

# How to Think Like a Computer Scientist

The PreTeXt Interactive Edition

# How to Think Like a Computer Scientist

The PreTeXt Interactive Edition



# Preface to the First and Second Editions

**by Jeffrey Elkner.** This book owes its existence to the collaboration made possible by the Internet and the free software movement. Its three authors—a college professor, a high school teacher, and a professional programmer—never met face to face to work on it, but we have been able to collaborate closely, aided by many other folks who have taken the time and energy to send us their feedback.

We think this book is a testament to the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

**How and why I came to use Python.** In 1999, the College Board’s Advanced Placement (AP) Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum at Yorktown High School in Arlington, Virginia, where I teach. Up to this point, Pascal was the language of instruction in both our first-year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first year course for the 1997-98 school year so that we would be in step with the College Board’s change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++’s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there had to be a better language choice for our first-year class, I went looking for an alternative to C++.

I needed a language that would run on the machines in our GNU/Linux lab as well as on the Windows and Macintosh platforms most students have at home. I wanted it to be free software, so that students could use it at home regardless of their income. I wanted a language that was used by professional programmers, and one that had an active developer community around it. It had to support both procedural and object-oriented programming. And most importantly, it had to be easy to learn and teach. When I investigated the choices with these goals in mind, Python stood out as the best candidate for the job.

I asked one of Yorktown’s talented students, Matt Ahrens, to give Python a try. In two months he not only learned the language but wrote an application called pyTicket that enabled our staff to report technology problems via the

Web. I knew that Matt could not have finished an application of that scale in so short a time in C++, and this accomplishment, combined with Matt's positive assessment of Python, suggested that Python was the solution I was looking for.

**Finding a textbook.** Having decided to use Python in both of my introductory computer science classes the following year, the most pressing problem was the lack of an available textbook.

Free documents came to the rescue. Earlier in the year, Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational materials. Allen had already written a first-year computer science textbook, *How to Think Like a Computer Scientist*. When I read this book, I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming rather than the features of a particular language. Reading it immediately made me a better teacher.

*How to Think Like a Computer Scientist* was not just an excellent book, but it had been released under the GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version of the book into the new language. While I would not have been able to write a textbook on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative development model used so well in software could also work for educational materials.

Working on this book for the last two years has been rewarding for both my students and me, and my students played a big part in the process. Since I could make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for mistakes in the book by giving them a bonus point each time they made a suggestion that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object-oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book sat in an unfinished state for the better part of a year until the open source community once again provided the needed means for its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene, Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our Website at <http://openbookproject.net><sup>1</sup> called **\*Python for Fun\***<sup>2</sup> and was working with some of my most advanced students as a master teacher, guiding them beyond where I could take them.

**Introducing programming with Python.** The process of translating and using *How to Think Like a Computer Scientist* for the past two years has confirmed Python's suitability for teaching beginning students. Python greatly

---

<sup>1</sup><http://openbookproject.net>

<sup>2</sup><http://openbookproject.net/py4fun>

simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text illustrates this point. It is the traditional hello, world program, which in the Java version of the book looks like this:

```
class Hello {  
  
    public static void main (String[] args) {  
        System.out.println ("Hello, world.");  
    }  
}
```

in the Python version it becomes:

```
print "Hello, world!"
```

Even though this is a trivial example, the advantages of Python stand out. Yorktown's Computer Science I course has no prerequisites, so many of the students seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The Java version has always forced me to choose between two unsatisfying options: either to explain the class Hello, public static void main, String[] args, {}, and {}, statements and risk confusing or intimidating some of the students right at the start, or to tell them, Just don't worry about all of that stuff now; we will talk about it later, and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming statement and to get them to write their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing the explanatory text of the program in each version of the book further illustrates what this means to the beginning student. There are seven paragraphs of explanation of Hello, world! in the Java version; in the Python version, there are only a few sentences. More importantly, the missing six paragraphs do not deal with the big ideas in computer programming but with the minutia of Java syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high-level language like Python allows a teacher to postpone talking about low-level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put first things first pedagogically. One of the best examples of this is the way in which Python handles variables. In Java a variable is a name for a place that holds a value if it is a built-in type, and a reference to an object if it is not. Explaining this distinction requires a discussion of how the computer stores data. Thus, the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both computer science and algebra). Bytes and addresses do not help the matter. In Python a variable is a name that refers to a thing. This is a far more intuitive concept for beginning students and is much closer to the meaning of variable that they learned in their math courses. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aids in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related

distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the keyword `def`, so I simply tell my students, When you define a function, begin with `def`, followed by the name of the function that you are defining; when you call a function, simply call (type) out its name. Parameters go with definitions; arguments go with calls. There are no return types, parameter types, or reference and value parameters to get in the way, so I am now able to teach functions in less than half the time that it previously took me, with better comprehension.

Using Python improved the effectiveness of our computer science program for all students. I saw a higher general level of success and a lower level of frustration than I experienced teaching with either C++ or Java. I moved faster with better results. More students left the course with the ability to create meaningful programs and with the positive attitude toward the experience of programming that this engenders.

**Building a community.** I have received email from all over the globe from people using this book to learn or to teach programming. A user community has begun to emerge, and many people have been contributing to the project by sending in materials for the companion Website at <http://openbookproject.net/pybiblio><sup>3</sup>.

With the continued growth of Python, I expect the growth in the user community to continue and accelerate. The emergence of this user community and the possibility it suggests for similar collaboration among educators have been the most exciting parts of working on this project for me. By working together, we can increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you. Please write to me at [jeff@elkner.net](mailto:jeff@elkner.net)<sup>4</sup>.Jeffrey Elkner  
Governor's Career and Technical Academy in Arlington  
Arlington, Virginia

---

<sup>3</sup><http://openbookproject.net/pybiblio>

<sup>4</sup><mailto:jeff@elkner.net>

# Preface to the Interactive Edition

by Brad Miller and David Ranum

There are many books available if you want to learn to program in Python. In fact, we have written a few of them ourselves. However, none of the books are like this one.

Programming is not a “spectator sport”. It is something you do, something you participate in. It would make sense, then, that the book you use to learn programming should allow you to be active. That is our goal.

This book is meant to provide you with an interactive experience as you learn to program in Python. You can read the text, watch videos, and write and execute Python code. In addition to simply executing code, there is a unique feature called ‘codelens’ that allows you to control the flow of execution in order to gain a better understanding of how the program works..

We have tried to use these different presentation techniques where they are most appropriate. In other words, sometimes it might be best to read a description of some aspect of Python. On a different occasion, it might be best to execute a small example program. Often we give you many different options for covering the material. Our hope is that you will find that your understanding will be enhanced because you are able to experience it in more than just one way.

# Contents

Preface to the First and Second Editions	iv
Preface to the Interactive Edition	viii
<b>1 General Introduction</b>	<b>1</b>
1.1 The Way of the Program . . . . .	1
1.2 Algorithms . . . . .	1
1.3 The Python Programming Language . . . . .	2
1.4 Executing Python in this Book . . . . .	4
1.5 More About Programs . . . . .	6
1.6 What is Debugging? . . . . .	7
1.7 Syntax errors . . . . .	7
1.8 Runtime Errors . . . . .	8
1.9 Semantic Errors . . . . .	8
1.10 Experimental Debugging . . . . .	9
1.11 Formal and Natural Languages . . . . .	9
1.12 A Typical First Program . . . . .	11
1.13 Comments . . . . .	12
1.14 Glossary . . . . .	12
1.15 Exercises . . . . .	14
<b>2 Simple Python Data</b>	<b>15</b>
2.1 Variables, Expressions and Statements . . . . .	15
2.2 Values and Data Types . . . . .	15
2.3 Type conversion functions . . . . .	18
2.4 Variables . . . . .	19
2.5 Variable Names and Keywords . . . . .	21
2.6 Statements and Expressions . . . . .	22
2.7 Operators and Operands . . . . .	23
2.8 Input . . . . .	25
2.9 Order of Operations . . . . .	27
2.10 Reassignment . . . . .	28
2.11 Updating Variables . . . . .	30
2.12 Glossary . . . . .	32
2.13 Exercises . . . . .	33

**3 Debugging Interlude 1** **35**

3.1 How to be a Successful Programmer . . . . .	35
3.2 How to Avoid Debugging . . . . .	35
3.3 Beginning tips for Debugging . . . . .	38
3.4 Know Your Error Messages . . . . .	38
3.5 Summary . . . . .	44
3.6 Exercises . . . . .	44

**4 Python Turtle Graphics** **45**

4.1 Hello Little Turtles! . . . . .	45
4.2 Our First Turtle Program . . . . .	45
4.3 Instances — A Herd of Turtles . . . . .	55
4.4 The <code>for</code> Loop . . . . .	59
4.5 Flow of Execution of the <code>for</code> Loop . . . . .	60
4.6 Iteration Simplifies our Turtle Program . . . . .	60
4.7 The <code>range</code> Function . . . . .	64
4.8 A Few More turtle Methods and Observations . . . . .	67
4.9 Summary of Turtle Methods . . . . .	71
4.10 Glossary . . . . .	71
4.11 Exercises . . . . .	72

**5 Python Modules** **75**

5.1 Modules and Getting Help . . . . .	75
5.2 More About Using Modules . . . . .	78
5.3 The <code>math</code> module . . . . .	78
5.4 The <code>random</code> module . . . . .	79
5.5 Creating Modules . . . . .	81
5.6 Glossary . . . . .	87
5.7 Exercises . . . . .	87

**6 Functions** **88**

6.1 Functions . . . . .	88
6.2 Functions that Return Values . . . . .	93
6.3 Unit Testing . . . . .	96
6.4 Variables and Parameters are Local . . . . .	98
6.5 The Accumulator Pattern . . . . .	100
6.6 Functions can Call Other Functions . . . . .	103
6.7 Flow of Execution Summary . . . . .	105
6.8 Using a Main Function . . . . .	107
6.9 Program Development . . . . .	109
6.10 Composition . . . . .	112
6.11 A Turtle Bar Chart . . . . .	112
6.12 Glossary . . . . .	115
6.13 Exercises . . . . .	115

**7 Selection** **122**

7.1 Boolean Values and Boolean Expressions . . . . .	122
7.2 Logical operators . . . . .	123
7.3 Precedence of Operators . . . . .	126
7.4 Conditional Execution: Binary Selection . . . . .	127

7.5 Omitting the else Clause: Unary Selection . . . . .	129
7.6 Nested conditionals . . . . .	131
7.7 Chained conditionals . . . . .	132
7.8 Boolean Functions . . . . .	134
7.9 Glossary . . . . .	136
7.10 Exercises . . . . .	137
<b>8 More About Iteration</b>	<b>143</b>
8.1 Iteration Revisited . . . . .	143
8.2 The <code>for</code> loop revisited . . . . .	143
8.3 The <code>while</code> Statement . . . . .	144
8.4 Randomly Walking Turtles . . . . .	148
8.5 The $3n + 1$ Sequence . . . . .	151
8.6 Newton's Method . . . . .	152
8.7 The Accumulator Pattern Revisited . . . . .	153
8.8 Other uses of <code>while</code> . . . . .	154
8.9 Algorithms Revisited . . . . .	156
8.10 Simple Tables . . . . .	156
8.11 2-Dimensional Iteration: Image Processing . . . . .	157
8.12 Image Processing on Your Own . . . . .	165
8.13 Glossary . . . . .	165
8.14 Exercises . . . . .	166
<b>9 Strings</b>	<b>169</b>
9.1 Strings Revisited . . . . .	169
9.2 A Collection Data Type . . . . .	169
9.3 Operations on Strings . . . . .	169
9.4 Index Operator: Working with the Characters of a String . . . . .	171
9.5 String Methods . . . . .	172
9.6 Length . . . . .	175
9.7 The Slice Operator . . . . .	176
9.8 String Comparison . . . . .	177
9.9 Strings are Immutable . . . . .	179
9.10 Traversal and the <code>for</code> Loop: By Item . . . . .	180
9.11 Traversal and the <code>for</code> Loop: By Index . . . . .	181
9.12 Traversal and the <code>while</code> Loop . . . . .	182
9.13 The <code>in</code> and <code>not in</code> operators . . . . .	182
9.14 The Accumulator Pattern with Strings . . . . .	183
9.15 Turtles and Strings and L-Systems . . . . .	184
9.16 Looping and Counting . . . . .	188
9.17 A <code>find</code> function . . . . .	188
9.18 Optional parameters . . . . .	189
9.19 Character classification . . . . .	191
9.20 Summary . . . . .	191
9.21 Glossary . . . . .	192
9.22 Exercises . . . . .	193
<b>10 Lists</b>	<b>202</b>
10.1 Lists . . . . .	202
10.2 List Values . . . . .	202
10.3 List Length . . . . .	203

10.4 Accessing Elements . . . . .	203
10.5 List Membership . . . . .	204
10.6 Concatenation and Repetition . . . . .	204
10.7 List Slices . . . . .	206
10.8 Lists are Mutable . . . . .	206
10.9 List Deletion . . . . .	207
10.10 Objects and References . . . . .	207
10.11 Aliasing . . . . .	209
10.12 Cloning Lists . . . . .	210
10.13 Repetition and References . . . . .	211
10.14 List Methods . . . . .	213
10.15 The Return of L-Systems . . . . .	215
10.16 Append versus Concatenate . . . . .	217
10.17 Lists and for loops . . . . .	217
10.18 The Accumulator Pattern with Lists . . . . .	219
10.19 Using Lists as Parameters . . . . .	222
10.20 Pure Functions . . . . .	223
10.21 Which is Better? . . . . .	223
10.22 Functions that Produce Lists . . . . .	224
10.23 List Comprehensions . . . . .	224
10.24 Nested Lists . . . . .	225
10.25 Strings and Lists . . . . .	225
10.26 List - Type Conversion Function . . . . .	226
10.27 Tuples and Mutability . . . . .	227
10.28 Tuple Assignment . . . . .	228
10.29 Tuples as Return Values . . . . .	228
10.30 Glossary . . . . .	229
10.31 Exercises . . . . .	229
<b>11 Files</b>	<b>235</b>
11.1 Working with Data Files . . . . .	235
11.2 Finding a File on your Disk . . . . .	235
11.3 Reading a File . . . . .	237
11.4 Iterating over lines in a file . . . . .	237
11.5 Alternative File Reading Methods . . . . .	238
11.6 Writing Text Files . . . . .	240
11.7 With Statements . . . . .	241
11.8 Fetching Something From The Web . . . . .	242
11.9 Glossary . . . . .	242
11.10 Exercises . . . . .	243
<b>12 Dictionaries</b>	<b>244</b>
12.1 Dictionaries . . . . .	244
12.2 Dictionary Operations . . . . .	245
12.3 Dictionary Methods . . . . .	246
12.4 Aliasing and Copying . . . . .	249
12.5 Sparse Matrices . . . . .	249
12.6 Glossary . . . . .	251
12.7 Exercises . . . . .	251

