you want to find `k`? You have two problems: first, there might be more than one key that maps to the value `v`. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search. Here is a function that takes a value and returns the first key that maps to that value:

```
1 | def reverse_lookup(dico, value):
2 |     for key in dico:
3 |         if dico[key] == value:
4 |             return key
5 |     raise ValueError('No such value in dictionary!')
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, `raise`. The `raise` statement causes an exception; in this case it causes a `ValueError`, which generally indicates that there is something wrong with the value of a parameter. If we get to the end of the loop, that means `value` doesn't appear in the dictionary as a value, so we raise an exception.

Here is an example of a successful reverse lookup:

```
>>> h = histogram('brontosaurus')
>>> k = reverse_lookup(h, 1)
>>> print(k)
b
```

And an unsuccessful one:

```
>>> h = histogram('brontosaurus')
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    k = reverse_lookup(h, 3)
  File "D:/Python/dictionaries.py", line 20, in reverse_lookup
    raise ValueError('No such value in dictionary!')
ValueError: No such value in dictionary!
```

The result when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

Exercise 11.4 Modify `reverse_lookup` so that it builds and returns a list of *all* keys that map to a value *v*, or an empty list if there are none. ■

11.4 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you were given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters.

Here is a function that inverts a dictionary:

```

1  def invert_dict(dico):
2      inverted = dict()
3      for key in dico:
4          value = dico[key]
5          if value not in inverted:
6              inverted[value] = [key]
7          else:
8              inverted[value].append(key)
9      return inverted

```

Each time through the loop, `key` gets a key from `dico` and `value` gets the corresponding value. If `value` is not in `inverted`, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```

>>> h = histogram('parrot')
>>> print(h)
{'p': 1, 'a': 1, 'r': 2, 'o': 1, 't': 1}
>>> inv = invert_dict(h)
>>> print(inv)
{1: ['p', 'a', 'o', 't'], 2: ['r']}

```

Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```

>>> lst = [1,2,3]
>>> dico = dict()
>>> dico[lst] = 'oops'
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    dico[lst] = 'oops'
TypeError: unhashable type: 'list'

```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be **hashable**. A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs. This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly. That's why the keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples, which we will see in Chapter 12. Since dictionaries are mutable, they can't be used as keys, but they *can* be used as values.

Exercise 11.5 Read the documentation of the dictionary method `setdefault` and use it to write a more concise version of `invert_dict`.

11.5 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

Scale down the input: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first `n` lines. If there is an error, you can reduce `n` to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers. A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a “sanity check” because it detects results that are “insane.” Another kind of check compares the results of two different computations to see if they are consistent. This is called a “consistency check.”

Pretty print the output: Formatting debugging output can make it easier to spot an error. The `pprint` module provides a `pprint` function that displays built-in types in a more human-readable format.

Again, time you spend building scaffolding can reduce the time you spend debugging.

11.6 Glossary

- **107 — dictionary:**. A mapping from a set of keys to their corresponding values.
- **108 — key-value pair:**. The representation of the mapping from a key to a value.
- **109 — item:**. Another name for a key-value pair.
- **110 — key:**. An object that appears in a dictionary as the first part of a key-value pair.
- **111 — value:**. An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word “value.”
- **112 — implementation:**. A way of performing a computation.
- **113 — hashtable:**. The algorithm used to implement Python dictionaries.
- **114 — hash function:**. A function used by a hashtable to compute the location for a key.
- **115 — hashable:**. A type that has a hash function. Immutable types like integers, floats and strings are hashable; mutable types like lists and dictionaries are not.
- **116 — lookup:**. A dictionary operation that takes a key and finds the corresponding value.
- **117 — reverse lookup:**. A dictionary operation that takes a value and finds one or more keys that map to it.
- **118 — singleton:**. A list (or other sequence) with a single element.
- **119 — call graph:**. A diagram that shows every frame created during the execution of a program, with an arrow from each caller to each callee.
- **120 — histogram:**. A histogram is an accurate representation of the distribution of numerical data. In simpler terms, it could be seen as a set of counters in a program.
- **121 — memo:**. A computed value stored to avoid unnecessary future computation.
- **122 — global variable:**. A variable defined outside a function. Global variables can be accessed from any function.
- **123 — flag:**. A boolean variable used to indicate whether a condition is true.
- **124 — declaration:**. A statement like `global` that tells the interpreter something about a variable.

11.7 Exercises

Exercise 11.6 If you did Exercise 10.5, you already have a function named `has_duplicates` that takes a list as a parameter and returns `True` if there is any object that appears more than once in the list.

Use a dictionary to write a faster, simpler version of `has_duplicates`. ■

Exercise 11.7 Two words are “rotate pairs” if you can rotate one of them and get the other (see `rotate_word` in Exercise 9.10).

Write a program that reads a wordlist and finds all the rotate pairs. ■

Exercise 11.8 Here's another Puzzler from *Car Talk*^a:

This was sent in by a fellow named Dan O'Leary. He came upon a common one-syllable, five-letter word recently that has the following unique property. When you remove the first letter, the remaining letters form a homophone of the original word, that is a word that sounds exactly the same. Replace the first letter, that is, put it back and remove the second letter and the result is yet another homophone of the original word. And the question is, what's the word?

Now I'm going to give you an example that doesn't work. Let's look at the five-letter word, 'wrack'. W-R-A-C-K, you know like to 'wrack with pain'. If I remove the first letter, I am left with a four-letter word, R-A-C-K. It's a perfect homophone. If you put the 'w' back, and remove the 'r', instead, you're left with the word, 'wack', which is a real word, it's just not a homophone of the other two words.

But there is, however, at least one word that Dan and we know of, which will yield two homophones if you remove either of the first two letters to make two, new four-letter words. The question is, what's the word?

You can use the dictionary from Exercise 11.1 to check whether a string is in the word list.

To check whether two words are homophones, you can use the CMU Pronouncing Dictionary. You can download it from www.speech.cs.cmu.edu/cgi-bin/cmudict.

Write a program that lists all the words that solve the Puzzler.

^a<https://www.cartalk.com/puzzler/browse>.

12. Tuples

12.1 Tuples are immutable

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable. Syntactically, a tuple is a comma-separated list of values:

```
1 | >>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
1 | >>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include a final comma:

```
1 | >>> t1 = ('a',)
2 | >>> type(t1)
3 | <type 'tuple'>
```

A value in parentheses is not a tuple:

```
1 | >>> t2 = ('a')
2 | >>> type(t2)
3 | <type 'str'>
```

Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

```
1 | >>> t = tuple()
2 | >>> print(t)
3 | ()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
1 | >>> t = tuple('lupins')
2 | >>> print(t)
3 | ('l', 'u', 'p', 'i', 'n', 's')
```

Because `tuple` is the name of a built-in function, you should avoid using it as a variable name. Most list operators also work on tuples. The bracket operator indexes an element:

```
1 | >>> t = ('a', 'b', 'c', 'd', 'e')
2 | >>> print(t[0])
3 | 'a'
```

And the slice operator selects a range of elements.

```
4 | >>> print(t[1:3])
5 | ('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
6 | >>> t[0] = 'A'
7 | TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
8 | >>> t = ('A',) + t[1:]
9 | >>> print(t)
10 | ('A', 'b', 'c', 'd', 'e')
```

12.2 Tuple assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap `a` and `b`:

```
1 | >>> temp = a
2 | >>> a = b
3 | >>> b = temp
```

This solution is cumbersome; **tuple assignment** is more elegant:

```
1 | >>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. The number of variables on the left and the number of values on the right have to be the same:

```
1 | >>> a, b = 1, 2, 3
2 | ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```
1 | >>> addr = 'monty@python.org'
2 | >>> uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
3 | >>> print(uname)
4 | monty
5 | >>> print(domain)
6 | python.org
```

12.3 Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute `x/y` and then `x%y`. It is better to compute them both at the same time.

The built-in function `divmod` takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

```
1 | >>> t = divmod(7, 3)
2 | >>> print(t)
3 | (2, 1)
```

Or use tuple assignment to store the elements separately:

```
1 | >>> quot, rem = divmod(7, 3)
2 | >>> print(quot)
3 | 2
4 | >>> print(rem)
5 | 1
```

Here is an example of a function that returns a tuple:

```
1 | def min_max(t):
2 |     return min(t), max(t)
```

`max` and `min` are built-in functions that find the largest and smallest elements of a sequence. `min_max` computes both and returns a tuple of two values.

12.4 Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with `*` **gathers** arguments into a tuple. For example, `printall` takes any number of arguments and prints them:

```
1 | def printall(*args):
2 |     print(args)
```

The gather parameter can have any name you like, but `args` is conventional. Here's how the function works:

```
1 | >>> printall(1, 2.0, '3')
2 | (1, 2.0, '3')
```

The complement of **gather** is **scatter**. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the `*` operator. For example, `divmod` takes exactly two arguments; it doesn't work with a tuple:

```
1 | >>> t = (7, 3)
2 | >>> divmod(t)
3 | TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
1 | >>> divmod(*t)
2 | (2, 1)
```

Exercise 12.1 Many of the built-in functions use variable-length argument tuples. For example, `max` and `min` can take any number of arguments:

```
>>> max(1, 2, 3)
3
```

But `sum` does not.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

Write a function called `sumall` that takes any number of arguments and returns their sum.

12.5 Lists and tuples

`zip` is a built-in function that takes two or more sequences and “zips” them into an iterator of tuples (for most purposes, an iterator behaves like a list) where each tuple contains one element from each sequence.

This example zips a string and a list, and build a list using the list constructor `list()`:

```
1 | >>> list(zip('abc', [1, 2, 3]))
2 | [('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples where each tuple contains a character from the string and the corresponding element from the list.