**13 Exceptions 255**

13.1 What is an exception? . . . . .	255
13.2 Runetime Stack and <code>raise</code> command. . . . .	255
13.3 Standard Exceptions. . . . .	257
13.4 Principles for using Exceptions . . . . .	259
13.5 Exceptions Syntax . . . . .	260
13.6 The <code>finally</code> clause of the <code>try</code> statement . . . . .	262
13.7 Glossary . . . . .	263
13.8 Exercises . . . . .	263

**14 Web Applications 264**

14.1 Web Applications . . . . .	264
14.2 How the Web Works. . . . .	264
14.3 How Web Applications Work . . . . .	265
14.4 Web Applications and HTML Forms . . . . .	266
14.5 Writing Web Applications With Flask . . . . .	268
14.6 More About Flask . . . . .	269
14.7 Input For A Flask Web Application . . . . .	272
14.8 Web Applications With a User Interface . . . . .	274
14.9 Glossary . . . . .	275

**15 GUI and Event Driven Programming 276**

15.1 Graphical User Interfaces . . . . .	276
15.2 Tkinter Standard Dialog Boxes . . . . .	276
15.3 GUI Widgets . . . . .	279
15.4 Layout Managers . . . . .	280
15.5 Widget Groupings . . . . .	281
15.6 Command Events . . . . .	282
15.7 Low-Level Event Processing . . . . .	282
15.8 The Design of GUI Programs . . . . .	283
15.9 Common Widget Properties . . . . .	285
15.10 Timer Events . . . . .	285
15.11 A Programming Example . . . . .	287
15.12 Managing GUI Program Complexity . . . . .	299
15.13 Exercises . . . . .	301
15.14 Glossary . . . . .	301

**16 Recursion 302**

16.1 What Is Recursion? . . . . .	302
16.2 Calculating the Sum of a List of Numbers . . . . .	302
16.3 The Three Laws of Recursion . . . . .	304
16.4 Converting an Integer to a String in Any Base . . . . .	305
16.5 Visualizing Recursion . . . . .	308
16.6 Sierpinski Triangle . . . . .	311
16.7 Glossary . . . . .	313
16.8 Programming Exercises. . . . .	314
16.9 Exercises . . . . .	316

<b>17 Classes and Objects - the Basics</b>	<b>317</b>
17.1 Object-oriented programming . . . . .	317
17.2 A change of perspective . . . . .	317
17.3 Objects Revisited . . . . .	318
17.4 User Defined Classes . . . . .	319
17.5 Improving our Constructor . . . . .	322
17.6 Adding Other Methods to our Class . . . . .	323
17.7 Objects as Arguments and Parameters . . . . .	324
17.8 Converting an Object to a String . . . . .	325
17.9 Instances as Return Values . . . . .	326
17.10 Glossary . . . . .	327
17.11 Exercises . . . . .	328
<b>18 Classes and Objects - Digging a Little Deeper</b>	<b>329</b>
18.1 Fractions . . . . .	329
18.2 Objects are Mutable . . . . .	330
18.3 Sameness . . . . .	331
18.4 Arithmetic Methods . . . . .	333
18.5 Glossary . . . . .	335
18.6 Exercises . . . . .	335
<b>19 Inheritance</b>	<b>337</b>
19.1 Pillars of OOP . . . . .	337
19.2 Introduction to Inheritance . . . . .	337
19.3 Extending . . . . .	338
19.4 Reuse Through Composition . . . . .	339
19.5 Class Diagrams . . . . .	340
19.6 Composition vs. Inheritance . . . . .	341
19.7 Case Study: Structured Postal Addresses . . . . .	342
<b>20 Unit Testing</b>	<b>348</b>
20.1 Introduction: Unit Testing . . . . .	348
20.2 Checking Assumptions With <code>assert</code> . . . . .	348
20.3 Testing Functions . . . . .	352
20.4 Designing Testable Functions . . . . .	356
20.5 Writing Unit Tests . . . . .	359
20.6 Test-First Development . . . . .	361
20.7 Testing with <code>pytest</code> . . . . .	364
20.8 Glossary . . . . .	368
20.9 Exercises . . . . .	369
<b>21 Labs</b>	<b>372</b>
21.1 Astronomy Animation . . . . .	372
21.2 Turtle Racing Lab . . . . .	375
21.3 Drawing a Circle . . . . .	377
21.4 Lessons from a Triangle . . . . .	378
21.5 Counting Letters . . . . .	378
21.6 Letter Count Histogram . . . . .	379
21.7 Approximating the Value of Pi . . . . .	380
21.8 Python Beyond the Browser . . . . .	382

21.9 Experimenting With the 3n+1 Sequence . . . . .	384
21.10 Plotting a sine Wave. . . . .	385

## Back Matter

21.11 Operator precedence table . . . . .	389
---	-----

# Chapter 1

# General Introduction

## 1.1 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.

## 1.2 Algorithms

If problem solving is a central part of computer science, then the solutions that you create through the problem solving process are also important. In computer science, we refer to these solutions as **algorithms**. An algorithm is a step by step list of instructions that if followed exactly will solve the problem under consideration.

Our goal in computer science is to take a problem and develop an algorithm that can serve as a general solution. Once we have such a solution, we can use our computer to automate the execution. As noted above, programming is a skill that allows a computer scientist to take an algorithm and represent it in a notation (a program) that can be followed by a computer. These programs are written in **programming languages**.

### Check your understanding

**Checkpoint 1.2.1** What is the most important skill for a computer scientist?

- A. To think like a computer.
- B. To be able to write code really well.
- C. To be able to solve problems.

- D. To be really good at math.

**Checkpoint 1.2.2** An algorithm is:

- A. A solution to a problem that can be solved by a computer.
- B. A step by step list of instructions that if followed exactly will solve the problem under consideration.
- C. A series of instructions implemented in a programming language.
- D. A special kind of notation used by computer scientists.

## 1.3 The Python Programming Language

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

As you might infer from the name high-level language, there are also **low-level languages**, sometimes referred to as machine languages or assembly languages. Machine language is the encoding of instructions in binary so that they can be directly executed by the computer. Assembly language uses a slightly easier format to refer to the low level instructions. Loosely speaking, computers can only execute programs written in low-level languages. To be exact, computers can actually only execute programs written in machine language. Thus, programs written in a high-level language (and even those in assembly language) have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages. However, the advantages to high-level languages are enormous.

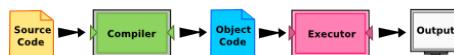
First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of 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 case, 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.



Many modern languages use both processes. They are first compiled into a lower level language, called **byte code**, and then interpreted by a program called a **virtual machine**. Python uses both processes, but because of the way programmers interact with it, it is usually considered an interpreted language.

There are two ways to use the Python interpreter: *shell mode* and *program mode*. In shell mode, you type Python expressions into the **Python shell**, and the interpreter immediately shows the result. The example below shows the Python shell at work.

```
$ python3
Python 3.2 (r32:88445, Mar 25 2011, 19:28:28)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 3
5
>>>
```

The `>>>` is called the **Python prompt**. The interpreter uses the prompt to indicate that it is ready for instructions. We typed `2 + 3`. The interpreter evaluated our expression and replied `5`. On the next line it gave a new prompt indicating that it is ready for more input.

Working directly in the interpreter is convenient for testing short bits of code because you get immediate feedback. Think of it as scratch paper used to help you work out problems.

Alternatively, you can write an entire program by placing lines of Python instructions in a file and then use the interpreter to execute the contents of the file as a whole. Such a file is often referred to as **source code**. For example, we used a text editor to create a source code file named `firstprogram.py` with the following contents:

```
| print("My first program adds two numbers, 2 and 3:")
| print(2 + 3)
```

By convention, files that contain Python programs have names that end with `.py`. Following this convention will help your operating system and other programs identify a file as containing python code.

```
$ python firstprogram.py
My first program adds two numbers, 2 and 3:
5
```

These examples show Python being run from a Unix command line. In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.

**Note 1.3.1 Want to learn more about Python?** If you would like to learn more about installing and using Python, here are some video links. [Installing Python for Windows](#)<sup>1</sup> shows you how to install the Python environment under Windows Vista, [Installing Python for Mac](#)<sup>2</sup> shows you how to install under Mac OS/X, and [Installing Python for Linux](#)<sup>3</sup> shows you how to install from the Linux command line. [Using Python](#)<sup>4</sup> shows you some details about the Python shell and source code.

### Check your understanding

---

<sup>1</sup><http://youtu.be/9EfGpN1Pnsg>

<sup>2</sup><http://youtu.be/MEmEJCLLI2k>

<sup>3</sup><http://youtu.be/RLPYBxfAud4>

<sup>4</sup>[http://youtu.be/kXbpB5\\_ywDw](http://youtu.be/kXbpB5_ywDw)

**Checkpoint 1.3.2** Source code is another name for:

- A. the instructions in a program, stored in a file.
- B. the language that you are programming in (e.g., Python).
- C. the environment/tool in which you are programming.
- D. the number (or "code") that you must input at the top of each program to tell the computer how to execute your program.

**Checkpoint 1.3.3**

What is the difference between a high-level programming language and a low-level programming language?

- A. It is high-level if you are standing and low-level if you are sitting.
- B. It is high-level if you are programming for a computer and low-level if you are programming for a phone or mobile device.
- C. It is high-level if the program must be processed before it can run, and low-level if the computer can execute it without additional processing.
- D. It is high-level if it is easy to program in and is very short; it is low-level if it is really hard to program in and the programs are really long.

**Checkpoint 1.3.4** Pick the best replacements for 1 and 2 in the following sentence: When comparing compilers and interpreters, a compiler is like 1 while an interpreter is like 2.

- A. 1 = a process, 2 = a function
- B. 1 = translating an entire book, 2 = translating a line at a time
- C. 1 = software, 2 = hardware
- D. 1 = object code, 2 = byte code

## 1.4 Executing Python in this Book



This book provides two special ways to execute Python programs. Both techniques are designed to assist you as you learn the Python programming language. They will help you increase your understanding of how Python programs work.

First, you can write, modify, and execute programs using a unique **active-code** interpreter that allows you to execute Python code right in the text itself (right from the web browser). Although this is certainly not the way real

programs are written, it provides an excellent environment for learning a programming language like Python since you can experiment with the language as you are reading.

```
print("My first program adds two numbers , 2 and 3:")
print(2 + 3)
```

Take a look at the activecode interpreter in action.

What you see depends on whether you are *logged in* or not! Code that you write and run is saved for all future sessions only if you are *logged in*! If you are logged in, you should see a green *Save & Run* button. If you are not logged in you see only *Run*. In the next discussion we will refer to both variants as the *Run* button.

If we use the Python code from the previous example and make it active, you will see that it can be executed directly by pressing the *Run* button. Try pressing the *Run* (or *Save & Run*) button above.

Now try modifying the activecode program shown above. First, modify the string in the first print statement by changing the word *adds* to the word *multiples*. Now press *Run*. You can see that the result of the program has changed. However, it still prints “5” as the answer. Modify the second print statement by changing the addition symbol, the “+”, to the multiplication symbol, “\*”. Press *Run* to see the new results.

As the name suggests, *Save & Run* also *saves* your latest version of the code, and you can recover it even in later sessions when *logged in*. If *not* logged in, *Run* saves versions *only until your browser leaves the current web page*, and then you lose all modifications.

After you have run your code the first time, a *Load History* button that was beside the *Run* button turns into a slider. If you click on the slider location box, you can use your left and right arrow buttons to switch to other versions you ran. Alternately you can drag the box on the slider. Now move the slider to see a previously saved state. You can just run it by pressing *Run*, or edit and then save and run it as the latest version.

In addition to activecode, you can also execute Python code with the assistance of a unique visualization tool. This tool, known as **codelens**, allows you to control the step by step execution of a program. It also lets you see the values of all variables (introduced in [Section 2.4](#)) as they are created and modified. The following example shows codelens in action on the same program as we saw above. Note that in activecode, the source code executes from beginning to end and you can see the final result. In codelens you can see and control the step by step progress. Note that the red arrow always points to the next line of code that is going to be executed. The light green arrow points to the line that was just executed.

```
print("My first program adds two numbers , 2 and 3:")
print(2 + 3)
```

The examples in this book use a mixture of the standard Python interpreter, source code, activecode, and codelens. You will be able to tell which is which by looking for either the Python prompt in the case of a shell mode program, the *run* button for the activecode, or the *forward/backward* buttons for codelens.

### Check your understanding

**Checkpoint 1.4.1** The activecode interpreter allows you to (select all that apply):

- A. save programs and reload saved programs.
- B. type in Python source code.

- C. execute Python code right in the text itself within the web browser.
- D. receive a yes/no answer about whether your code is correct or not.

**Checkpoint 1.4.2** Codelens allows you to (select all that apply):

- A. measure the speed of a program's execution.
- B. control the step by step execution of a program.
- C. write and execute your own Python code.
- D. execute the Python code that is in codelens.

## 1.5 More About Programs

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something as complex as rendering an html page in a web browser or encoding a video and streaming it across the network. It can also be a symbolic computation, such as searching for 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.

<b>input</b>	Get data from the keyboard, a file, or some other device.
<b>output</b>	Display data on the screen or send data to a file or other device.
<b>math and logic</b>	Perform basic mathematical operations like addition and multiplication and logical operations like <code>and</code> , <code>or</code> , and <code>not</code> .
<b>conditional execution</b>	Check for certain conditions and execute the appropriate sequence of statements.
<b>repetition</b>	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 more or less like these. Thus, we can describe 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 sequences of these basic instructions.

### Check your understanding

**Checkpoint 1.5.1** A program is:

- A. a sequence of instructions that specifies how to perform a computation.
- B. something you follow along at a play or concert.
- C. a computation, even a symbolic computation.
- D. the same thing as an algorithm.

## 1.6 What is Debugging?

Programming is a complex process. Since it is done by human beings, errors may often occur. Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**. Some claim that in 1945, a dead moth caused a problem on relay number 70, panel F, of one of the first computers at Harvard, and the term **bug** has remained in use since. For more about this historic event, see [first bug](#)<sup>1</sup>.

Three kinds of errors can occur in a program: [syntax errors](#)<sup>2</sup>, [runtime errors](#)<sup>3</sup>, and [semantic errors](#)<sup>4</sup>. It is useful to distinguish between them in order to track them down more quickly.

### Check your understanding

**Checkpoint 1.6.1** Debugging is:

- A. tracking down programming errors and correcting them.
- B. removing all the bugs from your house.
- C. finding all the bugs in the program.
- D. fixing the bugs in the program.

## 1.7 Syntax errors

Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without problems. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will display an error message and quit. You will not be able to complete the execution of your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. However, as you gain experience, you will make fewer errors and you will also be able to find your errors faster.

### Check your understanding

**Checkpoint 1.7.1** Which of the following is a syntax error?

- A. Attempting to divide by 0.
- B. Forgetting a colon at the end of a statement where one is required.
- C. Forgetting to divide by 100 when printing a percentage amount.

**Checkpoint 1.7.2** Who or what typically finds syntax errors?

- A. The programmer.
- B. The compiler / interpreter.

---

<sup>1</sup><http://en.wikipedia.org/wiki/File:H96566k.jpg>

<sup>2</sup>[http://en.wikipedia.org/wiki/Syntax\\_error](http://en.wikipedia.org/wiki/Syntax_error)

<sup>3</sup>[http://en.wikipedia.org/wiki/Runtime\\_error](http://en.wikipedia.org/wiki/Runtime_error)

<sup>4</sup>[http://en.wikipedia.org/wiki/Logic\\_error](http://en.wikipedia.org/wiki/Logic_error)

- C. The computer.
- D. The teacher / instructor.

## 1.8 Runtime Errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. 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.

### Check your understanding

**Checkpoint 1.8.1** Which of the following is a run-time error?

- A. Attempting to divide by 0.
- B. Forgetting a colon at the end of a statement where one is required.
- C. Forgetting to divide by 100 when printing a percentage amount.

**Checkpoint 1.8.2** Who or what typically finds runtime errors?

- A. The programmer.
- B. The interpreter.
- C. The computer.
- D. The teacher / instructor.

## 1.9 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. However, your program 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.

### Check your understanding

**Checkpoint 1.9.1** Which of the following is a semantic error?

- A. Attempting to divide by 0.
- B. Forgetting a colon at the end of a statement where one is required.
- C. Forgetting to divide by 100 when printing a percentage amount.

**Checkpoint 1.9.2** Who or what typically finds semantic errors?

- A. The programmer.
- B. The compiler / interpreter.
- C. The computer.

- D. The teacher / instructor.

## 1.10 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 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, When you have eliminated the impossible, whatever remains, however improbable, must be the truth. (A. Conan Doyle, *The Sign of Four*)

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 kernel that contains millions 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 displaying 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.

### Check your understanding

**Checkpoint 1.10.1** The difference between programming and debugging is:

- A. programming is the process of writing and gradually debugging a program until it does what you want.
- B. programming is creative and debugging is routine.
- C. programming is fun and debugging is work.
- D. there is no difference between them.

## 1.11 Formal and Natural Languages

**Natural languages** are the languages that 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.

**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  $3=+6\$$  is not.  $H2O$  is a syntactically correct chemical name, but  $2Zz$  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, and chemical elements. One of the problems with  $3=+6\$$  is that  $\$$  is not a legal token in mathematics (at least as far as we know). Similarly,  $2Zz$  is not legal because there is no element with the abbreviation  $Zz$ .

The second type of syntax rule pertains to the **structure** of a statement—that is, the way the tokens are arranged. The statement  $3=+6\$$  is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

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 other shoe fell”, you understand that the other shoe is the subject and fell is the verb. 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 shoe is and what it means to fall, 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 many differences:

## Glossary

**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.** Formal languages mean exactly what they say. On the other hand, natural languages are full of idiom and metaphor. If someone says, “The other shoe fell”, there is probably no shoe and nothing falling.

**Note 1.11.1** You'll need to find the original joke to understand the idiomatic meaning of the other shoe falling. *Yahoo! Answers* thinks it knows!

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

## Glossary

**poetry.** Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

**prose.** The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry

but still often ambiguous.

**program.** The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

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. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

### Check your understanding

**Checkpoint 1.11.2** The differences between natural and formal languages include:

- A. natural languages can be parsed while formal languages cannot.
- B. ambiguity, redundancy, and literalness.
- C. there are no differences between natural and formal languages.
- D. tokens, structure, syntax, and semantics.

**Checkpoint 1.11.3** True or False: Reading a program is like reading other kinds of text.

- A. True
- B. False

## 1.12 A Typical First Program

Traditionally, the first program written in a new language is called *Hello, World!* because all it does is display the words, Hello, World! In Python, the source code looks like this.

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

This is an example of using the **print function**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is the phrase:

Hello, World!

Here is the example in activecode. Give it a try!

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

The quotation marks in the program mark the beginning and end of the value. 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.

### Check your understanding

**Checkpoint 1.12.1** The print function:

- A. sends information to the printer to be printed on paper.
- B. displays a value on the screen.

- C. tells the computer to put the information in print, rather than cursive, format.
- D. tells the computer to speak the information.

## 1.13 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments.

A **comment** in a computer program is text that is intended only for the human reader - it is completely ignored by the interpreter. In Python, the `#` token starts a comment. The rest of the line is ignored. Here is a new version of *Hello, World!*.

```
#-----  
# This demo program shows off how elegant Python is!  
# Written by Joe Soap, December 2010.  
# Anyone may freely copy or modify this program.  
#-----  
  
print("Hello,World!")      # Isn't this easy!
```

Notice that when you run this program, it still only prints the phrase Hello, World! None of the comments appear. You'll also notice that we've left a blank line in the program. Blank lines are also ignored by the interpreter, but comments and blank lines can make your programs much easier for humans to parse. Use them liberally!

### Check your understanding

#### Checkpoint 1.13.1 What are comments for?

- A. To tell the computer what you mean in your program.
- B. For the people who are reading your code to know, in natural language, what the program is doing.
- C. Nothing, they are extraneous information that is not needed.
- D. Nothing in a short program. They are only needed for really large programs.

**Note 1.13.2** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 1.14 Glossary

### Glossary

**activecode.** A unique interpreter environment that allows Python to be executed from within a web browser.

**algorithm.** A general step by step process for solving a problem.

**bug.** An error in a program.

**byte code.** An intermediate language between source code and object code. Many modern languages first compile source code into byte code and then interpret the byte code with a program called a *virtual machine*.

**codelens.** An interactive environment that allows the user to control the step by step execution of a Python program

**comment.** Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

**compile.** To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

**debugging.** The process of finding and removing any of the three kinds of programming errors.

**exception.** Another name for a runtime error.

**executable.** Another name for object code that is ready to be executed.

**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.

**high-level language.** A programming language like Python that is designed to be easy for humans to read and write.

**interpret.** To execute a program in a high-level language by translating it one line at a time.

**low-level language.** A programming language that is designed to be easy for a computer to execute; also called machine language or assembly language.

**natural language.** Any one of the languages that people speak that evolved naturally.

**object code.** The output of the compiler after it translates the program.

**parse.** To examine a program and analyze the syntactic structure.

**portability.** A property of a program that can run on more than one kind of computer.

**print function.** A function used in a program or script that causes the Python interpreter to display a value on its output device.

**problem solving.** The process of formulating a problem, finding a solution, and expressing the solution.

**program.** A sequence of instructions that specifies to a computer actions and computations to be performed.

**programming language.** A formal notation for representing solutions.

**Python shell.** An interactive user interface to the Python interpreter. The user of a Python shell types commands at the prompt (`>>>`), and presses the return key to send these commands immediately to the interpreter for processing.

**runtime error.** An error that does not occur until the program has started to execute but that prevents the program from continuing.

**semantic error.** An error in a program that makes it do something other than what the programmer intended.

**semantics.** The meaning of a program.

**shell mode.** A style of using Python where we type expressions at the command prompt, and the results are shown immediately. Contrast with **source code**, and see the entry under **Python shell**.

**source code.** A program, stored in a file, in a high-level language before being compiled or interpreted.

**syntax.** The structure of a program.

**syntax error.** An error in a program that makes it impossible to parse — and therefore impossible to interpret.

**token.** One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

## 1.15 Exercises

1. This exercise will let you practice displaying information. Write a program that prints following:

- Your name.
- Your favorite color.
- Your favorite word, in double quotes.
- Your favorite poem, at least three lines long (If your favorite poem is longer than ten lines, give only the first three to ten lines.)
- The author of the poem. The name should be indented four spaces and preceded by a hyphen and a space. If you don't know who wrote the poem, use "Anonymous" as the author.

Use complete sentences and blank lines for readability. To print a blank line, use:

```
| print()
```

The parentheses are required, even though there is nothing between them. If you don't believe me, leave them out and see what the output looks like!

Here is what the output of a possible solution looks like:

```
My name is Juana Fulano.  
My favorite color is purple.  
My favorite word is "giraffe".
```

```
My favorite poem:
```

```
Greet the dawn,  
Rise with the sun.  
Rejoice in the sunset  
When day is done.  
- Anonymous
```

# Chapter 2

## Simple Python Data

### 2.1 Variables, Expressions and Statements



In order to get started learning any programming language there are a number of concepts and ideas that are necessary. The goal of this chapter is to introduce you to the basic vocabulary of programming and some of the fundamental building blocks of Python.

### 2.2 Values and Data Types

A **value** is one of the fundamental things — like a word or a number — that a program manipulates. The values we have seen so far are 5 (the result when we added `2 + 3`), and "Hello, World!". We often refer to these values as **objects** and we will use the words value and object interchangeably.

**Note 2.2.1** Actually, the 2 and the 3 that are part of the addition above are values(objects) as well.

These objects are classified into different **classes**, or **data types**: 4 is an *integer*, and "Hello, World!" is a *string*, so-called because it contains a string or sequence of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

If you are not sure what class a value falls into, Python has a function called **type** which can tell you.

```
print(type("Hello,World!"))
print(type(17))
print("Hello,World")
```

Not surprisingly, strings belong to the class **str** and integers belong to the class **int**.

**Note 2.2.2** When we show the value of a string using the `print` function, such as in the third example above, the quotes are not present in the output. The value of the string is the sequence of characters inside the quotes. The quotes are only necessary to help Python know what the value is.

You may have used function notation in a math class, like `y = f(x)`, likely only for functions that act on a single numeric value, and produce a single numeric value. Python has no such restrictions: Inputs and outputs may be of any type.

In the Python shell, it is not necessary to use the `print` function to see the values shown above. The shell evaluates the Python function and automatically prints the result. For example, consider the shell session shown below. When we ask the shell to evaluate `type("Hello, World!")`, it responds with the appropriate answer and then goes on to display the prompt for the next use.

```
Python 3.1.2 (r312:79360M, Mar 24 2010, 01:33:18)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> type("Hello, World!")
<class 'str'>
>>> type(17)
<class 'int'>
>>> "Hello, World"
'Hello, World'
>>>
```

Note that in the last example, we simply ask the shell to evaluate the string “Hello, World”. The result is as you might expect, the string itself.

Continuing with our discussion of data types, numbers with a decimal point belong to a class called `float`, because these numbers are represented in a format called *floating-point*. At this stage, you can treat the words *class* and *type* interchangeably. We’ll come back to a deeper understanding of what a class is in later chapters.

```
| print(type(17))
| print(type(3.2))
```

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

```
| print(type("17"))
| print(type("3.2"))
```

They’re strings!

Strings in Python can be enclosed in either single quotes (‘) or double quotes (“ - the double quote character), or three of the same separate quote characters (‘‘‘ or ‘‘‘‘).

```
| print(type('This_is_a_string.' ) )
| print(type("And_so_is_this." ) )
| print(type("'''and_this.'''") )
| print(type(''''and_even_this...'''') )
```

Double quoted strings can contain single quotes inside them, as in “Bruce’s beard”, and single quoted strings can have double quotes inside them, as in ‘The knights who say “Ni!”’. Strings enclosed with three occurrences of either quote symbol are called triple quoted strings. They can contain either single or double quotes:

```
| print(''''Oh no'', she exclaimed, "Ben's bike is broken!" '''')
```

Triple quoted strings can even span multiple lines:

```
| print("""This message will span
several lines
of the text.""")
```

Python doesn't care whether you use single or double quotes or the three-of-a-kind quotes to surround your strings. Once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value.

```
| print('This is a string.')
print("""And so is this."""")
```

So the Python language designers usually chose to surround their strings by single quotes. What do you think would happen if the string already contained single quotes?

When you type a large integer, you might be tempted to use commas between groups of three digits, as in `42,000`. This is not a legal integer in Python, but it does mean something else, which is legal:

```
| print(42000)
print(42,000)
```

Well, that's not what we expected at all! Because of the comma, Python chose to treat this as a *pair* of values. In fact, the `print` function can print any number of values as long as you separate them by commas. Notice that the values are separated by spaces when they are displayed.

```
| print(42, 17, 56, 34, 11, 4.35, 32)
print(3.4, "hello", 45)
```

Remember not to put commas or spaces in your integers, no matter how big they are. Also revisit what we said in the previous chapter: formal languages are strict, the notation is concise, and even the smallest change might mean something quite different from what you intended.

### Check your understanding

**Checkpoint 2.2.3** How can you determine the type of a variable?

- A. Print out the value and determine the data type based on the value printed.
- B. Use the `type` function.
- C. Use it in a known equation and print the result.
- D. Look at the declaration of the variable.

**Checkpoint 2.2.4** What is the data type of 'this is what kind of data'?

- A. Character
- B. Integer
- C. Float
- D. String

## 2.3 Type conversion functions

Sometimes it is necessary to convert values from one type to another. Python provides a few simple functions that will allow us to do that. The functions `int`, `float` and `str` will (attempt to) convert their arguments into types `int`, `float` and `str` respectively. We call these **type conversion** functions.

The `int` function can take a floating point number or a string, and turn it into an `int`. For floating point numbers, it *discards* the decimal portion of the number - a process we call *truncation towards zero* on the number line. Let us see this in action:

```
print(3.14, int(3.14))
print(3.9999, int(3.9999))      # This doesn't round to
                                the closest int!
print(3.0, int(3.0))
print(-3.999, int(-3.999))      # Note that the result is
                                closer to zero

print("2345", int("2345"))      # parse a string to
                                produce an int
print(17, int(17))              # int even works on
                                integers
print(int("23bottles"))
```

The last case shows that a string has to be a syntactically legal number, otherwise you'll get one of those pesky runtime errors. Modify the example by deleting the `bottles` and rerun the program. You should see the integer 23.