If the sequences are not the same length, the result has the length of the shorter one.

```
1 | >>> list(zip('abc', (1,2,3,4)))
2 | [('a', 1), ('b', 2), ('c', 3)]
```

You can use tuple assignment in a `for` loop to traverse a list of tuples:

```
1 | t = [('a', 0), ('b', 1), ('c', 2)]
2 | for letter, number in t:
3 |     print(number, letter)
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to `letter` and `number`. The output of this loop is:

```
0 a
1 b
2 c
```

If you combine `zip`, `for` and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, `has_match` takes two sequences, `t1` and `t2`, and returns `True` if there is an index `i` such that `t1[i] == t2[i]`:

```
1 | def has_match(t1, t2):
2 |     for x, y in zip(t1, t2):
3 |         if x == y:
4 |             return True
5 |     return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function `enumerate`:

```
1 | for index, element in enumerate('abc'):
2 |     print(index, element)
```

The output of this loop is:

```
0 a
1 b
2 c
```

Again.

12.6 Dictionaries and tuples

Dictionaries have a method called `items` that returns a `dict_items` object (kind of a list of tuples), where each tuple is a key-value pair. But a proper list can be created from the object using the `list` constructor.

```
1 >>> d = {'a':0, 'b':1, 'c':2}
2 >>> t = list(d.items())
3 >>> print(t)
4 [('a', 0), ('c', 2), ('b', 1)]
```

As you should expect from a dictionary, the items are in no particular order.

Conversely, you can use a list of tuples to initialize a new dictionary:

```
1 >>> t = [('a', 0), ('c', 2), ('b', 1)]
2 >>> d = dict(t)
3 >>> print(d)
4 {'a': 0, 'c': 2, 'b': 1}
```

Combining `dict` with `zip` yields a concise way to create a dictionary:

```
1 >>> d = dict(zip('abc', range(3)))
2 >>> print(d)
3 {'a': 0, 'c': 2, 'b': 1}
```

The dictionary method `update` also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary.

Combining `items`, tuple assignment and `for`, you get the idiom for traversing the keys and values of a dictionary:

```
1 for key, val in d.items():
2     print(val, key)
```

The output of this loop is:

```
0 a
2 c
1 b
```

Again.

It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined `last`, `first` and `number`, we could write:

```
1 | directory[(last,first)] = number
```

The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

```
1 | for last, first in directory:
2 |     print(first, last, directory[(last,first)])
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple `('Cleese', 'John')` would appear:

tuple

0	→	'Cleese'
1	→	'John'

But in a larger diagram you might want to leave out the details. For example, a diagram of the telephone directory might appear:

dict

<code>('Cleese', 'John')</code>	→	<code>'08700 100 222'</code>
<code>('Chapman', 'Graham')</code>	→	<code>'08700 100 222'</code>
<code>('Idle', 'Eric')</code>	→	<code>'08700 100 222'</code>
<code>('Gilliam', 'Terry')</code>	→	<code>'08700 100 222'</code>
<code>('Jones', 'Terry')</code>	→	<code>'08700 100 222'</code>
<code>('Palin', 'Michael')</code>	→	<code>'08700 100 222'</code>

Here the tuples are shown using Python syntax as a graphical shorthand. The telephone number in the diagram is the complaints line for the BBC, so please don't call it.

12.7 Comparing tuples

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
1 >>> (0, 1, 2) < (0, 3, 4)
2 True
3 >>> (0, 1, 2000000) < (0, 3, 4)
4 True
```

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on. This feature lends itself to a pattern called **DSU** for

Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

Sort the list of tuples, and

Undecorate by extracting the sorted elements of the sequence.

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
1 def sort_by_length(words):
2     t = []
3     for word in words:
4         t.append((len(word), word))
5
6     t.sort(reverse=True)
7
8     res = []
9     for length, word in t:
10        res.append(word)
11    return res
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length. `sort` compares the first element, length, first, and only considers the second element to break ties. The keyword argument `reverse=True` tells `sort` to go in decreasing order. The second loop traverses the list of tuples and builds a list of words in descending order of length.

Exercise 12.2 In this example, words with the same length appear in reverse alphabetical order. Modify this example so that words with the same length appear in random order. Hint: see the `random` function in the `random` module.

12.8 Sequences of sequences

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences. In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how and why do you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. But Python provides the built-in functions `sorted` and `reversed`, which take any sequence as a parameter and return a new list with the same elements in a different order.

12.9 Glossary

- **125 — tuple:** An immutable sequence of elements.
- **126 — tuple assignment:** An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.
- **127 — gather:** The operation of assembling a variable-length argument tuple.
- **128 — scatter:** The operation of treating a sequence as a list of arguments.
- **129 — DSU:** Abbreviation of “decorate-sort-undecorate,” a pattern that involves building a list of tuples, sorting, and extracting part of the result.
- **130 — data structure:** A collection of related values, often organized in lists, dictionaries, tuples, etc.
- **131 — shape (of a data structure):** A summary of the type, size and composition of a data structure.

12.10 Exercises

Exercise 12.3 Write a function called `most_frequent` that takes a string and prints the letters in decreasing order of frequency. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at wikipedia.org/wiki/Letter_frequencies.

Exercise 12.4 More anagrams!

1. Write a program that reads a word list from a file (see Exercise ??) and prints all the sets of words that are anagrams.

Here is an example of what the output might look like:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']  
['retainers', 'ternaries']  
['generating', 'greatening']  
['resmelts', 'smelters', 'termless']
```

Hint: you might want to build a dictionary that maps from a set of letters to a list of words that can be spelled with those letters. The question is, how can you represent the set of letters in a way that can be used as a key?

2. Modify the previous program so that it prints the largest set of anagrams first, followed by the second largest set, and so on.
3. In Scrabble a “bingo” is when you play all seven tiles in your rack, along with a letter on the board, to form an eight-letter word. What set of 8 letters forms the most possible bingos? Hint: there are seven.
4. Two words form a “metathesis pair” if you can transform one into the other by swapping two letters^a; for example, “converse” and “conserve.” Write a program that finds all of the metathesis pairs in the dictionary. Hint: don’t test all pairs of words, and don’t test all possible swaps.

^aThis exercise is inspired by an example at puzzlers.org.

Exercise 12.5 Here's another Car Talk Puzzler^a:

What is the longest English word, that remains a valid English word, as you remove its letters one at a time?

Now, letters can be removed from either end, or the middle, but you can't rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you're eventually going to wind up with one letter and that too is going to be an English word—one that's found in the dictionary. I want to know what's the longest word and how many letters does it have?

I'm going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we're left with the word spite, then we take the e off the end, we're left with spit, we take the s off, we're left with pit, it, and I.

Write a program to find all words that can be reduced in this way, and then find the longest one.

This exercise is a little more challenging than most, so here are some suggestions:

1. You might want to write a function that takes a word and computes a list of all the words that can be formed by removing one letter. These are the "children" of the word.
2. Recursively, a word is reducible if any of its children are reducible. As a base case, you can consider the empty string reducible.
3. The wordlist I provided, `words.txt`, doesn't contain single letter words. So you might want to add "I", "a", and the empty string.
4. To improve the performance of your program, you might want to memorize the words that are known to be reducible.

^a <https://www.cartalk.com/puzzler/browse>.

13. Files

13.1 Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off. Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network. One of the simplest ways for programs to maintain their data is by reading and writing text files. A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM.

13.2 Reading from a file

In this chapter we will be using a list of English words. There are lots of word lists available on the Web, but the one most suitable for our purpose is one of the word lists collected and contributed to the public domain by Grady Ward as part of the Moby lexicon project¹. It is a list of 113,809 official crosswords; that is, words that are considered valid in crossword puzzles and other word games. In the Moby collection, the filename is `113809of.fic`; I include a copy of this file, with the simpler name `words.txt`, along with Swampy.

¹wikipedia.org/wiki/Moby_Project.

This file is in plain text, so you can open it with a text editor, but you can also read it from Python. The built-in function `open` takes the name of the file as a parameter and returns a **file object** you can use to read the file.

```
>>> fin = open('words.txt')
>>> print(fin)
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

`fin` is a common name for a file object used for input. Mode `'r'` indicates that this file is open for reading (as opposed to `'w'` for writing).

The file object provides several methods for reading, including `readline`, which reads characters from the file until it gets to a newline and returns the result as a string:

```
>>> fin.readline()
'aa\r\n'
```

The first word in this particular list is “aa,” which is a kind of lava. The sequence `\r\n` represents two whitespace characters, a carriage return and a newline, that separate this word from the next.

The file object keeps track of where it is in the file, so if you call `readline` again, you get the next word:

```
>>> fin.readline()
'aah\r\n'
```

The next word is “aah,” which is, believe it or not, a perfectly legitimate word. If the whitespace is bothering you, we can get rid of it with the string method `strip`:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> print(word)
aahed
```

You can also use a file object as part of a `for` loop. This program reads `words.txt` and prints each word, one per line:

```
1 fin = open('words.txt')
2 for line in fin:
3     word = line.strip()
4     print(word)
```


13.3 Writing to a file

To write a file, you have to open it with mode `'w'` as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

Again, the file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
```

When you are done writing, you have to close the file.

```
>>> fout.close()
```

13.4 Format operator

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
>>> f.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator. The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string. For example, the format sequence `'%d'` means that the second operand should be formatted as an integer (d stands for “decimal”):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string `'42'`, which is not to be confused with the integer value 42. A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order. The following example uses `'%d'` to format an integer, `'%g'` to format a floating-point number (don't ask why), and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at docs.python.org/2.5/lib/typesseq-strings.html.