The type converter `float` can turn an integer, a float, or a syntactically legal string into a float.

```
print(float("123.45"))
print(type(float("123.45")))
```

The type converter `str` turns its argument into a string. Remember that when we print a string, the quotes are removed. However, if we print the type, we can see that it is definitely `str`.

```
print(str(17))
print(str(123.45))
print(type(str(123.45)))
```

### Check your understanding

**Checkpoint 2.3.1** What value is printed when the following statement executes?

```
| print( int(53.785) )
```

- A. Nothing is printed. It generates a runtime error.
- B. 53
- C. 54
- D. 53.785

## 2.4 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.

**Assignment statements** create new variables and also give them values to refer to.

```
message = "What's up, Doc?"
n = 17
pi = 3.14159
```

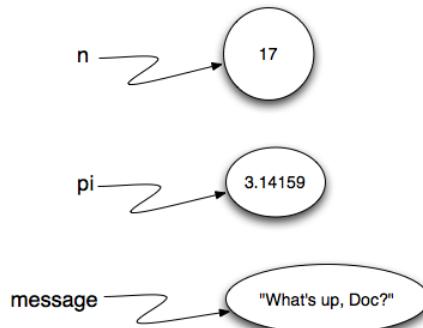
This example makes three assignments. The first assigns the string value "What's up, Doc?" to a new variable named `message`. The second gives the integer 17 to `n`, and the third assigns the floating-point number 3.14159 to a variable called `pi`.

The **assignment token**, `=`, should not be confused with *equality* (we will see later that equality uses the `==` token). The assignment statement links a *name*, on the left hand side of the operator, with a *value*, on the right hand side. This is why you will get an error if you enter:

```
| 17 = n
```

**Note 2.4.1** When reading or writing code, say to yourself “`n` is assigned 17” or “`n` gets the value 17” or “`n` is a reference to the object 17” or “`n` refers to the object 17”. Don’t say “`n` equals 17”.

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, known as a **reference diagram**, is often called a **state snapshot** because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable’s state of mind). This diagram shows the result of executing the assignment statements shown above.



If you ask Python to evaluate a variable, it will produce the value that is currently linked to the variable. In other words, evaluating a variable will give

you the value that is referred to by the variable.

```
message = "What's up, Doc?"
n = 17
pi = 3.14159

print(message)
print(n)
print(pi)
```

In each case the result is the value of the variable. To see this in even more detail, we can run the program using codelens.

```
message = "What's up, Doc?"
n = 17
pi = 3.14159

print(message)
print(n)
print(pi)
```

Now, as you step through the statements, you can see the variables and the values they reference as those references are created.

Variables also have types; again, we can ask the interpreter what they are.

```
message = "What's up, Doc?"
n = 17
pi = 3.14159

print(type(message))
print(type(n))
print(type(pi))
```

The type of a variable is the type of the object it currently refers to.

We use variables in a program to “remember” things, like the current score at the football game. But variables are *variable*. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable.

**Note 2.4.2** This is different from math. In math, if you give  $x$  the value 3, it cannot change to refer to a different value half-way through your calculations!

To see this, read and then run the following program. You’ll notice we change the value of day three times, and on the third assignment we even give it a value that is of a different type.

```
day = "Thursday"
print(day)
day = "Friday"
print(day)
day = 21
print(day)
```

A great deal of programming is about having the computer remember things. For example, we might want to keep track of the number of missed calls on your phone. Each time another call is missed, we will arrange to update or change the variable so that it will always reflect the correct value.

### Check your understanding

**Checkpoint 2.4.3** What is printed when the following statements execute?

```

day = "Thursday"
day = 32.5
day = 19
print(day)

```

- A. Nothing is printed. A runtime error occurs.
- B. Thursday
- C. 32.5
- D. 19

## 2.5 Variable Names and Keywords

**Variable names** can be arbitrarily long. They can contain both letters and digits, but they have to begin with a letter or an underscore. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. `Bruce` and `bruce` are different variables.

**Warning 2.5.1** Variable names can never contain spaces.

The underscore character (`_`) can also appear in a name. It is often used in names with multiple words, such as `my_name` or `price_of_tea_in_china`. There are some situations in which names beginning with an underscore have special meaning, so a safe rule for beginners is to start all names with a letter.

If you give a variable an illegal name, you get a syntax error. In the example below, each of the variable names is illegal.

```

76trombones = "big parade"
more$ = 1000000
class = "Computer Science 101"

```

`76trombones` is illegal because it does not begin with a letter. `more$` is illegal because it contains an illegal character, the dollar sign. But what's wrong with `class`?

It turns out that `class` is one of the Python **keywords**. Keywords define the language's syntax rules and structure, and they cannot be used as variable names. Python has thirty-something keywords (and every now and again improvements to Python introduce or eliminate one or two):

**Table 2.5.2**

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

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.

Programmers generally choose names for their variables that are meaningful to the human readers of the program — they help the programmer document, or remember, what the variable is used for.

**Warning 2.5.3** Beginners sometimes confuse “meaningful to the human readers” with “meaningful to the computer”. So they'll wrongly think that because they've called some variable `average` or `pi`, it will somehow automagically cal-

culate an average, or automatically associate the variable `pi` with the value 3.14159. No! The computer doesn't attach semantic meaning to your variable names.

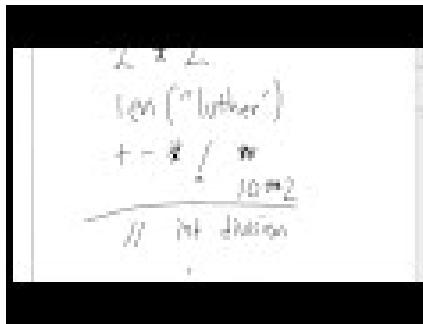
So you'll find some instructors who deliberately don't choose meaningful names when they teach beginners — not because they don't think it is a good habit, but because they're trying to reinforce the message that you, the programmer, have to write some program code to calculate the average, or you must write an assignment statement to give a variable the value you want it to have.

### Check your understanding

**Checkpoint 2.5.4** True or False: the following is a legal variable name in Python: `A_good_grade_is_A+`

- A. True
- B. False

## 2.6 Statements and Expressions



A **statement** is an instruction that the Python interpreter can execute. We have only seen the assignment statement so far. Some other kinds of statements that we'll see shortly are `while` statements, `for` statements, `if` statements, and `import` statements. (There are other kinds too!)

An **expression** is a combination of values, variables, operators, and calls to functions. Expressions need to be evaluated. If you ask Python to `print` an expression, the interpreter **evaluates** the expression and displays the result.

```
print(1 + 1)
print(len("hello"))
```

In this example `len` is a built-in Python function that returns the number of characters in a string. We've previously seen the `print` and the `type` functions, so this is our third example of a function!

The *evaluation of an expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable. Evaluating a variable gives the value that the variable refers to.

```
y = 3.14
x = len("hello")
print(x)
print(y)
```

If we take a look at this same example in the Python shell, we will see one of the distinct differences between statements and expressions.

```

>>> y = 3.14
>>> x = len("hello")
>>> print(x)
5
>>> print(y)
3.14
>>> y
3.14
>>>

```

Note that when we enter the assignment statement, `y = 3.14`, only the prompt is returned. There is no value. This is due to the fact that statements, such as the assignment statement, do not return a value. They are simply executed.

On the other hand, the result of executing the assignment statement is the creation of a reference from a variable, `y`, to a value, `3.14`. When we execute the `print` function working on `y`, we see the value that `y` is referring to. In fact, evaluating `y` by itself results in the same response.

## 2.7 Operators and Operands

**Operators** are special tokens that represent computations like addition, multiplication and division. The values the operator works on are called **operands**.

The following are all legal Python expressions whose meaning is more or less clear:

```

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

```

The tokens `+`, `-`, and `*`, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (`*`) is the token for multiplication, and `**` is the token for exponentiation. Addition, subtraction, multiplication, and exponentiation all do what you expect.

```

print(2 + 3)
print(2 - 3)
print(2 * 3)
print(2 ** 3)
print(3 ** 2)

```

When a variable name appears in the place of an operand, it is replaced with the value that it refers to before the operation is performed. For example, what if we wanted to convert 645 minutes into hours. In Python 3, division is denoted by the operator token `/` which always evaluates to a floating point result.

```

minutes = 645
hours = minutes / 60
print(hours)

```

What if, on the other hand, we had wanted to know how many *whole* hours there are and how many minutes remain. To help answer this question, Python gives us a second flavor of the division operator. This version, called **integer division**, uses the token `//`. It always *truncates* its result down to the next smallest integer (to the left on the number line).

```

print(7 / 4)
print(7 // 4)

minutes = 645
hours = minutes // 60
print(hours)

print(6//4)
print(-6//4)

```

Pay particular attention to the first two examples above. Notice that the result of floating point division is 1.75 but the result of the integer division is simply 1. Take care that you choose the correct flavor of the division operator. If you're working with expressions where you need floating point values, use the division operator `/`. If you want an integer result, use `//`.

The **modulus operator**, sometimes also called the **remainder operator** or **integer remainder operator** works on integers (and integer expressions) 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      # This is the integer division
operator
print(quotient)
remainder = 7 % 3
print(remainder)

```

In the above example, 7 divided by 3 is 2 when we use integer division and there is a remainder of 1 when we use the modulus operator.

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.

Finally, returning to our time example, the remainder operator is extremely useful for doing conversions, say from seconds, to hours, minutes and seconds. If we start with a number of seconds, say 7684, the following program uses integer division and remainder to convert to an easier form. Step through it to be sure you understand how the division and remainder operators are being used to compute the correct values.

```

total_secs = 7684
hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes = secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining % 60

```

### Check your understanding

**Checkpoint 2.7.1** What value is printed when the following statement executes?

```
| print(18 / 4)
```

- A. 4.5
- B. 5
- C. 4
- D. 2

**Checkpoint 2.7.2** What value is printed when the following statement executes?

```
| print(18 // 4)
```

- A. 4.25
- B. 5
- C. 4
- D. 2

**Checkpoint 2.7.3** What value is printed when the following statement executes?

```
| print(18 % 4)
```

- A. 4.25
- B. 5
- C. 4
- D. 2

## 2.8 Input



The program in the previous section works fine but is very limited in that it only works with one value for `total_secs`. What if we wanted to rewrite the program so that it was more general. One thing we could do is allow the user to enter any value they wish for the number of seconds. The program could then print the proper result for that starting value.

In order to do this, we need a way to get `input` from the user. Luckily, in Python there is a built-in function to accomplish this task. As you might expect, it is called `input`.

```
| n = input("Please enter your name: ")
```

The `input` function allows the user to provide a **prompt string**. When the function is evaluated, the prompt is shown. The user of the program can enter the name and press return. When this happens the text that has been entered is returned from the `input` function, and in this case assigned to the variable `n`. Make sure you run this example a number of times and try some different names in the input box that appears.

```
| n = input("Please enter your name: ")
print("Hello", n)
```

It is very important to note that the `input` function returns a string value. Even if you asked the user to enter their age, you would get back a string like "17". It would be your job, as the programmer, to convert that string into an int or a float, using the `int` or `float` converter functions we saw earlier.

To modify our previous program, we will add an input statement to allow the user to enter the number of seconds. Then we will convert that string to an integer. From there the process is the same as before. To complete the example, we will print some appropriate output.

```
str_seconds = input("Please enter the number of seconds you wish to convert")
total_secs = int(str_seconds)

hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes = secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining % 60

print("Hrs=", hours, "mins=", minutes, "secs=", secs_finally_remaining)
```

The variable `str_seconds` will refer to the string that is entered by the user. As we said above, even though this string may be 7684, it is still a string and not a number. To convert it to an integer, we use the `int` function. The result is referred to by `total_secs`. Now, each time you run the program, you can enter a new value for the number of seconds to be converted.

#### Check your understanding

**Checkpoint 2.8.1** What is printed when the following statements execute?

```
n = input("Please enter your age:")
# user types in 18
print(type(n))
```

- A. <class 'str'>
- B. <class 'int'>
- C. <class 18>
- D. 18

**Checkpoint 2.8.2** Click on all of the variables of type `int` in the code below

```
seconds = input("Please enter the number of seconds you wish to convert")

hours = int(seconds) // 3600

total_secs = int(seconds)

secs_still_remaining = total_secs % 3600
print(secs_still_remaining)
```

**Checkpoint 2.8.3** Click on all of the variables of type `str` in the code below

```

seconds = input("Please enter the number of seconds
you wish to convert")

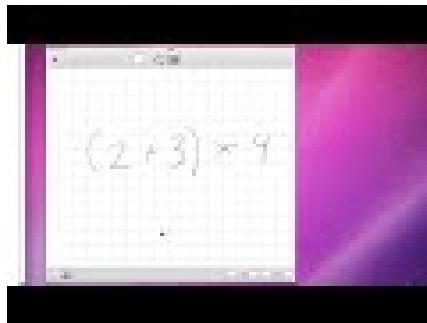
hours = int(seconds) // 3600

total_secs = int(seconds)

secs_still_remaining = total_secs % 3600
print(secs_still_remaining)

```

## 2.9 Order of Operations



When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does.

- 1 Parentheses 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 though it doesn't change the result.
- 2 Exponentiation has the next highest precedence, so  $2**1+1$  is 3 and not 4, and  $3*1**3$  is 3 and not 27. Can you explain why?
- 3 Multiplication and both division operators have the same precedence, which is higher than addition and subtraction, which also have the same precedence. So  $2*3-1$  yields 5 rather than 4, and  $5-2*2$  is 1, not 6.
- 4 Operators with the *same* precedence (except for  $**$ ) are evaluated from left-to-right. In algebra we say they are *left-associative*. So in the expression  $6-3+2$ , the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been  $6-(3+2)$ , which is 1.

**Note 2.9.1** An exception to the left-to-right left-associative rule is the exponentiation operator  $**$ . A useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:

```

print(2 ** 3 ** 2)      # the right-most ** operator gets
                        # done first!
print((2 ** 3) ** 2)    # use parentheses to force the order
                        # you want!

```

See [Section 21.11](#) for *all* the operators introduced in this book. You will also see many upcoming non-mathematical Python operators.

### Check your understanding

**Checkpoint 2.9.2** What is the value of the following expression:

```
| 16 - 2 * 5 // 3 + 1
```

- A. 14
- B. 24
- C. 3
- D. 13.667

**Checkpoint 2.9.3** What is the value of the following expression:

```
| 2 ** 2 ** 3 * 3
```

- A. 768
- B. 128
- C. 12
- D. 256

Here are animations for the above expressions:

**Checkpoint 2.9.4** An interactive Runestone problem goes here, but there is not yet a static representation.

**Checkpoint 2.9.5** An interactive Runestone problem goes here, but there is not yet a static representation.

## 2.10 Reassignment

### 2.10.1 Introduction

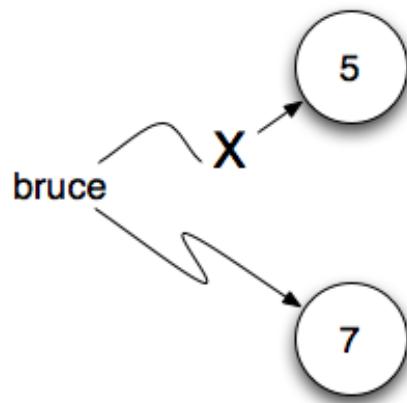


As we have mentioned previously, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
bruce = 5
print(bruce)
bruce = 7
print(bruce)
```

The first time `bruce` is printed, its value is 5, and the second time, its value is 7. The assignment statement changes the value (the object) that `bruce` refers to.

Here is what **reassignment** looks like in a reference diagram:



It is important to note that in mathematics, a statement of equality is always true. If  $a$  is equal to  $b$  now, then  $a$  will always equal to  $b$ . In Python, an assignment statement can make two variables refer to the same object and therefore have the same value. They appear to be equal. However, because of the possibility of reassignment, they don't have to stay that way:

```
a = 5
b = a      # after executing this line, a and b are now equal
print(a, b)
a = 3      # after executing this line, a and b are no longer
           equal
print(a, b)
```

Line 4 changes the value of  $a$  but does not change the value of  $b$ , so they are no longer equal. We will have much more to say about equality in a later chapter.

### 2.10.2 Developing your mental model of How Python Evaluates

It's important to start to develop a good mental model of the steps Python takes when evaluating an assignment statement. In an assignment statement Python first evaluates the code on the right hand side of the assignment operator. It then gives a name to whatever that is. The (very short) visualization below shows what is happening.

**Checkpoint 2.10.1** An interactive Runestone problem goes here, but there is not yet a static representation.

In the first statement  $a = 5$  the literal number 5 evaluates to 5, and is given the name  $a$ . In the second statement, the variable  $a$  evaluates to 5 and so 5 now ends up with a second name  $b$ .

**Note 2.10.2** In some programming languages, a different symbol is used for assignment, such as  $<-$  or  $:=$ . The intent is that this will help to avoid confusion. Python chose to use the tokens  $=$  for assignment, and  $==$  for equality. This is a popular choice also found in languages like C, C++, Java, and C#.

#### Check your understanding

**Checkpoint 2.10.3** After the following statements, what are the values of  $x$  and  $y$ ?

```
| x = 15
| y = x
| x = 22
```

- A. x is 15 and y is 15
- B. x is 22 and y is 22
- C. x is 15 and y is 22
- D. x is 22 and y is 15

## 2.11 Updating Variables



One of the most common forms of reassignment is an **update** where the new value of the variable depends on the old. For example,

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

This means get the current value of x, add one, and then update x with the new value. The new value of x is the old value of x plus 1. Although this assignment statement may look a bit strange, remember that executing assignment is a two-step process. First, evaluate the right-hand side expression. Second, let the variable name on the left-hand side refer to this new resulting object. The fact that x appears on both sides does not matter. The semantics of the assignment statement makes sure that there is no confusion as to the result. The visualizer makes this very clear.

**Checkpoint 2.11.1** An interactive Runestone problem goes here, but there is not yet a static representation.

```
x = 6          # initialize x
print(x)
x = x + 1      # update x
print(x)
```

If you try to update a variable that doesn't exist, you get an error because Python evaluates the expression on the right side of the assignment operator before it assigns the resulting value to the name on the left. Before you can update a variable, you have to **initialize** it, usually with a simple assignment. In the above example, x was initialized to 6.

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**. Sometimes programmers also talk about **bumping** a variable, which means the same as incrementing it by 1.

### Note 2.11.2 Advanced Topics.

- [Topic 1:](#)<sup>1</sup> Python Beyond the Browser. This is a gentle introduction to

using Python from the command line. We'll cover this later, but if you are curious about what Python looks like outside of this eBook, you can have a look here. There are also instructions for installing Python on your computer here.

- **Topic 2:**<sup>2</sup> Dive Into Python 3, this is an online textbook by Mark Pilgrim. If you have already had some programming experience, this book takes you off the deep end with both feet.

### Check your understanding

**Checkpoint 2.11.3** What is printed when the following statements execute?

```
x = 12  
x = x - 1  
print(x)
```

- A. 12
- B. -1
- C. 11
- D. Nothing. An error occurs because x can never be equal to x - 1.

**Checkpoint 2.11.4** What is printed when the following statements execute?

```
x = 12  
x = x - 3  
x = x + 5  
x = x + 1  
print(x)
```

- A. 12
- B. 9
- C. 15
- D. Nothing. An error occurs because x cannot be used that many times in assignment statements.

**Checkpoint 2.11.5** Construct the code that will result in the value 134 being printed.

- mybankbalance=100  
mybankbalance=mybankbalance+34  
print(mybankbalance)

**Note 2.11.6** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

---

<sup>1</sup>at\_1\_1.html

<sup>2</sup><http://www.diveintopython3.net/>

## 2.12 Glossary

### Glossary

**assignment statement.** A statement that assigns a value to a name (variable). To the left of the assignment operator, `=`, is a name. To the right of the assignment token is an expression which is evaluated by the Python interpreter and then assigned to the name. The difference between the left and right hand sides of the assignment statement is often confusing to new programmers. In the following assignment:

```
| n = n + 1
```

`n` plays a very different role on each side of the `=`. On the right it is a *value* and makes up part of the *expression* which will be evaluated by the Python interpreter before assigning it to the name on the left.

**assignment token.** `=` is Python's assignment token, which should not be confused with the mathematical comparison operator using the same symbol.

**class.** see **data type** below

**comment.** Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

**data type.** A set of values. The type of a value determines how it can be used in expressions. So far, the types you have seen are integers (`int`), floating-point numbers (`float`), and strings (`str`).

**decrement.** Decrease by 1.

**evaluate.** To simplify an expression by performing the operations in order to yield a single value.

**expression.** A combination of operators and operands (variables and values) that represents a single result value. Expressions are evaluated to give that result.

**float.** A Python data type which stores *floating-point* numbers. Floating-point numbers are stored internally in two parts: a *base* and an *exponent*. When printed in the standard format, they look like decimal numbers. Beware of rounding errors when you use `floats`, and remember that they are only approximate values.

**increment.** Both as a noun and as a verb, increment means to increase by 1.

**initialization (of a variable).** To initialize a variable is to give it an initial value. Since in Python variables don't exist until they are assigned values, they are initialized when they are created. In other programming languages this is not the case, and variables can be created without being initialized, in which case they have either default or *garbage* values.

**int.** A Python data type that holds positive and negative **whole** numbers.

**integer division.** An operation that divides one integer by another and yields an integer. Integer division yields only the whole number of times that the numerator is divisible by the denominator and discards any remainder.

**keyword.** A reserved word that is used by the compiler to parse program; you cannot use keywords like `if`, `def`, and `while` as variable names.

**modulus operator.** Also called remainder operator or integer remainder operator. Gives the remainder after performing integer division.

**object.** Also known as a data object (or data value). The fundamental things that programs are designed to manipulate (or that programmers ask to do things for them).

**operand.** One of the values on which an operator operates.

**operator.** A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

**prompt string.** Used during interactive input to provide the user with hints as to what type of value to enter.

**reference diagram.** A picture showing a variable with an arrow pointing to the value (object) that the variable refers to. See also **state snapshot**.

**rules of precedence.** The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

**state snapshot.** A graphical representation of a set of variables and the values to which they refer, taken at a particular instant during the program's execution.

**statement.** An instruction that the Python interpreter can execute. So far we have only seen the assignment statement, but we will soon meet the `import` statement and the `for` statement.

**str.** A Python data type that holds a string of characters.

**type conversion function.** A function that can convert a data value from one type to another.

**value.** A number or string (or other things to be named later) that can be stored in a variable or computed in an expression.

**variable.** A name that refers to a value.

**variable name.** A name given to a variable. Variable names in Python consist of a sequence of letters (a..z, A..Z, and \_) and digits (0..9) that begins with a letter. In best programming practice, variable names should be chosen so that they describe their use in the program, making the program *self documenting*.

## 2.13 Exercises

- Evaluate the following numerical expressions in your head, then use the active code window to check your results:

1 5 \*\* 2

2 9 \* 5

3 15 / 12

4 12 / 15

5 15 // 12

6 12 // 15

7 5 % 2

8 9 % 5

9 15 % 12

10 12 % 15

11 6 % 6

```
12 0 % 7
```

```
|   print(5 ** 2)
```

2. What is the order of the arithmetic operations in the following expression. Evaluate the expression by hand and then check your work.

```
|   2 + (3 - 1) * 10 / 5 * (2 + 3)
```

3. Many people keep time using a 24 hour clock (11 is 11am and 23 is 11pm, 0 is midnight). If it is currently 13 and you set your alarm to go off in 50 hours, it will be 15 (3pm). Write a Python program to solve the general version of the above problem. Ask the user for the time now (in hours), and then ask for the number of hours to wait for the alarm. Your program should output what the time will be on the clock when the alarm goes off.
4. It is possible to name the days 0 through 6 where day 0 is Sunday and day 6 is Saturday. If you go on a wonderful holiday leaving on day number 3 (a Wednesday) and you return home after 10 nights you would return home on a Saturday (day 6) Write a general version of the program which asks for the starting day number, and the length of your stay, and it will tell you the number of day of the week you will return on.

```
| # Problem 4
```

```
| # My Name:
```

5. Take the sentence: *All work and no play makes Jack a dull boy.* Store each word in a separate variable, then print out the sentence on one line using `print`.
6. Add parenthesis to the expression  $6 * 1 - 2$  to change its value from 4 to -6.
7. The formula for computing the final amount if one is earning compound interest is given on Wikipedia as

$$A = P \left(1 + \frac{r}{n}\right)^{nt}$$

Where,

- $P$  = principal amount (initial investment)
- $r$  = annual nominal interest rate (as a decimal)
- $n$  = number of times the interest is compounded per year
- $t$  = number of years

Write a Python program that assigns the principal amount of 10000 to variable  $P$ , assign to  $n$  the value 12, and assign to  $r$  the interest rate of 8% (0.08). Then have the program prompt the user for the number of years,  $t$ , that the money will be compounded for. Calculate and print the final amount after  $t$  years.

8. Write a program that will compute the area of a circle. Prompt the user to enter the radius and print a nice message back to the user with the answer.
9. Write a program that will compute the area of a rectangle. Prompt the user to enter the width and height of the rectangle. Print a nice message with the answer.
10. Write a program that will compute MPG for a car. Prompt the user to enter the number of miles driven and the number of gallons used. Print a nice message with the answer.
11. Write a program that will convert degrees celsius to degrees fahrenheit.
12. Write a program that will convert degrees fahrenheit to degrees celsius.

# Chapter 3

## Debugging Interlude 1

### 3.1 How to be a Successful Programmer

One of the most important skills you need to acquire to complete this book successfully is the ability to debug your programs. Debugging might be the most under-appreciated, and under-taught, skill in introductory computer science. For that reason we are introducing a series of “debugging interludes.” Debugging is a skill that you need to master over time, and some of the tips and tricks are specific to different aspects of Python programming. So look for additional debugging interludes throughout the rest of this book.

Programming is an odd thing in a way. Here is why. As programmers we spend 99% of our time trying to get our program to work. We struggle, we stress, we spend hours deep in frustration trying to get our program to execute correctly. Then when we do get it going we celebrate, hand it in, and move on to the next homework assignment or programming task. But here is the secret, when you are successful, you are happy, your brain releases a bit of chemical that makes you feel good. You need to organize your programming so that you have lots of little successes. It turns out your brain doesn’t care all that much if you have successfully written hello world, or a fast fourier transform (trust me its hard) you still get that little release that makes you happy. When you are happy you want to go on and solve the next little problem. Essentially I’m telling you once again, start small, get something small working, and then add to it.

### 3.2 How to Avoid Debugging

Perhaps the most important lesson in debugging is that it is **largely avoidable** – if you work carefully.

**1 Understand the Problem** You must have a firm grasp on **what** you are trying to accomplish but not necessarily **how** to do it. You do not need to understand the entire problem. But you must understand at least a portion of it and what the program should do in a specific circumstance – what output should be produced for some given input. This will allow you to test your progress. You can then identify if a solution is correct or whether there remains work to do or bugs to fix. This is probably the single biggest piece of advice for programmers at every level.

**2 Start Small** It is tempting to sit down and crank out an entire program at once. But, when the program – inevitably – does not work, you have

a myriad of options for things that might be wrong. Where to start? Where to look first? How to figure out what went wrong? I'll get to that in the next section. So, start with something really small. Maybe just two lines and then make sure that runs. Hitting the run button is quick and easy. It gives you immediate feedback about whether what you have just done works or not. Another immediate benefit of having something small working is that you have something to turn in. Turning in a small, incomplete program, is almost always better than nothing.

**3 Keep Improving It** Once you have a small part of your program working, the next step is to figure out something small to add to it – how can you move closer to a correct solution. As you add to your program, you gain greater insight into the underlying problem you are trying to solve. If you keep adding small pieces of the program one at a time, it is much easier to figure out what went wrong. (This of course means you must be able to recognize if there is an error. And that is done through testing.) As long as you always test each new bit of code, it is most likely that any error is in the new code you have just added. Less new code means its easier to figure out where the problem is.

This notion of **Get something working and keep improving it** is a mantra that you can repeat throughout your career as a programmer. It's a great way to avoid the frustrations mentioned above. Think of it this way. Every time you have a little success, your brain releases a tiny bit of chemical that makes you happy. So, you can keep yourself happy and make programming more enjoyable by creating lots of small victories for yourself.

**Note 3.2.1** The technique of start small and keep improving is the basis of **Agile** software development. This practice is used widely in the industry.

Ok, let's look at an example. Let's solve the problem posed in question 3 at the end of the Simple Python Data chapter. Ask the user for the time now (in hours 0 - 23), and ask for the number of hours to wait. Your program should output what the time will be on the clock when the alarm goes off. For example, if `current_time` is 8 and `wait_time` is 5, `final_time` should be 13 (1 pm).

So, where to start? The problem requires two pieces of input from the user, so let's start there and make sure we can get the data we need.

```
current_time = input("what is the current time (in hours)?")
wait_time = input("How many hours do you want to wait")

print(current_time)
print(wait_time)
```

So far so good. Now let's take the next step. We need to figure out what the time will be after waiting `wait_time` number of hours. A reasonable solution is to simply add `wait_time` to `current_time` and print out the result. So lets try that.

```
current_time = input("What is the current time (in hours 0 - 23)?")
wait_time = input("How many hours do you want to wait")

print(current_time)
print(wait_time)

final_time = current_time + wait_time
```

```
|   print(final_time)
```

Hmm, when you run this example you see that something unexpected has happened. You would not realize this was an error unless you first knew what the program was supposed to do.

**Checkpoint 3.2.2** Which of the following best describes what is wrong with the previous example?

- A. Python is stupid and does not know how to add properly.
- B. There is nothing wrong here.
- C. Python is doing string concatenation, not integer addition.

This error was probably pretty simple to spot, because we printed out the value of `final_time` and it is easy to see that the numbers were just concatenated together rather than added.