13.5 Filenames and paths

Files are organized into **directories** (also called “folders”). Every running program has a “current directory,” which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories (“os” stands for “operating system”). `os.getcwd` returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> print(cwd)
/home/dinsdale
```

`cwd` stands for “current working directory.” The result in this example is `/home/dinsdale`, which is the home directory of a user named `dinsdale`. A string like `cwd` that identifies a file is called a **path**. A **relative path** starts from the current directory; an **absolute**

`path` starts from the topmost directory in the file system. The paths we have seen so far are simple filenames, so they are relative to the current directory. To find the absolute path to a file, you can use `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path.exists` checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, `os.path.isdir` checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Similarly, `os.path.isfile` checks whether it's a file.

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
1 def walk(dir):
2     for name in os.listdir(dir):
3         path = os.path.join(dir, name)
4
5         if os.path.isfile(path):
6             print(path)
7         else:
8             walk(path)
```

`os.path.join` takes a directory and a file name and joins them into a complete path.

Exercise 13.1 Modify `walk` so that instead of printing the names of the files, it returns a list of names. ■

Exercise 13.2 The `os` module provides a function called `walk` that is similar to this one but more versatile. Read the documentation and use it to print the names of the files in a given directory and its subdirectories. ■

13.6 Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `IOError`:

```
>>> fin = open('bad_file')
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
IOError: [Errno 21] Is a directory
```

To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities.

It is better to go ahead and try, and deal with problems if they happen, which is exactly what the `try` statement does. The syntax is similar to an `if` statement:

```
1  try:
2      fin = open('bad_file')
3      for line in fin:
4          print(line)
5      fin.close()
6  except:
7      print('Something went wrong.')
```

Python starts by executing the `try` clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and executes the `except` clause.

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

13.6.1 finally clause

There are two options to the `try-except` statement. The first one is the clause `finally`. All code included in the `finally` clause will be executed, whether an exception occurred or not. This is a good place to clean up our program. For example this is a good place to close a file if it has been open in the `try` clause. The code below shows how it is done.

```
1  fin = None
2  try:
3      print('Try to open a file.')
4      fin = open('words.txt')
5      for line in fin:
6          print(line)
7  except:
8      print('Something went wrong.')
9  finally:
10     print('cleaning up')
11     if fin is not None:
12         print('closing the file')
13         fin.close()
```

First we set the variable `fin` to `None`. In the `try` clause, we open a file and then assign it to `fin`. Two things can happen:

- An exception occurs while we are trying to open the file, `fin` is not assigned any new object and contains the value `None`. The program jumps directly to the `except` clause and executes the code in the `except` block. Then the program jumps to the `finally` clause and executes the code there.
- The file is opened successfully, `fin` is assigned the file object, and the rest of the code in the `try` statement is executed. Then the program jumps to the `finally` clause and executes the code there.

13.6.2 else clause

The second option is the `else` clause, which should be after the `except` clause and before the `finally` clause. The code in the `else` clause is executed only if no exceptions were raised. It is executed before the `finally` clause. The `else` clause is used for all code that does not raise any exception.

In our previous code, it would be the place to read the lines in the file. The refactored code is:

```
1  fin = None
2  try:
3      fin = open('word.txt')
4  except:
5      print('Something went wrong.')
6  else:
7      print('Do your thing if all went well.')
8      for line in fin:
9          print(line)
10 finally:
11     print('cleaning up.')
12     if fin is not None:
13         print('closing the file.')
14         fin.close()
```

As you can see, the `try` clause contains only the code that may raise an exception.

- If an exception occurs whilst opening the file, the program jumps to the `except` clause and then executes the `finally` clause. In this case, the output is:

```
TRY: open file.
EXCEPT: Something went wrong.
FINALLY: cleaning up.
```

- If no exceptions are raised, the program skips the `except` clause, and jumps to the `else` clause. Once the code in the `else` clause has been executed, the program jumps to the `finally` clause. In this case, the output is:

```
TRY: open file.
--> file open successfully.
ELSE: Do your thing if all went well.
--> line 1 of text file

--> line 2 of text file

--> last line of text file

FINALLY: cleaning up.
--> closing the file.
```

13.7 Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

```
1 def linecount(filename):
2     count = 0
3     for line in open(filename):
4         count += 1
5     return count
6
7 print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

Now you have a module object `wc`:

```
>>> print(wc)
<module 'wc' from 'wc.py'>
```

That provides a function called `linecount`:

```
>>> wc.linecount('wc.py')
7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it executes the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't execute them.

Programs that will be imported as modules often use the following idiom:

```
1 if __name__ == '__main__':
2     print(linecount('wc.py'))
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `__main__`; in that case, the test code is executed. Otherwise, if the module is being imported, the test code is skipped.

Exercise 13.3 Type this example into a file named `wc.py` and run it as a script. Then run the Python interpreter and `import wc`. What is the value of `__name__` when the module is being imported?

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function `reload`, but it can

be tricky, so the safest thing to do is restart the interpreter and then import the module again. ■

13.8 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2      3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies might cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at wikipedia.org/wiki/Newline. Or, of course, you could write one yourself.

13.9 Glossary

- **132 — persistent:** Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.
- **133 — format operator:** An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.
- **134 — format string:** A string, used with the format operator, that contains format sequences.
- **135 — format sequence:** A sequence of characters in a format string, like %d, that specifies how a value should be formatted.
- **136 — text file:** A sequence of characters stored in permanent storage like a hard drive.
- **137 — directory:** A named collection of files, also called a folder.
- **138 — path:** A string that identifies a file.
- **139 — relative path:** A path that starts from the current directory.
- **140 — absolute path:** A path that starts from the topmost directory in the file system.
- **141 — catch:** To prevent an exception from terminating a program using the try and except statements.

13.10 Exercises

Exercise 13.4 The `urllib` module provides methods for manipulating URLs and downloading information from the web. The following example downloads and prints a secret message from `thinkpython.com`:

```
import urllib

conn = urllib.urlopen('http://thinkpython.com/secret.html')
for line in conn.fp:
    print(line.strip())
```

Run this code and follow the instructions you see there.

Exercise 13.5 Write a program that reads `words.txt` and prints only the words with more than 20 characters (not counting whitespace).

Exercise 13.6 Write a function named `avoids` that takes a word and a string of forbidden letters, and that returns `True` if the word doesn't use any of the forbidden letters.

Modify your program to prompt the user to enter a string of forbidden letters and then print the number of words that don't contain any of them. Can you find a combination

of 5 forbidden letters that excludes the smallest number of words? ■

Exercise 13.7 In 1939 Ernest Vincent Wright published a 50,000 word novel called *Gadsby* that does not contain the letter “e.” Since “e” is the most common letter in English, that’s not easy to do.

In fact, it is difficult to construct a solitary thought without using that most common symbol. It is slow going at first, but with caution and hours of training you can gradually gain facility.

All right, I’ll stop now.

Write a function called `has_no_e` that returns `True` if the given word doesn’t have the letter “e” in it.

Modify your program from the previous section to print only the words that have no “e” and compute the percentage of the words in the list have no “e.”

■

Exercise 13.8 Write a function named `uses_only` that takes a word and a string of letters, and that returns `True` if the word contains only letters in the list. Can you make a sentence using only the letters `acefhlo`? Other than “Hoe alfalfa?”

■

Exercise 13.9 Write a function named `uses_all` that takes a word and a string of required letters, and that returns `True` if the word uses all the required letters at least once. How many words are there that use all the vowels `aeiou`? How about `aeiouy`?

■

Exercise 13.10 Write a function called `is_abecedarian` that returns `True` if the letters in a word appear in alphabetical order (double letters are ok). How many abecedarian words are there?

■

Exercise 13.11 A palindrome is a word that reads the same forward and backward, like “rotator” and “noon.” Write a boolean function named `is_palindrome` that takes a string as a parameter and returns `True` if it is a palindrome.

Modify your program from the previous section to print all of the palindromes in the word list and then print the total number of palindromes.

■



Part III - User Defined Data Structures

14	Classes and objects	149
15	Classes and functions	157
16	Classes and methods	164

14. Classes and objects

14.1 User-defined types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called `Point` that represents a point in two-dimensional space. In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, $(0,0)$ represents the origin, and (x,y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, `x` and `y`.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is (a little) more complicated than the other options, but it has advantages that will be apparent soon. A user-defined type is also called a **class**. A class definition looks like this:

```
1 | class Point(object):  
2 |     """represents a point in 2-D space"""
```

This header indicates that the new class is a `Point`, which is a kind of `object`, which is a built-in type. The body is a docstring that explains what the class is for. You can define variables and functions inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a class object.

```
>>> print(Point)
<class '__main__.Point'>
```

Because `Point` is defined at the top level, its “full name” is `__main__.Point`. The class object is like a factory for creating objects. To create a `Point`, you call `Point` as if it were a function.

```
>>> blank = Point()
>>> print(blank)
<__main__.Point instance at 0xb7e9d3ac>
```

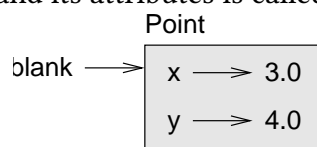
The return value is a reference to a `Point` object, which we assign to `blank`. Creating a new object is called **instantiation**, and the object is an **instance** of the class. When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix `0x` means that the following number is in hexadecimal).

14.2 Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.whitespace`. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**. As a noun, “AT-trib-ute” is pronounced with emphasis on the first syllable, as opposed to “a-TRIB-ute,” which is a verb. The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**:



The variable `blank` refers to a `Point` object, which contains two attributes. Each attribute refers to a floating-point number. You can read the value of an attribute using the same syntax:

```
>>> print(blank.y)
4.0
>>> x = blank.x
>>> print(x)
3.0
```

The expression `blank.x` means, “Go to the object `blank` refers to and get the value of `x`.” In this case, we assign that value to a variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

You can use dot notation as part of any expression. For example:

```
>>> print('( %g, %g)' % (blank.x, blank.y))
(3.0, 4.0)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> print(distance)
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
1 def print_point(p):
2     print('( %g, %g)' % (p.x, p.y))
```

`print_point` takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass `blank` as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, `p` is an alias for `blank`, so if the function modifies `p`, `blank` changes.

Exercise 14.1 Write a function called `distance` that takes two `Points` as arguments and returns the distance between them. The distance between two points $A = (x_A, y_A)$ and $B = (x_B, y_B)$ is given by:

$$d(A, B) = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2} \quad (14.1)$$

14.3 Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.

Here is the class definition:

```

1  class Rectangle(object):
2      """represent a rectangle.
3      attributes: width, height, corner (bottom left corner).
4      """

```

The docstring lists the attributes: `width` and `height` are numbers; `corner` is a `Point` object that specifies the lower-left corner. To represent a rectangle, you have to instantiate a `Rectangle` object and assign values to the attributes:

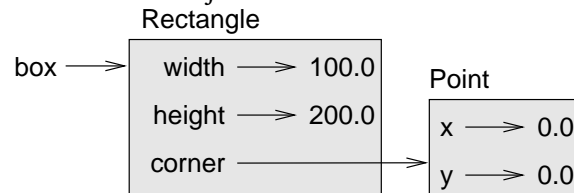
```

1  box = Rectangle()
2  box.width = 100.0
3  box.height = 200.0
4  box.corner = Point()
5  box.corner.x = 0.0
6  box.corner.y = 0.0

```

The expression `box.corner.x` means, “Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`.”

The figure shows the state of this object:



An object that is an attribute of another object is **embedded**.

14.4 Instances as return values

Functions can return instances. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```

1  def find_center(box):
2      p = Point()
3      p.x = box.corner.x + box.width/2.0
4      p.y = box.corner.y + box.height/2.0
5      return p

```

Here is an example that passes `box` as an argument and assigns the resulting `Point` to `center`:

```

>>> center = find_center(box)
>>> print_point(center)
(50.0, 100.0)

```


14.5 Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of `width` and `height`:

```
1 | box.width = box.width + 50
2 | box.height = box.width + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a `Rectangle` object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
1 | def grow_rectangle(rect, dwidth, dheight) :
2 |     rect.width += dwidth
3 |     rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> print(box.width)
100.0
>>> print(box.height)
200.0
>>> grow_rectangle(box, 50, 100)
>>> print(box.width)
150.0
>>> print(box.height)
300.0
```

Inside the function, `rect` is an alias for `box`, so if the function modifies `rect`, `box` changes.

Exercise 14.2 Write a function named `move_rectangle` that takes a `Rectangle` and two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `x` coordinate of `corner` and adding `dy` to the `y` coordinate of `corner`. ■

14.6 Copying

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The `copy` module contains a function called `copy` that can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` and `p2` contain the same data, but they are not the same `Point`.

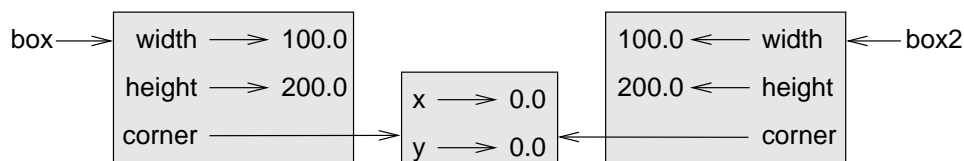
```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

The `is` operator indicates that `p1` and `p2` are not the same object, which is what we expected. But you might have expected `==` to yield `True` because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the `==` operator is the same as the `is` operator; it checks object identity, not object equivalence. Thankfully, this behavior can be changed—we'll see how later.

If you use `copy.copy` to duplicate a `Rectangle`, you will find that it copies the `Rectangle` object but not the embedded `Point`.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Here is what the object diagram looks like:



This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects. For most applications, this is not what you want. In this example, invoking `grow_rectangle` on one of the `Rectangles` would not affect the other, but invoking `move_rectangle` on either would affect both! This behavior is confusing and error-prone. Fortunately, the `copy` module contains a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

box3 and box are completely separate objects.

Exercise 14.3 Write a version of `move_rectangle` that creates and returns a new `Rectangle` instead of modifying the old one. ■

14.7 Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an `AttributeError`:

```
>>> p = Point()
>>> print(p.z)
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<<class '__main__.Point'>
```

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

14.8 Glossary

- 142 — **class**:. A user-defined type. A class definition creates a new class object.
- 143 — **class object**:. An object that contains information about a user-defined type. The class object can be used to create instances of the type.
- 144 — **instance**:. An object that belongs to a class.
- 145 — **attribute**:. One of the named values associated with an object.
- 146 — **embedded (object)**:. An object that is stored as an attribute of another object.

- **147 — shallow copy:** To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.
- **148 — deep copy:** To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.
- **149 — object diagram:** A diagram that shows objects, their attributes, and the values of the attributes.

15. Classes and functions

15.1 Time

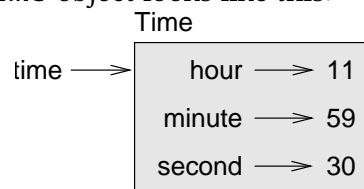
As another example of a user-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
1 class Time(object):  
2     """represents the time of day.  
3         attributes: hour, minute, second"""
```

We can create a new `Time` object and assign attributes for hours, minutes, and seconds:

```
1 time = Time()  
2 time.hour = 11  
3 time.minute = 59  
4 time.second = 30
```

The state diagram for the `Time` object looks like this:



Exercise 15.1 Write a function called `print_time` that takes a `Time` object and prints it in the form `hour:minute:second`.

Hint: the format sequence `'%.2d'` prints an integer using at least two digits, including a leading zero if necessary. ■

Exercise 15.2 Write a boolean function called `is_after` that takes two `Time` objects, `t1` and `t2`, and returns `True` if `t1` follows `t2` chronologically and `False` otherwise.

Challenge: don't use an `if` statement. ■

15.2 Pure functions

In the next few sections, we'll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

Here is a simple prototype of `add_time`:

```

1  def add_time(t1, t2):
2      sum = Time()
3      sum.hour = t1.hour + t2.hour
4      sum.minute = t1.minute + t2.minute
5      sum.second = t1.second + t2.second
6      return sum

```

The function creates a new `Time` object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To test this function, I'll create two `Time` objects: `start` contains the start time of a movie, like *Monty Python and the Holy Grail*, and `duration` contains the run time of the movie, which is one hour 35 minutes. `add_time` figures out when the movie will be done.

```

>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0
>>>
>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0
>>>
>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00

```

The result, 10 : 80 : 00 might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to “carry” the extra seconds into the minute column or the extra minutes into the hour column.

Here’s an improved version:

```
1  def add_time(t1, t2):
2      sum = Time()
3      sum.hour = t1.hour + t2.hour
4      sum.minute = t1.minute + t2.minute
5      sum.second = t1.second + t2.second
6
7      if sum.second >= 60:
8          sum.second -= 60
9          sum.minute += 1
10
11     if sum.minute >= 60:
12         sum.minute -= 60
13         sum.hour += 1
14
15     return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

15.3 Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**. `increment`, which adds a given number of seconds to a `Time` object, can be written naturally as a modifier. Here is a rough draft:

```
1  def increment(time, seconds):
2      time.second += seconds
3
4      if time.second >= 60:
5          time.second -= 60
6          time.minute += 1
7
8      if time.minute >= 60:
9          time.minute -= 60
10         time.hour += 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before. Is this function correct? What happens if the parameter `seconds` is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until `time.second` is less than sixty. One solution is to replace the `if` statements with `while` statements. That would make the function correct, but not very efficient.

Exercise 15.3 Write a correct version of `increment` that doesn't contain any loops.

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage.

Exercise 15.4 Write a “pure” version of `increment` that creates and returns a new `Time` object rather than modifying the parameter.

15.4 Prototyping versus planning

The development plan I am demonstrating is called “prototype and patch.” For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way. This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is **planned development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a `Time` object is really a three-digit number in base 60 (see wikipedia.org/wiki/Sexagesimal.)! The `second` attribute is the “ones column,” the `minute` attribute is the “sixties column,” and the `hour` attribute is the “thirty-six hundreds column.”

When we wrote `add_time` and `increment`, we were effectively doing addition in base 60, which is why we had to carry from one column to the next. This observation suggests another approach to the whole problem—we can convert `Time` objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts `Times` to integers:

```
1 | def time_to_int(time):
2 |     minutes = time.hour * 60 + time.minute
3 |     seconds = minutes * 60 + time.second
4 |     return seconds
```

And here is the function that converts integers to `Times` (recall that `divmod` divides the first argument by the second and returns the quotient and remainder as a tuple).

```
1 | def int_to_time(seconds):
2 |     time = Time()
3 |     minutes, time.second = divmod(seconds, 60)
```



```

4 |     time.hour, time.minute = divmod(minutes, 60)
5 |     return time

```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that `time_to_int(int_to_time(x)) == x` for many values of `x`. This is an example of a consistency check. Once you are convinced they are correct, you can use them to rewrite `add_time`:

```

1 | def add_time(t1, t2):
2 |     seconds = time_to_int(t1) + time_to_int(t2)
3 |     return int_to_time(seconds)

```

This version is shorter than the original, and easier to verify.