So what do we do about the problem? We will need to convert both `current_time` and `wait_time` to `int`. At this stage of your programming development, it can be a good idea to include the type of the variable in the variable name itself. So let's look at another iteration of the program that does that, and the conversion to integer.

```
current_time_str = input("What is the current time (in
    hours 0-23)?")
wait_time_str = input("How many hours do you want to wait")

current_time_int = int(current_time_str)
wait_time_int = int(wait_time_str)

final_time_int = current_time_int + wait_time_int
print(final_time_int)
```

Now, that's a lot better, and in fact depending on the hours you chose, it may be exactly right. If you entered 8 for `current_time` and 5 for `wait_time` then 13 is correct. But if you entered 17 (5 pm) for `current_time` and 9 for `wait_time` then the result of 26 is not correct.

This illustrates an important aspect of **testing**: it is important to test your code on a range of inputs. It is especially important to test your code on **boundary conditions**. For this particular problem, you should test your program with `current_time` of 0, 23, and some values in between. You should test your `wait_time` for 0, and some larger values. What about negative numbers? Negative numbers don't make sense, and since we don't really have the tools to deal with telling the user when something is wrong we will not worry about that just yet.

So to account for those numbers that are bigger than 23, we need one final step: using the modulus operator.

```
current_time_str = input("What is the current time (in
    hours 0-23)?")
wait_time_str = input("How many hours do you want to wait")

current_time_int = int(current_time_str)
wait_time_int = int(wait_time_str)

final_time_int = current_time_int + wait_time_int

final_answer = final_time_int % 24
```

```
|   print("The time after waiting is:", final_answer)
```

Of course even in this simple progression, there are other ways you could have gone astray. We'll look at some of those and how you track them down in the next section.

### 3.3 Beginning tips for Debugging

Debugging a program is a different way of thinking than writing a program. The process of debugging is much more like being a detective. Here are a few rules to get you thinking about debugging.

- 1 Everyone is a suspect (Except Python)! It's common for beginner programmers to blame Python, but that should be your last resort. Remember that Python has been used to solve CS1 level problems millions of times by millions of other programmers. So, Python is probably not the problem.
- 2 Find clues. This is the biggest job of the detective and right now there are two important kinds of clues for you to understand.
  - Error Messages
  - Print Statements

### 3.4 Know Your Error Messages

#### 3.4.1 Introduction

Many problems in your program will lead to an error message. For example as I was writing and testing this chapter of the book I wrote the following version of the example program in the previous section.

```
current_time_str = input("What is the current time (in hours 0-23)?")
wait_time_str = input("How many hours do you want to wait")

current_time_int = int(current_time_str)
wait_time_int = int(wait_time_int)

final_time_int = current_time_int + wait_time_int
print(final_time_int)
```

Can you see what is wrong, just by looking at the code? Maybe, maybe not. Our brain tends to see what we think is there, so sometimes it is very hard to find the problem just by looking at the code. Especially when it is our own code and we are sure that we have done everything right!

Let's try the program again, but this time in an activecode:

```
current_time_str = input("What is the current time (in hours 0-23)?")
wait_time_str = input("How many hours do you want to wait")

current_time_int = int(current_time_str)
wait_time_int = int(wait_time_int)

final_time_int = current_time_int + wait_time_int
print(final_time_int)
```

Aha! Now we have an error message that might be useful. The name error tells us that `wait_time_int` is not defined. It also tells us that the error is on line 5. That's **really** useful information. Now look at line five and you will see that `wait_time_int` is used on both the left and the right hand side of the assignment statement.

**Note 3.4.1** The error descriptions you see in activecode may be different (and more understandable!) than in a regular Python interpreter. The interpreter in activecode is limited in many ways, but it is intended for beginners, including the wording chosen to describe errors.

**Checkpoint 3.4.2** Which of the following explains why `wait_time_int = int(wait_time_int)` is an error?

- A. You cannot use a variable on both the left and right hand sides of an assignment statement.
- B. `wait_time_int` does not have a value so it cannot be used on the right hand side.
- C. This is not really an error, Python is broken.

In writing and using this book over the last few years we have collected a lot of statistics about the programs in this book. Here are some statistics about error messages for the exercise we have been looking at.

**Table 3.4.3**

Message	Number	Percent
ParseError:	4999	54.74%
TypeError:	1305	14.29%
NameError:	1009	11.05%
ValueError:	893	9.78%
URLError:	334	3.66%
TokenError:	244	2.67%
SyntaxError:	227	2.49%
TimeLimitError:	44	0.48%
IndentationError:	28	0.31%
AttributeError:	27	0.30%
ImportError:	16	0.18%
IndexError:	6	0.07%

Nearly 90% of the error messages encountered for this problem are ParseError, TypeError, NameError, or ValueError. We will look at these errors in three stages:

- First we will define what these four error messages mean.
- Then, we will look at some examples that cause these errors to occur.
- Finally we will look at ways to help uncover the root cause of these messages.

### 3.4.2 ParseError

Parse errors happen when you make an error in the syntax of your program. Syntax errors are like making grammatical errors in writing. If you don't use periods and commas in your writing then you are making it hard for other readers to figure out what you are trying to say. Similarly Python has certain

grammatical rules that must be followed or else Python can't figure out what you are trying to say.

Usually ParseErrors can be traced back to missing punctuation characters, such as parentheses, quotation marks, or commas. Remember that in Python commas are used to separate parameters to functions. Parentheses must be balanced, or else Python thinks that you are trying to include everything that follows as a parameter to some function.

Here are a couple examples of Parse errors in the example program we have been using. See if you can figure out what caused them.

#### Checkpoint 3.4.4

**Finding Clues** How can you help yourself find these problems? One trick that can be very valuable in this situation is to simply start by commenting out the line number that is flagged as having the error. If you comment out line four, the error message now changes to point to line 5. Now you ask yourself, am I really that bad that I have two lines in a row that have errors on them? Maybe, so taken to the extreme, you could comment out all of the remaining lines in the program. Now the error message changes to `TokenError: EOF in multi-line statement` This is a very technical way of saying that Python got to the end of file (EOF) while it was still looking for something. In this case a right parenthesis.

#### Checkpoint 3.4.5

**Finding Clues** If you follow the same advice as for the last problem, comment out line one, you will immediately get a different error message. Here's where you need to be very careful and not panic. The error message you get now is: `NameError: name 'current_time_str' is not defined on line .` You might be very tempted to think that this is somehow related to the earlier problem and immediately conclude that there is something wrong with the variable name `current_time_str` but if you reflect for a minute you will see that by commenting out line one you have caused a new and unrelated error. That is you have commented out the creation of the name `current_time_str`. So of course when you want to convert it to an `int` you will get the `NameError`. Yes, this can be confusing, but it will become much easier with experience. It's also important to keep calm, and evaluate each new clue carefully so you don't waste time chasing problems that are not really there.

Uncomment line 1 and you are back to the ParseError. Another track is to eliminate a possible source of error. Rather than commenting out the entire line you might just try to assign `current_time_str` to a constant value. For example you might make line one look like this: `current_time_str = "10"` `#input("What is the "current time" (in hours 0-23)?")`. Now you have assigned `current_time_str` to the string 10, and commented out the input statement. And now the program works! So you conclude that the problem must have something to do with the input function.

### 3.4.3 TypeError

TypeErrors occur when you try to combine two objects that are not compatible. For example you try to add together an integer and a string. Usually type errors can be isolated to lines that are using mathematical operators, and usually the line number given by the error message is an accurate indication of the line.

Here's an example of a type error created by a Polish learner. See if you can find and fix the error.

```

a = input('wpisz godzine')
x = input('wpisz liczbe godzin')
int(x)
int(a)
h = x // 24
s = x % 24
print(h, s)
a = a + s
print('godzina teraz', a)

```

**Note 3.4.6 Solution.** In finding this error there are few lessons to think about. First, you may find it very disconcerting that you cannot understand the whole program. Unless you speak Polish then this won't be an issue. But, learning what you can ignore, and what you need to focus on is a very important part of the debugging process. Second, types and good variable names are important and can be very helpful. In this case `a` and `x` are not particularly helpful names, and in particular they do not help you think about the types of your variables, which as the error message implies is the root of the problem here. The rest of the lessons we will get back to in a minute.

The error message provided to you gives you a pretty big hint. `TypeError: unsupported operand type(s) for FloorDiv: 'str' and 'number'` on line: 5 On line five we are trying to use integer division on `x` and 24. The error message tells you that you are trying to divide a string by a number. In this case you know that 24 is a number so `x` must be a string. But how? You can see the function call on line 3 where you are converting `x` to an integer. `int(x)` or so you think. This is lesson three and is one of the most common errors we see in introductory programming. What is the difference between `int(x)` and `x = int(x)`

- The expression `int(x)` converts the string referenced by `x` to an integer but it does not store it anywhere. It is very common to assume that `int(x)` somehow changes `x` itself, as that is what you are intending! The thing that makes this very tricky is that `int(x)` is a valid expression, so it doesn't cause any kind of error, but rather the error happens later on in the program.
- The assignment statement `x = int(x)` is very different. Again, the `int(x)` expression converts the string referenced by `x` to an integer, but this time it also changes what `x` references so that `x` now refers to the integer value returned by the `int` function.

So, the solution to this problem is to change lines 3 and 4 so they are assignment statements.

**Finding Clues** One thing that can help you in this situation is to print out the values and the types of the variables involved in the statement that is causing the error. You might try adding a `print` statement after line 4 `print(x, type(x))` You will see that at least we have confirmed that `x` is of type string. Now you need to start to work backward through the program. You need to ask yourself, where is `x` used in the program? `x` is used on lines 2, 3, and of course 5 and 6 (where we are getting an error). So maybe you move the `print` statement to be after line 2 and again after 3. Line 3 is where you expect the value of `x` to be changed to an integer. Could line 4 be mysteriously changing `x` back to a string? Not very likely. So the value and type of `x` is just what you would expect it to be after line 2, but not after line 3. This helps you isolate the problem to line 3. In fact if you employ one of our earlier techniques of commenting out line 3 you will see that this has no impact on the error, and

is a big clue that line 3 as it is currently written is useless.

### 3.4.4 NameError

Name errors almost always mean that you have used a variable before it has a value. Often NameErrors are simply caused by typos in your code. They can be hard to spot if you don't have a good eye for catching spelling mistakes. Other times you may simply mis-remember the name of a variable or even a function you want to call. You have seen one example of a NameError at the beginning of this section. Here is another one. See if you can get this program to run successfully:

```

str_time = input("What time is it now?")
str_wait_time = input("What is the number of hours to
    wait?")
time = int(str_time)
wai_time = int(str_wait_time)

time_when_alarm_go_off = time + wait_time
print(time_when_alarm_go_off)

```

1.

**Note 3.4.7 Solution.** In this example, the student seems to be a fairly bad speller, as there are a number of typos to fix. The first one is identified as `wait_time` is not defined on line 6. Now in this example you can see that there is `str_wait_time` on line 2, and `wai_time` on line 4 and `wait_time` on line 6. If you do not have very sharp eyes its easy to miss that there is a typo on line 4.

**Finding Clues** With name errors one of the best things you can do is use the editor, or browser search function. Quite often if you search for the exact word in the error message one of two things will happen:

- 1 The word you are searching for will appear only once in your code, it's also likely that it will be on the right hand side of an assignment statement, or as a parameter to a function. That should confirm for you that you have a typo somewhere. If the name in question is what you thought it should be then you probably have a typo on the left hand side of an assignment statement on a line before your error message occurs. Start looking backward at your assignment statements. In some cases it's really nice to leave all the highlighted strings from the search function visible as they will help you very quickly find a line where you might have expected your variable to be highlighted.
- 2 The second thing that may happen is that you will be looking directly at a line where you expected the search to find the string in question, but it will not be highlighted. Most often that will be the typo right there.

Here is another one for you to try:

```

n = input("What time is it now (in hours)?")
n = imt(n)
m = input("How many hours do you want to wait?")
m = int(m)
q = m % 12
print("The time is now", q)

```

2.

**Note 3.4.8 Solution.** This one is once again a typo, but the typo is not in a variable name, but rather, the name of a function. The search strategy would

help you with this one easily, but there is another clue for you as well. The editor in the textbook, as well as almost all Python editors in the world provide you with color clues. Notice that on line 2 the function `int` is not highlighted blue like the word `int` on line 4.

And one last bit of code to fix.

```
present_time = input("Enter the present time in hours:")
set_alarm = input("Set the hours for alarm:")
int(present_time, set_time, alarm_time)
alarm_time = present_time + set_alarm
print(alarm_time)
```

### 3.

**Note 3.4.9 Solution.** In this example the error message is about `set_time` not defined on line 3. In this case the undefined name is not used in an assignment statement, but is used as a parameter (incorrectly) to a function call. A search on `set_time` reveals that in fact it is only used once in the program. Did the author mean `set_alarm`? If we make that assumption we immediately get another error `NameError: name 'alarm_time' is not defined on line: 3`. The variable `alarm_time` is defined on line 4, but that does not help us on line 3. Furthermore we now have to ask the question is this function call `int(present_time, set_alarm, alarm_time)` even the correct use of the `int` function? The answer to that is a resounding no. Let's list all of the things wrong with line 3:

- 1 `set_time` is not defined and never used, the author probably meant `set_alarm`.
- 2 `alarm_time` cannot be used as a parameter before it is defined, even on the next line!
- 3 `int` can only convert one string to an integer at a time.
- 4 Finally, `int` should be used in an assignment statement. Even if `int` was called with the correct number of parameters it would have no real effect.

### 3.4.5 ValueError

Value errors occur when you pass a parameter to a function and the function is expecting a certain limitations on the values, and the value passed is not compatible. We can illustrate that with this particular program in two different ways.

```
current_time_str = input("What is the current time (in
    hours 0-23)?")
current_time_int = int(current_time_str)

wait_time_str = input("How many hours do you want to wait")
wait_time_int = int(wait_time_int)

final_time_int = current_time_int + wait_time_int
print(final_time_int)
```

Run the program but instead of typing in anything to the dialog box just click OK. You should see the following error message: `ValueError: invalid literal for int() with base 10: ''` on line: 4 This error is not because you have made a mistake in your program. Although sometimes we do want to check the user input to make sure its valid, but we don't have all the tools we

need for that yet. The error happens because the user did not give us something we can convert to an integer, instead we gave it an empty string. Try running the program again. Now this time enter “ten” instead of the number 10. You will get a similar error message.

Run the program but instead of typing in anything to the dialog box just click OK. You should see the following error message: `ValueError: invalid literal for int() with base 10: ''` on line: This error is not because you have made a mistake in your program. Although sometimes we do want to check the user input to make sure its valid, but we don’t have all the tools we need for that yet. The error happens because the user did not give us something we can convert to an integer, instead we gave it an empty string. Try running the program again. Now this time enter “ten” instead of the number 10. You will get a similar error message.

ValueErrors are not always caused by user input error, but in this program that is the case. We’ll look again at ValueErrors again when we get to more complicated programs. For now it is worth repeating that you need to keep track of the restrictions needed for your variables, and understand what your function is expecting. You can do this by writing comments in your code, or by naming your variables in a way that reminds you of their proper form.

## 3.5 Summary

- Make sure you take the time to understand error messages. They can help you a lot.
- `print` statements are your friends. Use them to help you uncover what is **really** happening in your code.
- Work backward from the error. Many times an error message is caused by something that has happened before it in the program. Always remember that python evaluates a program top to bottom.

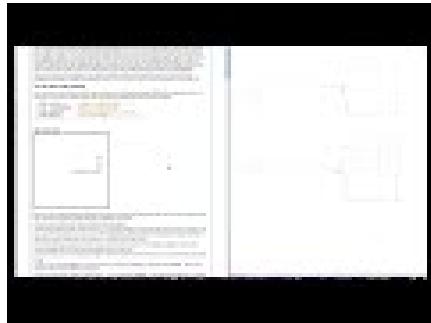
## 3.6 Exercises

This page is intentionally blank (for now)

# Chapter 4

# Python Turtle Graphics

## 4.1 Hello Little Turtles!



There are many *modules* in Python that provide very powerful features that we can use in our own programs. Some of these can send email or fetch web pages. Others allow us to perform complex mathematical calculations. In this chapter we will introduce a module that allows us to create a data object called a **turtle** that can be used to draw pictures.

Turtle graphics, as it is known, is based on a very simple metaphor. Imagine that you have a turtle that understands English. You can tell your turtle to do simple commands such as go forward and turn right. As the turtle moves around, if its tail is down touching the ground, it will draw a line (leave a trail behind) as it moves. If you tell your turtle to lift up its tail it can still move around but will not leave a trail. As you will see, you can make some pretty amazing drawings with this simple capability.

**Note 4.1.1** The turtles are fun, but the real purpose of the chapter is to teach ourselves a little more Python and to develop our theme of *computational thinking*, or *thinking like a computer scientist*. Most of the Python covered here will be explored in more depth later.

## 4.2 Our First Turtle Program

Let's try a couple of lines of Python code to create a new turtle and start drawing a simple figure like a rectangle. We will refer to our first turtle using the variable name alex, but remember that you can choose any name you wish as long as you follow the naming rules from the previous chapter.

The program as shown will only draw the first two sides of the rectangle.

After line 4 you will have a straight line going from the center of the drawing canvas towards the right. After line 6, you will have a canvas with a turtle and a half drawn rectangle. Press the run button to try it and see.

```
import turtle # allows us to use the turtles library
wn = turtle.Screen() # creates a graphics window
alex = turtle.Turtle() # create a turtle named alex
alex.forward(150) # tell alex to move forward by 150 units
alex.left(90) # turn by 90 degrees
alex.forward(75) # complete the second side of a rectangle
```

Here are a couple of things you'll need to understand about this program.

The first line tells Python to load a **module** named **turtle**. That module brings us two new types that we can use: the **Turtle** type, and the **Screen** type. The dot notation **turtle.Turtle** means “*The Turtle type that is defined within the turtle module*”. (Remember that Python is case sensitive, so the module name, **turtle**, with a lowercase t, is different from the type **Turtle** because of the uppercase T.)

We then create and open what the turtle module calls a screen (we would prefer to call it a window, or in the case of this web version of Python simply a canvas), which we assign to variable **wn**. Every window contains a **canvas**, which is the area inside the window on which we can draw.

In line 3 we create a turtle. The variable **alex** is made to refer to this turtle. These first three lines set us up so that we are ready to do some drawing.

In lines 4-6, we instruct the **object** **alex** to move and to turn. We do this by **invoking** or activating **alex**'s **methods** — these are the instructions that all turtles know how to respond to. Here the dot indicates that the methods invoked belong to and refer to the object **alex**.

**Note 4.2.1 Complete the rectangle ....** Modify the program by adding the commands necessary to have *alex* complete the rectangle.

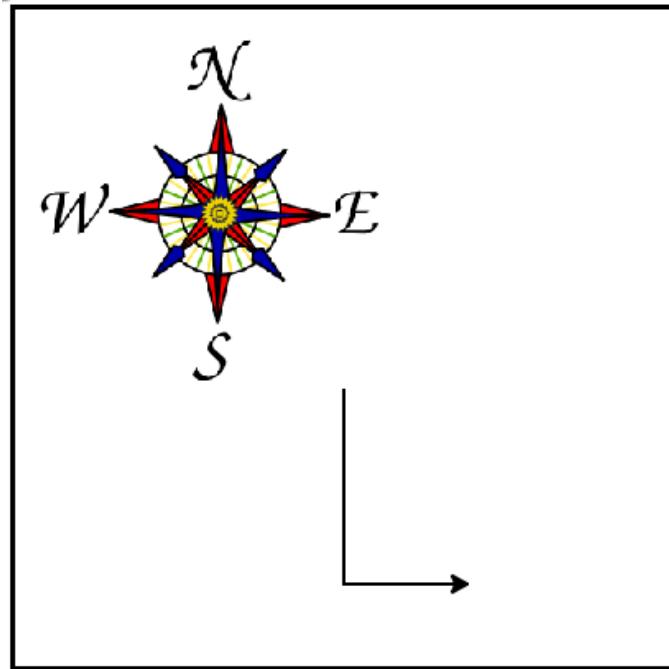
### Check your understanding

**Checkpoint 4.2.2** Which direction does the Turtle face when it is created?

- A. North
- B. South
- C. East
- D. West

### Mixed up programs

**Checkpoint 4.2.3** The following program uses a turtle to draw a capital L as shown in the picture to the left of this text,

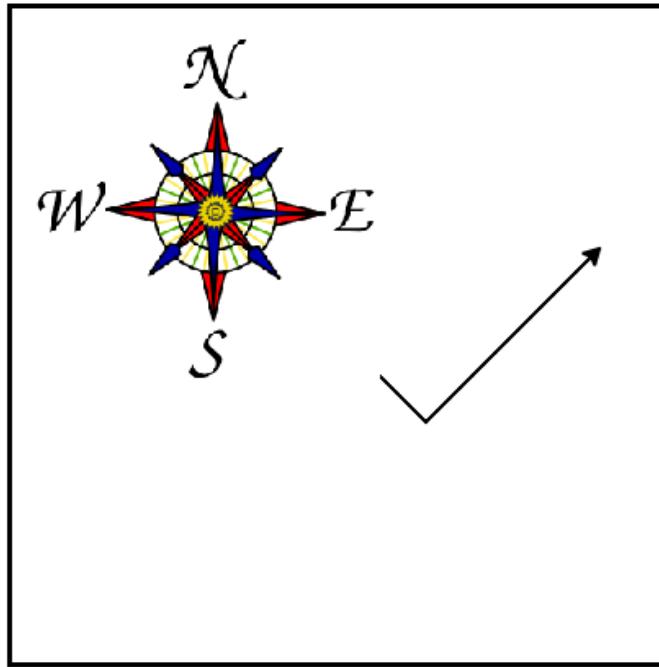


But the lines are mixed up. The program should do all necessary set-up: import the turtle module, get the window to draw on, and create the turtle. Remember that the turtle starts off facing east when it is created. The turtle should turn to face south and draw a line that is 150 pixels long and then turn to face east and draw a line that is 75 pixels long. We have added a compass to the picture to indicate the directions north, south, west, and east.

Drag the blocks of statements from the left column to the right column and put them in the right order. Then click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order.

- `ella.left(90)`  
`ella.forward(75)`
- `import_turtle`  
`window_=turtle.Screen()`  
`ella_=turtle.Turtle()`
- `ella.right(90)`  
`ella.forward(150)`

**Checkpoint 4.2.4** The following program uses a turtle to draw a checkmark as shown to the left:

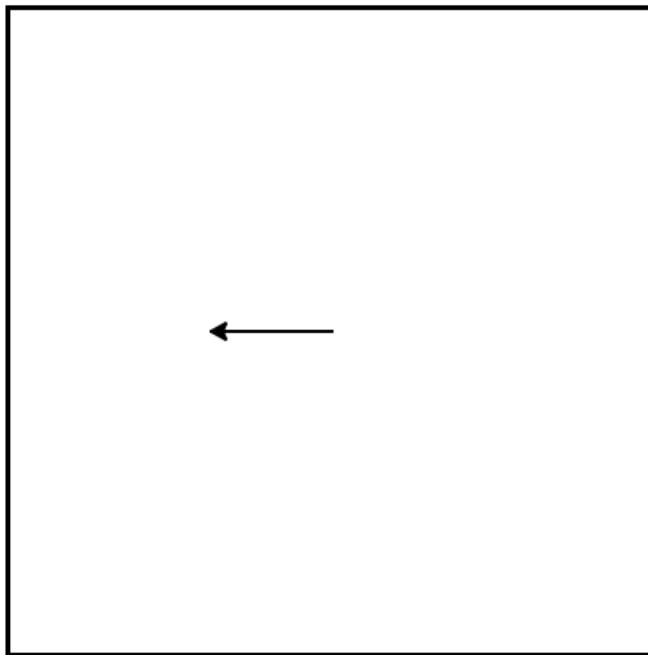


But the lines are mixed up. The program should do all necessary set-up: import the turtle module, get the window to draw on, and create the turtle. The turtle should turn to face southeast, draw a line that is 75 pixels long, then turn to face northeast, and draw a line that is 150 pixels long. We have added a compass to the picture to indicate the directions north, south, west, and east. Northeast is between north and east. Southeast is between south and east.

Drag the blocks of statements from the left column to the right column and put them in the right order. Then click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order.

- `import turtle`
- `maria.right(45)`  
`maria.forward(75)`
- `maria.left(90)`  
`maria.forward(150)`
- `maria=turtle.Turtle()`
- `window=turtle.Screen()`

**Checkpoint 4.2.5** The following program uses a turtle to draw a single line to the west as shown to the left,



But the program lines are mixed up. The program should do all necessary set-up: import the turtle module, get the window to draw on, and create the turtle. The turtle should then turn to face west and draw a line that is 75 pixels long.

Drag the blocks of statements from the left column to the right column and put them in the right order. Then click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order.

- `import turtle`
- `window = turtle.Screen()`
- `jamal = turtle.Turtle()`
- `jamal.left(180)`
- `jamal.forward(75)`

An object can have various methods — things it can do — and it can also have **attributes** — (sometimes called *properties*). For example, each turtle has a *color* attribute. The method invocation `alex.color("red")` will make `alex` red and the line that it draws will be red too.

The color of the turtle, the width of its pen(tail), the position of the turtle within the window, which way it is facing, and so on are all part of its current **state**. Similarly, the window object has a background color which is part of its state.

Quite a number of attributes and methods exist that allow us to modify the turtle and window objects. In the example below, we show just show a couple and have only commented those lines that are different from the previous example. Note also that we have decided to call our turtle object *tess*.

```
import turtle

wn = turtle.Screen()
wn.bgcolor("lightgreen") # set the window background color

tess = turtle.Turtle()
tess.color("blue") # make tess blue
tess.pensize(3) # set the width of her pen
```

```

tess.forward(50)
tess.left(120)
tess.forward(50)

wn.exitonclick() # wait for a user click on the canvas

```

The last line plays a very important role. The `wn` variable refers to the window shown above. When we invoke its `exitonclick` method, the program pauses execution and waits for the user to click the mouse somewhere in the window. When this click event occurs, the response is to close the turtle window and exit (stop execution of) the Python program.

Each time we run this program, a new drawing window pops up, and will remain on the screen until we click on it.

#### Note 4.2.6 Extend this program ....

- 1 Modify this program so that before it creates the window, it prompts the user to enter the desired background color. It should store the user's responses in a variable, and modify the color of the window according to the user's wishes. (Hint: you can find a list of permitted color names at [https://www.w3schools.com/colors/colors\\_names.asp](https://www.w3schools.com/colors/colors_names.asp)<sup>1</sup>. It includes some quite unusual ones, like "PeachPuff" and "HotPink".)
- 2 Do similar changes to allow the user, at runtime, to set tess' color.
- 3 Do the same for the width of tess' pen. *Hint:* your dialog with the user will return a string, but tess' `pensize` method expects its argument to be an `int`. That means you need to convert the string to an `int` before you pass it to `pensize`.

#### Check your understanding

**Checkpoint 4.2.7** Consider the following code:

```

import turtle
wn = turtle.Screen()
alex = turtle.Turtle()
alex.forward(150)
alex.left(90)
alex.forward(75)

```

What does the line "import turtle" do?

- A. It creates a new turtle object that can be used for drawing.
- B. It defines the module `turtle` which will allow you to create a `Turtle` object and draw with it.
- C. It makes the turtle draw half of a rectangle on the screen.
- D. Nothing, it is unnecessary.

**Checkpoint 4.2.8** Why do we type `turtle.Turtle()` to get a new `Turtle` object?

- A. This is simply for clarity. It would also work to just type "`Turtle()`" instead of "`turtle.Turtle()`".
- B. The period (.) is what tells Python that we want to invoke a new object.

---

<sup>1</sup>[https://www.w3schools.com/colors/colors\\_names.asp](https://www.w3schools.com/colors/colors_names.asp)

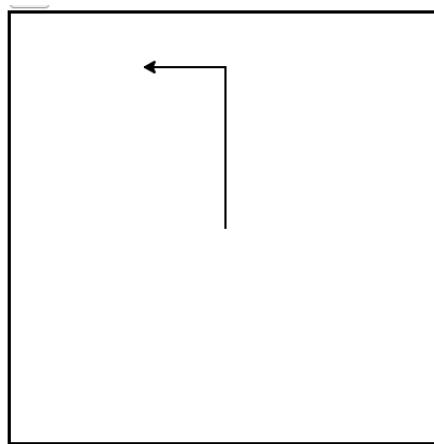
- C. The first "turtle" (before the period) tells Python that we are referring to the turtle module, which is where the object "Turtle" is found.

**Checkpoint 4.2.9** True or False: A Turtle object can have any name that follows the naming rules from Chapter 2.

A. True

B. False

**Checkpoint 4.2.10** Which of the following code would produce the following image?



```
1 import turtle  
2 wn = turtle.Screen()  
3 alex = turtle.Turtle()  
4 alex.right(90)  
5 alex.forward(150)  
6 alex.left(90)  
7 alex.forward(75)
```

A.

---

```
1 import turtle
2 wn = turtle.Screen()
3 alex = turtle.Turtle()
4 alex.left(180)
5 alex.forward(150)
6 alex.left(90)
7 alex.forward(75)
8
9
```

B.