Exercise 15.5 Rewrite `increment` using `time_to_int` and `int_to_time`. ■

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better. But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (`time_to_int` and `int_to_time`), we get a program that is shorter, easier to read and debug, and more reliable. It is also easier to add features later. For example, imagine subtracting two `Times` to find the duration between them. The naïve approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

Exercise 15.6 Implement a function `time_difference(time1, time2)` which returns the time difference between `time1` and `time2`. The returned time must be positive regardless of `time1` being later or not than `time2`. ■

15.5 Debugging

A `Time` object is well-formed if the values of `minutes` and `seconds` are between 0 and 60 (including 0 but not 60) and if `hours` is positive. `hours` and `minutes` should be integral values, but we might allow `seconds` to have a fraction part.

Requirements like these are called **invariants** because they should always be true. To put it a different way, if they are not true, then something has gone wrong.

Writing code to check your invariants can help you detect errors and find their causes. For example, you might have a function like `valid_time` that takes a `Time` object and returns `False` if it violates an invariant:

```
1 def valid_time(time):
2     if time.hours < 0 or time.minutes < 0 or time.seconds < 0:
3         return False
4     if time.minutes >= 60 or time.seconds >= 60:
5         return False
6     return True
```

Then at the beginning of each function you could check the arguments to make sure they are valid:

```
1 def add_time(t1, t2):
2     if not valid_time(t1) or not valid_time(t2):
3         raise ValueError('invalid Time object in add_time')
4     seconds = time_to_int(t1) + time_to_int(t2)
5     return int_to_time(seconds)
```

Or you could use an `assert` statement, which checks a given invariant and raises an exception if it fails:

```
1 def add_time(t1, t2):
2     assert valid_time(t1) and valid_time(t2)
3     seconds = time_to_int(t1) + time_to_int(t2)
4     return int_to_time(seconds)
```

`assert` statements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

15.6 Glossary

- **150 — prototype and patch:** A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.
- **151 — planned development:** A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.
- **152 — pure function:** A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.
- **153 — modifier:** A function that changes one or more of the objects it receives as arguments. Most modifiers are fruitless.
- **154 — functional programming style:** A style of program design in which the majority of functions are pure.
- **155 — invariant:** A condition that should always be true during the execution of a program.

15.7 Exercises

Exercise 15.7 Write a function called `mul_time` that takes a `Time` object and a number and returns a new `Time` object that contains the product of the original `Time` and the number.

Then use `mul_time` to write a function that takes a `Time` object that represents the finishing time in a race, and a number that represents the distance, and returns a `Time` object that represents the average pace (time per mile).

Exercise 15.8 Write a class definition for a `Date` object that has attributes `day`, `month` and `year`. Write a function called `increment_date` that takes a `Date` object, `date` and an integer, `n`, and returns a new `Date` object that represents the day `n` days after `date`. Hint: “Thirty days hath September...” Challenge: does your function deal with leap years correctly? See wikipedia.org/wiki/Leap_year.

Exercise 15.9 The `datetime` module provides `date` and `time` objects that are similar to the `Date` and `Time` objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at docs.python.org/lib/datetime-date.html.

1. Use the `datetime` module to write a program that gets the current date and prints the day of the week.
2. Write a program that takes a birthday as input and prints the user’s age and the number of days, hours, minutes and seconds until their next birthday.

16. Classes and methods

16.1 Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming. It is not easy to define object-oriented programming, but we have already seen some of its characteristics:

- Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.
- Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

For example, the `Time` class defined in Chapter ?? corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the `Point` and `Rectangle` classes correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in the `Time` program, there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` object as an argument. This observation is the motivation for **methods**; a method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for user-defined types.

Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it simply by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

16.2 What is an instance?

Before we get into creating a class itself, we need to understand an important distinction. A class is something that just contains structure – it defines how something should be laid out or structured, but doesn't actually fill in the content. For example, the `Time` class says that a time needs to have hours, minutes and seconds, but it does not actually say what the `time` is. This is where instances come in. An instance is a specific copy of the class that does contain all of the content. For example, if I create a time `t` of 1 hour 45 minutes and 10 seconds, then `t` is an instance of `Time`.

This can sometimes be a very difficult concept to master, so let's look at it from another angle. Let's say that the government has a particular tax form that it requires everybody to fill out. Everybody has to fill out the same type of form, but the content that people put into the form differs from person to person. A class is like the form: it specifies what content should exist. Your copy of the form with your specific information is like an instance of the class: it specifies what the content actually is.

16.3 Printing objects

In Chapter ??, we defined a class named `Time` and in Exercise ??, you wrote a function named `print_time`:

```
1 class Time(object):
2     """represents the time of day.
3         attributes: hour, minute, second"""
4
5     def print_time(time):
6         print('%s.2d:%s.2d:%s.2d' %
7               (time.hour, time.minute, time.second))
```

To call this function, you have to pass a `Time` object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
```

```
>>> print_time(start)
09:45:00
```

To make `print_time` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
1 class Time(object):
2     """represents the time of day.
3         attributes: hour, minute, second"""
4
5     def print_time(time):
6         print('%s.2d:%s.2d:%s.2d' %
7               (time.hour, time.minute, time.second))
```

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

In this use of dot notation, `print_time` is the name of the method (again), and `start` is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case `start` is assigned to `time`.

16.4 The `self` variable

By convention, the first parameter of a method is called `self`, so it would be more common to write `print_time` like this:

```
1 class Time(object):
2     """represents the time of day.
3         attributes: hour, minute, second"""
4
5     def print_time(self):
6         print('%s.2d:%s.2d:%s.2d' %
7               (self.hour, self.minute, self.second))
```

Invoking the method remains the same:

```
>>> start.print_time()
09:45:00
```

The reason for this convention is an implicit metaphor:

- The syntax for a function call, `print_time(start)`, suggests that the function is the active agent. It says something like, “Hey `print_time`! Here’s an object for you to print.”
- In object-oriented programming, the objects are the active agents. A method invocation like `start.print_time()` says “Hey `start`! Please print yourself.”

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions, and makes it easier to maintain and reuse code.

R You might have noticed that the `print_time` method have this `self` variable as parameter, but that when you call the method you do not have to pass any value in the parameter. Why don’t we have to pass in the `self` parameter? This phenomena is a special behavior of Python: when you call a method of an instance, Python automatically figures out what `self` should be (from the instance) and passes it to the function. In the case of `print_time`, Python first creates `self` and then passes it in.

To make it a little bit clearer as to what is going on, we can look at two different ways of calling `print_time`.

- The first way is the standard way of doing it:
`start.print_time()`
- The second, while not conventional, is equivalent:
`Time.print_time(start)`
In this use of dot notation, `Time` is the name of the class, and `print_time` is the name of the method. `start` is passed as a parameter.

Note how in the second example we had to pass in the instance because we did not call the method via the instance. Python can’t figure out what the instance is if it doesn’t have any information about it.

Exercise 16.1 Rewrite `time_to_int` (from Section 15.4) as a method. It is probably not appropriate to rewrite `int_to_time` as a method; it’s not clear what object you would invoke it on! ■

16.5 Another example

Here’s a version of `increment` (from Section 15.3) rewritten as a method:

```

1  # inside class Time:
2
3      def increment(self, seconds):
4          seconds += self.time_to_int()
5          return int_to_time(seconds)

```

This version assumes that `time_to_int` is written as a method, as in Exercise 16.1. Also, note that it is a pure function, not a modifier.

Here's how you would invoke `increment`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

The subject, `start`, gets assigned to the first parameter, `self`. The argument, `1337`, gets assigned to the second parameter, `seconds`.

This mechanism can be confusing, especially if you make an error. For example, if you invoke `increment` with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

16.6 A more complicated example

`is_after` (from Exercise 15.2) is slightly more complicated because it takes two `Time` objects as parameters. In this case it is conventional to name the first parameter `self` and the second parameter `other`:

```
1  # inside class Time:
2
3      def is_after(self, other):
4          return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: “end is after start?”

16.7 The `init` method

The `init` method (short for “initialization”) is a special method that gets invoked when an object is instantiated. Its full name is `__init__` (two underscore characters, followed by `init`, and then two more underscores). An `init` method for the `Time` class might look like this:


```
1 | # inside class Time:
2 |
3 |     def __init__(self, hour=0, minute=0, second=0):
4 |         self.hour = hour
5 |         self.minute = minute
6 |         self.second = second
```

It is common for the parameters of `__init__` to have the same names as the attributes. The statement

```
1 |         self.hour = hour
```

stores the value of the parameter `hour` as an attribute of `self`.

The parameters are optional, so if you call `Time` with no arguments, you get the default values.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides `hour`:

```
>>> time = Time(9)
>>> time.print_time()
09:00:00
```

If you provide two arguments, they override `hour` and `minute`.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.

Exercise 16.2 Write an `init` method for the `Point` class that takes `x` and `y` as optional parameters and assigns them to the corresponding attributes. ■

16.8 The `__str__` method

`__str__` is a special method, like `__init__`, that is supposed to return a string representation of an object.

For example, here is a `str` method for `Time` objects:

```

1  # inside class Time:
2
3      def __str__(self):
4          return '%.2d:%.2d:%.2d'
5                  % (self.hour, self.minute, self.second))

```

When you print an object, Python invokes the `str` method:

```

>>> time = Time(9, 45)
>>> print(time)
09:45:00

```

When I write a new class, I almost always start by writing `__init__`, which makes it easier to instantiate objects, and `__str__`, which is useful for debugging.

Exercise 16.3 Write a `str` method for the `Point` class. Create a `Point` object and print it. ■

16.9 Operator overloading

By defining other special methods, you can specify the behavior of operators on user-defined types. For example, if you define a method named `__add__` for the `Time` class, you can use the `+` operator on `Time` objects.

Here is what the definition might look like:

```

1  # inside class Time:
2
3      def __add__(self, other):
4          seconds = self.time_to_int() + other.time_to_int()
5          return int_to_time(seconds)

```

And here is how you could use it:

```

>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00

```

When you apply the `+` operator to `Time` objects, Python invokes `__add__`. When you print the result, Python invokes `__str__`. So there is quite a lot happening behind the scenes!

Changing the behavior of an operator so that it works with user-defined types is called **operator overloading**. For every operator in Python there is a corresponding special method, like `__add__`. For more details, see docs.python.org/ref/specialnames.html.

Exercise 16.4 Write an add method for the Point class. ■

16.10 Type-based dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of `__add__` that checks the type of `other` and invokes either `add_time` or `increment`:

```

1  # inside class Time:
2
3      def __add__(self, other):
4          if isinstance(other, Time):
5              return self.add_time(other)
6          else:
7              return self.increment(other)
8
9      def add_time(self, other):
10         seconds = self.time_to_int() + other.time_to_int()
11         return int_to_time(seconds)
12
13     def increment(self, seconds):
14         seconds += self.time_to_int()
15         return int_to_time(seconds)

```

The built-in function `isinstance` takes a value and a class object, and returns `True` if the value is an instance of the class.

If `other` is a Time object, `__add__` invokes `add_time`. Otherwise it assumes that the parameter is a number and invokes `increment`. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments.

Here are examples that use the `+` operator with different types:

```

>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17

```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```

>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'

```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how to do that. But there is a clever

solution for this problem: the special method `__radd__`, which stands for “right-side add.” This method is invoked when a `Time` object appears on the right side of the `+` operator. Here’s the definition:

```
1  # inside class Time:
2
3      def __radd__(self, other):
4          return self.__add__(other)
```

And here’s how it’s used:

```
>>> print(1337 + start)
10:07:17
```

Unfortunately we are not done yet. For example, adding a Boolean `True` to a time has no sense and should raise an exception. However, our current solution allows such operation has shown here:

```
>>> print(start + True)
09:45:01
>>> print(start + 1.1)
09:45:01
```

The following code solves the issue by managing legal and illegal addition operations, ensuring that only additions with another `Time` object or an `int` are allowed.

```
1  # inside class Time:
2
3      def __add__(self, other):
4          if isinstance(other, Time):
5              return self.add_time(other)
6          elif isinstance(other, int)
7              and not isinstance(other, bool)):
8              return self.increment(other)
9          else:
10             raise TypeError("can't add Time with " +
11                             type(other).__qualname__)
```

We have now achieved the desired behaviour, where trying to do an illegal addition raises an exception.

```
>>> print(start + True)
Traceback (most recent call last):
...
TypeError: can't add Time with bool

>>> print(start + 1.5)
Traceback (most recent call last):
...
```

```
TypeError: can't add Time with float
```

```
>>> print(start + '11:05:03')
Traceback (most recent call last):
...
TypeError: can't add Time with str
```

R You may have noticed the Boolean expression `not isinstance(other, bool)` in the `elif` clause. One could wonder why we needed to this additional condition. As we mentioned earlier in Section ??, in Python `True` and `1` can be used interchangeably, the same applies between `False` and `0`. Therefore, adding `True` to an `int` is a valid operation.

So why don't we allow the the addition between `Time` object and Booleans?

We apply this restriction on the addition operator to preserve the semantic coherence of the `Time` type. The moral of the story is "It's not because we can do something that we should do it!". When building new types, it is essential to preserve semantic coherence for this type, and refrain from using any shortcut.

Exercise 16.5 In most cases it does not make sense to add a string to a `Time`. However, the statement `print(start + '11:05:03')` tries to add a `Time` object to a well formatted string representing a valid time. At the moment, the statement raises an exception, change the definition of `__add__` in order to accept well formatted string representing a time value.

Hint: The following strings are not well formatted time values:

- `' :10:15 '` needs at least one digit for hours. Can have more than two digits for hours, for example both `'102:10:15 '` and `'02:10:15 '` are valid.
- `'01:1:15 '` and `'01:10:1 '` are invalid. Both minutes and seconds need **exactly** two digits.
- `'01:75:60 '` is invalid for two reasons, minutes and seconds must be between 00 and 59.
- `'01:01:15.5 '` only whole seconds are allowed.

Exercise 16.6 Write an `add` method for `Points` that works with either a `Point` object or a tuple:

- If the second operand is a `Point`, the method should return a new `Point` whose `x` coordinate is the sum of the `x` coordinates of the operands, and likewise for the `y` coordinates.
- If the second operand is a tuple, the method should add the first element of the tuple to the `x` coordinate and the second element to the `y` coordinate, and return

a new Point with the result.

16.11 Polymorphism

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.

Many of the functions we wrote for strings will actually work for any kind of sequence. For example, in Section 11.1 we used `histogram` to count the number of times each letter appears in a word.

```
1  def histogram(s):
2      d = dict()
3      for c in s:
4          if c not in d:
5              d[c] = 1
6          else:
7              d[c] = d[c]+1
8      return d
```

This function also works for lists, tuples, and even dictionaries, as long as the elements of `s` are hashable, so they can be used as keys in `d`.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that can work with several types are called **polymorphic**. Polymorphism can facilitate code reuse. For example, the built-in function `sum`, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

Since Time objects provide an `add` method, they work with `sum`:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

In general, if all of the operations inside a function work with a given type, then the function works with that type.

The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

16.12 Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you are a stickler for type theory, it is a dubious practice to have objects of the same type with different attribute sets. It is usually a good idea to initialize all of an object's attributes in the `init` method.

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr` (see Section 14.7).

Another way to access the attributes of an object is through the special attribute `__dict__`, which is a dictionary that maps attribute names (as strings) and values:

```
>>> p = Point(3, 4)
>>> print p.__dict__
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
1 def print_attributes(obj):
2     for attr in obj.__dict__:
3         print attr, getattr(obj, attr)
```

`print_attributes` traverses the items in the object's dictionary and prints each attribute name and its corresponding value.

The built-in function `getattr` takes an object and an attribute name (as a string) and returns the attribute's value.

16.13 Glossary

object-oriented language: A language that provides features, such as user-defined classes and method syntax, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

subject: The object a method is invoked on.

operator overloading: Changing the behavior of an operator like `+` so it works with a user-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.

16.14 Exercises

Exercise 16.7 This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python.

1. Write a definition for a class named `Kangaroo` with the following methods:
 - (a) An `__init__` method that initializes an attribute named `pouch_contents` to an empty list.
 - (b) A method named `put_in_pouch` that takes an object of any type and adds it to `pouch_contents`.
 - (c) A `__str__` method that returns a string representation of the `Kangaroo` object and the contents of the pouch.

Test your code by creating two `Kangaroo` objects, assigning them to variables named `kanga` and `roo`, and then adding `roo` to the contents of `kanga`'s pouch.

2. Download `thinkpython.com/code/BadKangaroo.py`. It contains a solution to the previous problem with one big, nasty bug. Find and fix the bug. If you get stuck, you can download `thinkpython.com/code/GoodKangaroo.py`, which explains the problem and demonstrates a solution.



IV

Back Matter

Bibliography	178
Index	179

Bibliography

- [1] Arthur Conan Doyle. *The sign of four*. Broadview Press, 2010 (cited on page 5).
- [2] David Goodger and Guido Van Rossum. *PEP 257 – Docstring Conventions*. May 2001. URL: <https://www.python.org/dev/peps/pep-0257/> (cited on pages 22, 69).
- [3] David Goodger et al. *PEP 8 – Style Guide for Python Code*. July 2001. URL: <https://www.python.org/dev/peps/pep-0008/> (cited on pages 22, 67).
- [4] Google. *Google Python Style Guide*. Dec. 2018. URL: <https://github.com/google/styleguide/blob/gh-pages/pyguide.md> (cited on pages 68, 70).
- [5] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. 2nd. New York, NY, USA: McGraw-Hill, Inc., 1982. ISBN: 0070342075 (cited on page 66).
- [6] Joe Kunk. *To Comment or Not to Comment*. Jan. 2011. URL: <https://visualstudiomagazine.com/articles/2011/01/06/to-comment-or-not-to-comment.aspx> (cited on page 64).
- [7] Edward Loper. *Epydoc*. Jan. 2008. URL: <http://epydoc.sourceforge.net/index.html> (cited on page 73).
- [8] Edward Loper. *Epydoc Fields*. Jan. 2008. URL: <http://epydoc.sourceforge.net/epydoc.html#epydoc-fields> (cited on page 75).
- [9] Edward Loper. *The Epytext Markup Language*. Jan. 2008. URL: <http://epydoc.sourceforge.net/epytext.html> (cited on pages 73, 74).
- [10] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st edition. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0132350882, 9780132350884 (cited on page 68).

-
- [11] James Mertz. *Documenting Python Code: A Complete Guide*. July 2016. URL: <https://realpython.com/documenting-python-code/#commenting-vs-documenting-code> (cited on page 65).
 - [12] NumPyDoc. *NumPyDoc docstring guide*. June 2019. URL: <https://numpydoc.readthedocs.io/en/latest/format.html> (cited on page 71).
 - [13] Tim Peters. *PEP 20 – The Zen of Python*. Aug. 2004. URL: <https://www.python.org/dev/peps/pep-0020/> (cited on page 23).
 - [14] William Strunk. *The elements of style*. Penguin, 2007 (cited on page 67).
 - [15] Peter Vogel. *Why You Shouldn't Comment (or Document) Code*. June 2013. URL: <https://visualstudiomagazine.com/articles/2013/06/01/roc-rocks.aspx> (cited on page 65).

Index

- abecedarian, 89, 149
- absolute path, 141, 147
- access, 100
- accumulator, 112
 - list, 104
 - sum, 104
- Ackerman function, 84
- add method, 172
- addition with carrying, 43
- algorithm, 4, 10, 43
 - Euclid, 85
 - square root, 45
- aliasing, 107, 108, 112, 153, 155, 178
 - copying to avoid, 112
- alternative execution, 29
- ambiguity, 6
- anagram, 113
- anagram set, 135
- and operator, 28
- append method, 103, 110, 113
- argument, 47, 52, 54, 55, 63, 109
 - gather, 128
 - keyword, 133
 - list, 109
 - optional, 93, 107, 120
 - variable-length tuple, 128
- argument scatter, 129
- arithmetic operator, 16
- assert statement, 164
- assignment, 20, 35, 99
 - augmented, 104, 112
 - item, 91, 100, 126
 - multiple, 44
 - tuple, 127, 128, 130, 134
- assignment statement, 13
- attribute
 - __dict__, 177
 - initializing, 177
 - instance, 152, 157
- AttributeError, 157
- augmented assignment, 104, 112
- base case, 81
- binary search, 114
- bingo, 135
- birthday, 165
- birthday paradox, 113
- bisect module, 114
- bisection search, 114
- bisection, debugging by, 44
- body, 38, 52, 63
- bool type, 28
- boolean expression, 28
- boolean function, 160
- boolean operator, 93
- borrowing, subtraction with, 43, 163

- bracket
 - curly, 115
- bracket operator, 87, 100, 126
- branch, 30, 31
- break statement, 40
- bug, 4, 10
 - worst, 178
- calculator, 11, 21
- call graph, 123
- Car Talk, 124, 136
- carrying, addition with, 43, 161, 162
- case-sensitivity, variable names, 20
- chained conditional, 31
- character, 87
- class, 151, 157
 - Date, 165
 - Kangaroo, 178
 - Point, 151, 171
 - Rectangle, 153
 - Time, 159
- class definition, 151
- class object, 152, 157
- close method, 139
- Collatz conjecture, 39
- colon, 52
- commutativity, 19, 173
- comparison
 - string, 94
 - tuple, 133
- compile, 3, 10
- composition, 51, 55, 63
- compound statement, 29
- concatenation, 19, 21, 55, 89, 91, 107
 - list, 102, 110, 113
- condition, 29, 38
- conditional
 - chained, 31
 - nested, 32
- conditional execution, 29
- conditional statement, 29
- consistency check, 122, 163
- conversion
 - type, 48
- copy
 - deep, 156
 - shallow, 156
 - slice, 90, 102
 - to avoid aliasing, 112
- copy module, 155
- copying objects, 155
- count method, 93
- counter, 92, 97, 117
- counting and looping, 92
- crosswords, 137
- cummings, e. e., 4
- cumulative sum, 105
- curly bracket, 115
- data structure, 134
- Date class, 165
- datetime module, 165
- dead code, 83
- debugging, 4, 9, 19, 61, 95, 111, 122, 146, 157, 163, 177
 - by bisection, 44
 - experimental, 5
- declaration, 123
- decorate-sort-undecorate pattern, 133
- decrement, 36, 44
- deep copy, 156, 158
- deepcopy function, 156
- def keyword, 52
- default value, 171
 - avoiding mutable, 178
- definition
 - class, 151
 - function, 51
 - recursive, 136
- del operator, 105
- deletion, element of list, 105
- delimiter, 107, 112
- development plan
 - planned, 162
 - prototype and patch, 160, 162
- diagram
 - call graph, 123
 - object, 152, 154, 156, 158, 159
 - stack, 56, 109
 - state, 14, 35, 96, 100, 108, 132, 152, 154, 156, 159
- `__dict__` attribute, 177
- dict function, 115
- dictionary, 115, 123, 131

- initialize, 131
- invert, 120
- lookup, 119
- looping with, 119
- reverse lookup, 119
- traversal, 131, 177
- directory, 140, 147
 - walk, 141
 - working, 140
- dispatch
 - type-based, 176
- dispatch, type-based, 173
- divisibility, 27
- division
 - floating-point, 16
 - floor, 16, 62
 - integer division, 16
- divmod, 128, 162
- Docstring, 66
- docstring, 151
- documentation, 11
- dot notation, 50, 63, 92, 152
- DSU pattern, 133, 134
- duplicate, 113, 123
- element, 99, 112
- element deletion, 105
- elif keyword, 31
- else keyword, 29
- email address, 127
- embedded object, 154, 157, 178
 - copying, 156
- empty list, 99
- empty string, 96, 107
- encapsulation, 43, 92
- end of line character, 146
- enumerate function, 130
- epsilon, 43
- equality and assignment, 35
- equivalence, 108
- equivalent, 112
- error
 - runtime, 5, 19, 62, 82
 - semantic, 5, 13, 20, 96
 - syntax, 4, 19
- error message, 4, 5, 9, 13, 19
- Euclid's algorithm, 85
- eval function, 45
- evaluate, 16
- exception, 5, 10, 19
 - AttributeError, 157
 - IndexError, 88, 95, 100
 - IOError, 142
 - KeyError, 116
 - NameError, 56
 - OverflowError, 62
 - RuntimeError, 82
 - SyntaxError, 51
 - TypeError, 87, 91, 121, 126, 129, 140, 170
 - ValueError, 50, 119, 127
- exception, catching, 142
- executable, 3, 10
- exercise, secret, 147
- exists function, 141
- experimental debugging, 5
- expression, 16, 20
 - boolean, 28
- extend method, 103
- False special value, 28
- file, 137
 - writing, 139
- file object, 138
- filename, 140
- filter pattern, 104, 112
- find function, 91
- flag, 123
- float function, 48
- float type, 13
- floating-point, 20, 43
- floating-point division, 16
- floor division, 16, 20, 62
- flow diagram
 - if statement, 29
 - if-elif-else statement, 31
 - if-else statement, 30
 - nested if statements, 32
- flow of execution, 38, 53, 63
- folder, 140
- for loop, 37, 88, 101, 130
- formal language, 6, 10
- format operator, 139, 147
- format sequence, 139, 147

- format string, 139, 147
- frame, 56, 63, 81
- frequency, 118
 - letter, 135
- function, 51, 62, 166
 - ack, 84
 - deepcopy, 156
 - dict, 115
 - enumerate, 130
 - eval, 45
 - exists, 141
 - find, 91
 - float, 48
 - getattr, 177
 - getcwd, 140
 - hasattr, 157, 177
 - input, 49
 - int, 48
 - isinstance, 173
 - len, 64, 88, 116
 - list, 106
 - log, 50
 - max, 128, 129
 - min, 128, 129
 - open, 138, 139, 142
 - randint, 113
 - random, 133
 - recursive, 79
 - reload, 145
 - repr, 146
 - reversed, 134
 - sorted, 134
 - str, 48
 - sum, 129
 - tuple, 126
 - type, 157
 - zip, 129
- function argument, 54
- function call, 47, 63
- function definition, 51, 53, 62, 63
- function frame, 56, 63, 81
- function object, 52, 64
- function parameter, 54
- function type
 - modifier, 161
 - pure, 160
- function, math, 50
- function, non-void, 57
- function, reasons for, 58
- function, trigonometric, 50
- function, tuple as return value, 127
- function, void, 57
- functional programming style, 164
- gather, 128, 134
- GCD (greatest common divisor), 85
- generalization, 163
- get method, 118
- getattr function, 177
- getcwd function, 140
- global variable, 123
- greatest common divisor (GCD), 85
- grid, 65
- guardian pattern, 83, 95
- hasattr function, 157, 177
- hash function, 121, 123
- hashable, 121, 123, 131
- hashtable, 117, 123
- header, 52, 63
- Hello, World, 7
- help utility, 11
- hexadecimal, 152
- high-level language, 3, 10
- histogram, 118, 123
- homophone, 124
- identical, 112
- identity, 108
- IDLE, 8
 - launch, 8
- if statement, 29
- immutability, 91, 96, 109, 121, 125, 134
- implementation, 117, 123
- import statement, 63, 145
- in operator, 93, 100, 116
- increment, 36, 44, 161, 169
- incremental development, 83
- indentation, 52, 168
- index, 87, 95, 96, 100, 112, 115
 - looping with, 101
 - negative, 88
 - slice, 90, 102

- starting at zero, 87, 100
- IndexError, 88, 95, 100
- infinite loop, 38, 44
- infinite recursion, 82
- init method, 170, 177
- initialization
 - variable, 44
- initialization (before update), 36
- input function, 49
- instance, 152, 157
 - as argument, 153
 - as return value, 154
- instance attribute, 152, 157
- instantiation, 152
- int function, 48
- int type, 13
- integer, 20
- interactive mode, 3, 10, 15, 57
- interlocking words, 114
- interpret, 3, 10
- invariant, 163, 164
- invert dictionary, 120
- invocation, 92, 97
- IOError, 142
- is operator, 107, 156
- isinstance function, 173
- item, 96, 99
 - dictionary, 123
- item assignment, 91, 100, 126
- item update, 101
- items method, 131
- iteration, 35, 37, 44
- join method, 107
- Kangaroo class, 178
- key, 115, 123
- key-value pair, 115, 123, 131
- keyboard input, 49
- KeyError, 116
- keys method, 119
- keyword, 14, 15, 20
 - def, 52
 - elif, 31
 - else, 29
- keyword argument, 133
- language
 - formal, 6
 - high-level, 3
 - low-level, 3
 - natural, 6
 - programming, 2
 - safe, 5
- len function, 64, 88, 116
- letter frequency, 135
- letter rotation, 97, 123
- Linux, 6
- lipogram, 149
- list, 99, 106, 112, 134
 - as argument, 109
 - comprehension, 105
 - concatenation, 102, 110, 113
 - copy, 102
 - element, 100
 - empty, 99
 - function, 106
 - index, 100
 - membership, 100
 - method, 103
 - nested, 99, 101
 - of tuples, 129
 - operation, 102
 - repetition, 102
 - slice, 102
 - traversal, 101, 112
- literalness, 6
- local variable, 55, 63
- log function, 50
- logical operator, 28
- lookup, 123
- lookup, dictionary, 119
- loop, 37, 38, 130
 - for, 37, 88, 101
 - infinite, 38
 - traversal, 88
 - while, 37
- looping
 - with dictionaries, 119
 - with indices, 101
 - with strings, 92
- looping and counting, 92
- low-level language, 3, 10
- map pattern, 104, 112

- mapping, 100, 112
- math function, 50
- max function, 128, 129
- McCloskey, Robert, 89
- membership
 - binary search, 114
 - bisection search, 114
 - dictionary, 116
 - list, 100
 - set, 117
- memo, 123
- metaphor, method invocation, 168
- metathesis, 135
- method, 92, 97, 166, 177
 - `__str__`, 171
 - add, 172
 - append, 103, 110
 - close, 139
 - count, 93
 - extend, 103
 - get, 118
 - init, 170
 - items, 131
 - join, 107
 - keys, 119
 - pop, 105
 - radd, 174
 - readline, 138
 - remove, 105
 - setdefault, 122
 - sort, 103, 111, 133
 - split, 106, 127
 - string, 98
 - strip, 138
 - update, 131
 - values, 117
 - void, 103
- method append, 113
- method syntax, 168
- method, list, 103
- min function, 128, 129
- modifier, 161, 164
- module, 50, 63
 - bisect, 114
 - copy, 155
 - datetime, 165
 - os, 140
 - pprint, 122
 - random, 113, 133
 - reload, 145
 - urllib, 147
- module object, 50, 145
- module, writing, 144
- modulus operator, 27
- Monty Python and the Holy Grail, 160
- multiple assignment, 35, 44
- mutability, 91, 100, 102, 108, 125, 134, 155
- mutable object, as default value, 178
-
- NameError, 56
- natural language, 6, 10
- negative index, 88
- nested conditional, 32
- nested list, 99, 101, 112
- newline, 49
- Newton's method, 41
- non-void function, 57, 63
- None special value, 57, 83, 103, 106
- not operator, 28
-
- object, 91, 96, 107, 108, 112, 151
 - class, 152
 - copying, 155
 - embedded, 154, 157, 178
 - file, 138
 - function, 52, 64
 - module, 145
 - mutable, 155
 - printing, 167
- object code, 3, 10
- object diagram, 152, 154, 156, 158, 159
- object-oriented language, 177
- object-oriented programming, 166, 177
- open function, 138, 139, 142
- operand, 16, 20
- operator, 20
 - and, 28
 - boolean, 93
 - bracket, 87, 100, 126
 - del, 105
 - format, 139, 147
 - in, 93, 100, 116
 - is, 107, 156

- logical, 28
- modulus, 27
- not, 28
- or, 28
- overloading, 177
- relational, 28
- slice, 90, 97, 102, 110, 126
- string, 19
- truth table, 28
- update, 104
- operator overloading, 172
- operator, arithmetic, 16
- optional argument, 93, 107, 120
- optional parameter, 171
- or operator, 28
- order of operations, 18, 20
- os module, 140
- other (parameter name), 170
- OverflowError, 62
- overloading, 177
- override, 171
- palindrome, 84, 97
- parameter, 54, 56, 63, 109
 - gather, 128
 - optional, 171
 - other, 170
 - self, 168
- parentheses
 - argument in, 47
 - empty, 52, 92
 - matching, 4
 - overriding precedence, 18
 - parameters in, 54, 55
 - tuples in, 125
- parse, 6, 11
- pass statement, 29
- path, 140, 147
 - absolute, 141
 - relative, 141
- pattern
 - decorate-sort-undecorate, 133
 - DSU, 133
 - filter, 104, 112
 - guardian, 83, 95
 - map, 104, 112
 - reduce, 104, 112
 - search, 91, 97, 119
 - swap, 127
- PEMDAS, 18
- PEP 8, 22
- persistence, 137, 147
- pi, 46, 51
- plain text, 138
- planned development, 162, 164
- Point class, 151, 171
- point, mathematical, 151
- polymorphism, 176, 177
- pop method, 105
- portability, 3, 10
- pprint module, 122
- precedence, 20
- precondition, 113
- pretty print, 122
- print statement, 8, 11, 172
- problem solving, 2, 10
- program, 4, 10
- programming language, 2
- prompt, 3, 10, 49
- prototype and patch, 160, 162, 164
- pure function, 160, 164
- Puzzler, 124, 136
- Python 3.0, 16, 49, 129
- python.org, 11
- quotation mark, 8, 12, 13, 90
- radd method, 174
- radian, 50
- raise statement, 119, 164
- Ramanujan, Srinivasa, 46
- randint function, 113
- random function, 133
- random module, 113, 133
- readline method, 138
- Rectangle class, 153
- recursion, 79
 - base case, 81
 - infinite, 82
- recursive definition, 136
- reduce pattern, 104, 112
- reducible word, 124, 136
- redundancy, 6
- reference, 108, 109, 112

- aliasing, 108
- relational operator, 28
- relative path, 141, 147
- reload function, 145
- remove method, 105
- repetition, 36
 - list, 102
- repr function, 146
- representation, 151, 153
- return statement, 80
- return value, 47, 63, 154
 - tuple, 127
- reverse lookup, dictionary, 119, 123
- reverse word pair, 113
- reversed function, 134
- rotation
 - letters, 123
- rotation, letter, 97
- rules of precedence, 18, 20
- running pace, 11, 21, 165
- runtime error, 5, 19, 62, 82
- RuntimeError, 82
- safe language, 5
- sanity check, 122
- scaffolding, 83, 122
- scatter, 129, 134
- Scrabble, 135
- script, 3, 10
- script mode, 3, 10, 15, 57
- search, 119
- search pattern, 91, 97
- search, binary, 114
- search, bisection, 114
- secret exercise, 147
- self (parameter name), 168
- semantic error, 5, 10, 13, 20, 96
- semantics, 5, 10, 166
- sequence, 87, 96, 99, 106, 125, 134
- set
 - anagram, 135
- set membership, 117
- setdefault method, 122
- sexagesimal, 162
- shallow copy, 156, 158
- shape, 134
- sine function, 50
- singleton, 121, 123, 125
- slice, 96
 - copy, 90, 102
 - list, 102
 - string, 90
 - tuple, 126
 - update, 103
- slice operator, 90, 97, 102, 110, 126
- sort method, 103, 111, 133
- sorted function, 134
- source code, 3, 10
- special case, 161
- special value
 - False, 28
 - None, 57, 83, 103, 106
 - True, 28
- split method, 106, 127
- square root, 41
- stack diagram, 56, 63, 81, 83, 109
- state diagram, 14, 20, 35, 96, 100, 108, 132, 152, 154, 156, 159
- statement, 15, 20
 - assert, 164
 - assignment, 13, 35
 - break, 40
 - compound, 29
 - conditional, 29
 - for, 37, 88, 101
 - if, 29
 - import, 63, 145
 - pass, 29
 - print, 8, 11, 172
 - raise, 119, 164
 - return, 80
 - try, 142
 - while, 37
- step size, 97
- str function, 48
- __str__ method, 171
- string, 12, 20, 106, 134
 - comparison, 94
 - empty, 107
 - immutable, 91
 - method, 92
 - operation, 19
 - slice, 90

- string method, 98
- string representation, 146, 171
- string type, 13
- strip method, 138
- structure, 6
- subject, 168, 177
- subtraction
 - with borrowing, 43
- subtraction with borrowing, 163
- sum function, 129
- Swampy, 137
- swap pattern, 127
- syntax, 4, 10, 166
- syntax error, 4, 10, 19
- SyntaxError, 51
- temporary variable, 83
- testing
 - interactive mode, 4
- text
 - plain, 138
- text file, 147
- Time class, 159
- token, 6, 11
- traceback, 57, 61, 63, 82, 120
- traversal, 88, 91, 95, 96, 104, 112, 118, 119, 130, 133
 - dictionary, 177
 - list, 101
- traverse
 - dictionary, 131
- trigonometric function, 50
- True special value, 28
- try statement, 142
- tuple, 125, 127, 134
 - as key in dictionary, 131
 - assignment, 127
 - comparison, 133
 - in brackets, 132
 - singleton, 125
 - slice, 126
- tuple assignment, 128, 130, 134
- tuple function, 126
- type, 12, 13, 20
 - bool, 28
 - dict, 115
 - file, 137
 - float, 13
 - int, 13
 - list, 99
 - str, 13
 - tuple, 125
 - user-defined, 151, 159
- type conversion, 48
- type function, 157
- type-based dispatch, 173, 176, 177
- TypeError, 87, 91, 121, 126, 129, 140, 170
- underscore character, 14
- uniqueness, 113
- update, 36, 42, 44
 - item, 101
 - slice, 103
- update method, 131
- update operator, 104
- URL, 147
- urllib module, 147
- use before def, 19, 53
- user-defined type, 151, 159
- value, 12, 20, 107, 108, 123
 - tuple, 127
- ValueError, 50, 119, 127
- values method, 117
- variable, 13, 20
 - local, 55
 - temporary, 83
 - updating, 36
- variable-length argument tuple, 128
- void function, 57, 63
- void method, 103
- walk, directory, 141
- while loop, 37
- whitespace, 61, 146, 147
- word count, 144
- word, reducible, 124, 136
- working directory, 140
- worst bug, 178
- zero, index starting at, 87, 100
- zip function, 129
 - use with dict, 131