---

```
1 import turtle
2 wn = turtle.Screen()
3 alex = turtle.Turtle()
4 alex.left(270)
5 alex.forward(150)
6 alex.left(90)
7 alex.forward(75)
8
```

C.

---

```
1 import turtle
2 wn = turtle.Screen()
3 alex = turtle.Turtle()
4 alex.right(270)
5 alex.forward(150)
6 alex.right(90)
7 alex.forward(75)
```

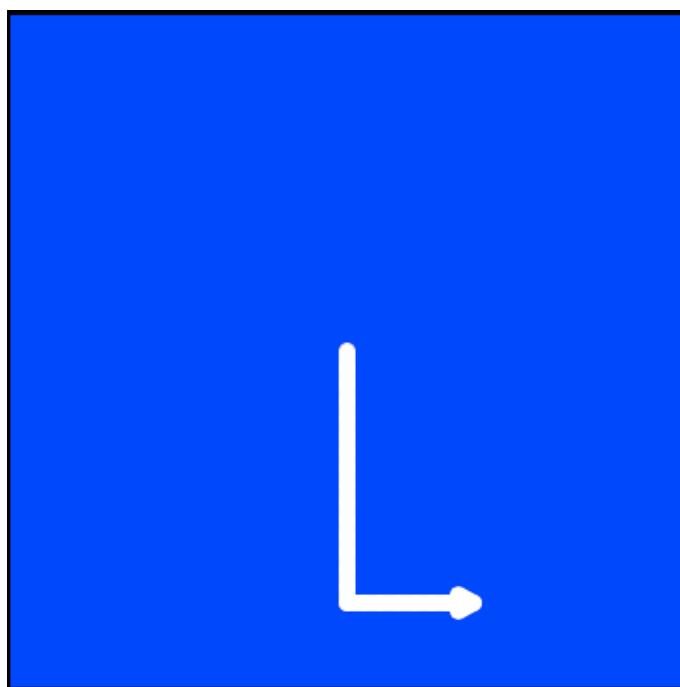
D.

```
1 import turtle  
2 wn = turtle.Screen()  
3 alex = turtle.Turtle()  
4 alex.left(90);  
5 alex.forward(150)  
6 alex.left(90)  
7 alex.forward(75)  
8
```

E.

## Mixed up programs

**Checkpoint 4.2.11** The following program uses a turtle to draw a capital L in white on a blue background as shown to the left,



But the lines are mixed up. The program should do all necessary set-up and create the turtle and set the pen size to 10. The turtle should then turn to face south, draw a line that is 150 pixels long, turn to face east, and draw a line that is 75 pixels long. Finally, set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order. Then click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order.

- import\_turtle  
wn=turtle.Screen()
- jamal.color("white")  
jamal.pensize(10)

- jamal.left(90)  
jamal.forward(75)  
wn.exitonclick()
- wn.bgcolor("blue")  
jamal=turtle.Turtle()
- jamal.right(90)  
jamal.forward(150)

**Checkpoint 4.2.12** The following program uses a turtle to draw a capital T in white on a green background as shown to the left,



But the lines are mixed up. The program should do all necessary set-up, create the turtle, and set the pen size to 10. After that the turtle should turn to face north, draw a line that is 150 pixels long, turn to face west, and draw a line that is 50 pixels long. Next, the turtle should turn 180 degrees and draw a line that is 100 pixels long. Finally, set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order. Then click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order.

- jamal.left(90)  
jamal.forward(150)
- wn.exitonclick()
- import turtle  
wn=turtle.Screen()  
wn.bgcolor("green")  
jamal=turtle.Turtle()  
jamal.color("white")  
jamal.pensize(10)

- jamal.left(90)  
jamal.forward(50)
- jamal.right(180)  
jamal.forward(100)

## 4.3 Instances — A Herd of Turtles

Just like we can have many different integers in a program, we can have many turtles. Each of them is an independent object and we call each one an **instance** of the Turtle type (class). Each instance has its own attributes and methods — so alex might draw with a thin black pen and be at some position, while tess might be going in her own direction with a fat pink pen. So here is what happens when alex completes a square and tess completes her triangle:

```

import turtle
wn = turtle.Screen()                      # Set up the window and
    its attributes
wn.bgcolor("lightgreen")

tess = turtle.Turtle()                     # create tess and set some
    attributes
tess.color("hotpink")
tess.pensize(5)

alex = turtle.Turtle()                     # create alex

tess.forward(80)                          # Let tess draw an
    equilateral triangle
tess.left(120)
tess.forward(80)
tess.left(120)
tess.forward(80)
tess.left(120)                           # complete the triangle

tess.right(180)                          # turn tess around
tess.forward(80)                          # move her away from the
    origin

alex.forward(50)                         # make alex draw a square
alex.left(90)
alex.forward(50)
alex.left(90)
alex.forward(50)
alex.left(90)
alex.forward(50)
alex.left(90)

wn.exitonclick()

```

Here are some *How to think like a computer scientist* observations:

- There are 360 degrees in a full circle. If you add up all the turns that a turtle makes, *no matter what steps occurred between the turns*, you can easily figure out if they add up to some multiple of 360. This should convince you that alex is facing in exactly the same direction as he was

when he was first created. (Geometry conventions have 0 degrees facing East and that is the case here too!)

- We could have left out the last turn for alex, but that would not have been as satisfying. If you’re asked to draw a closed shape like a square or a rectangle, it is a good idea to complete all the turns and to leave the turtle back where it started, facing the same direction as it started in. This makes reasoning about the program and composing chunks of code into bigger programs easier for us humans!
- We did the same with tess: she drew her triangle and turned through a full 360 degrees. Then we turned her around and moved her aside. Even the blank line 18 is a hint about how the programmer’s *mental chunking* is working: in big terms, tess’ movements were chunked as “draw the triangle” (lines 12-17) and then “move away from the origin” (lines 19 and 20).
- One of the key uses for comments is to record your mental chunking, and big ideas. They’re not always explicit in the code.
- And, uh-huh, two turtles may not be enough for a herd, but you get the idea!

### Check your understanding

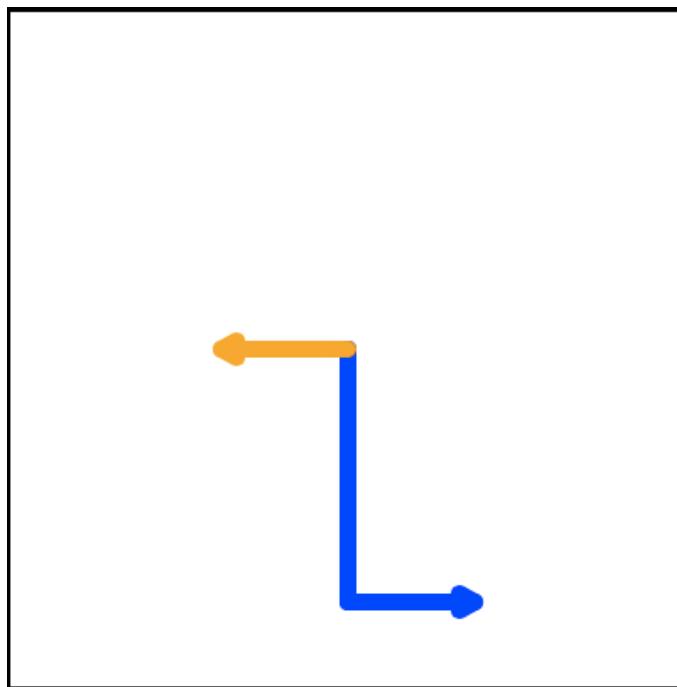
**Checkpoint 4.3.1** True or False: You can only have one active turtle at a time. If you create a second one, you will no longer be able to access or use the first.

A. True

B. False

### Mixed up programs

**Checkpoint 4.3.2** The following program has one turtle, “jamal”, draw a capital L in blue and then another, “tina”, draw a line to the west in orange as shown to the left:

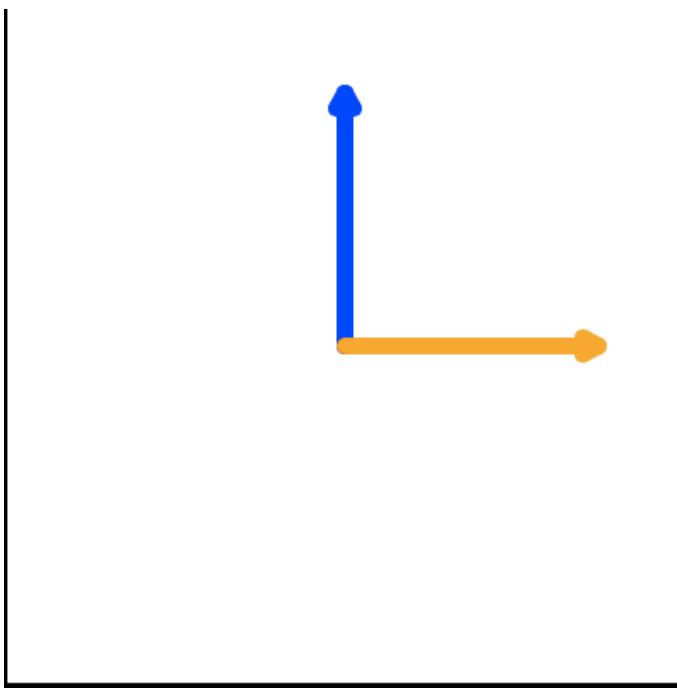


The program should do all set-up, have “jamal” draw the L, and then have “tina” draw the line. Finally, it should set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order. Then click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order.

- import<sub>1</sub>turtle  
wn<sub>1</sub>=<sub>1</sub>turtle.Screen()
- wn.exitonclick()
- jamal<sub>1</sub>=<sub>1</sub>turtle.Turtle()  
jamal.pensize(10)  
jamal.color("blue")  
jamal.right(90)  
jamal.forward(150)
- jamal.left(90)  
jamal.forward(75)
- tina<sub>1</sub>=<sub>1</sub>turtle.Turtle()  
tina.pensize(10)  
tina.color("orange")  
tina.left(180)  
tina.forward(75)

**Checkpoint 4.3.3** The following program has one turtle, “jamal”, draw a line to the north in blue and then another, “tina”, draw a line to the east in orange as shown to the left:

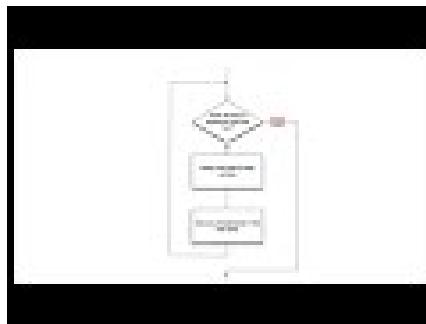


The program should import the turtle module, get the window to draw on, create the turtle “jamal”, have it draw a line to the north, then create the turtle “tina”, and have it draw a line to the east. Finally, it should set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order. Then click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order.

- `wn=turtle.Screen()`
- `tina=turtle.Turtle()  
tina.pensize(10)  
tina.color("orange")  
tina.forward(150)`
- `import turtle`
- `wn.exitonclick()`
- `jamal=turtle.Turtle()  
jamal.color("blue")  
jamal.pensize(10)`
- `jamal.left(90)  
jamal.forward(150)`

## 4.4 The `for` Loop



When we drew the square, it was quite tedious. We had to move then turn, move then turn, etc. etc. four times. If we were drawing a hexagon, or an octagon, or a polygon with 42 sides, it would have been a nightmare to duplicate all that code.

A basic building block of all programs is to be able to repeat some code over and over again. In computer science, we refer to this repetitive idea as **iteration**. In this section, we will explore some mechanisms for basic iteration.

In Python, the **for** statement allows us to write programs that implement iteration. As a simple example, let's say we have some friends, and we'd like to send them each an email inviting them to our party. We don't quite know how to send email yet, so for the moment we'll just print a message for each friend.

```
for name in ["Joe", "Amy", "Brad", "Angelina", "Zuki",
    "Thandi", "Paris"]:
    print("Hi", name, "Please come to my party on"
        "Saturday!")
```

Take a look at the output produced when you press the `run` button. There is one line printed for each friend. Here's how it works:

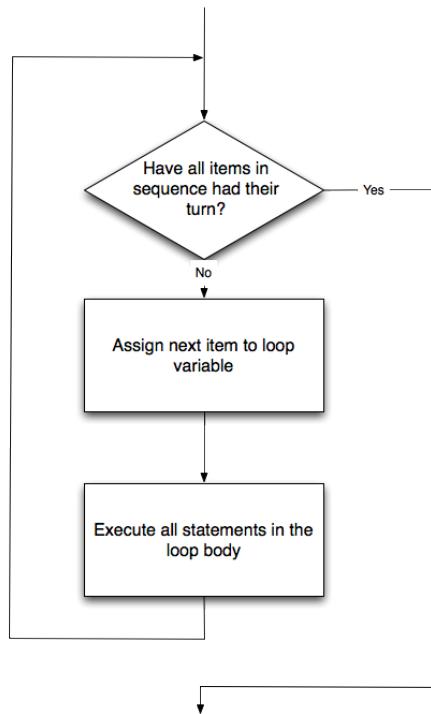
- **name** in this `for` statement is called the **loop variable**.
- The list of names in the square brackets is called a Python **list**. Lists are very useful. We will have much more to say about them later.
- Line 2 is the **loop body**. The loop body is always indented. The indentation determines exactly what statements are “in the loop”. The loop body is performed one time for each name in the list.
- On each *iteration* or *pass* of the loop, a check is done to see if there are still more items to be processed. If there are none left (this is called the **terminating condition** of the loop), the loop has finished. Program execution continues at the next statement after the loop body.
- If there are items still to be processed, the loop variable is updated to refer to the next item in the list. This means, in this case, that the loop body is executed here 7 times, and each time `name` will refer to a different friend.
- At the end of each execution of the body of the loop, Python returns to the `for` statement, to see if there are more items to be handled.

## 4.5 Flow of Execution of the for Loop

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the **control flow**, or the **flow of execution** of the program. When humans execute programs, they often use their finger to point to each statement in turn. So you could think of control flow as “Python’s moving finger”.

Control flow until now has been strictly top to bottom, one statement at a time. We call this type of control **sequential**. In Python flow is sequential as long as successive statements are indented the *same* amount. The `for` statement introduces indented sub-statements after the `for`-loop heading.

Flow of control is often easy to visualize and understand if we draw a flowchart. This flowchart shows the exact steps and logic of how the `for` statement executes.



A codelens demonstration is a good way to help you visualize exactly how the flow of control works with the `for` loop. Try stepping forward and backward through the program by pressing the buttons. You can see the value of `name` change as the loop iterates through the list of friends.

```

for name in ["Joe", "Amy", "Brad", "Angelina", "Zuki",
             "Thandi", "Paris"]:
    print("Hi ", name, ", Please come to my party on "
          "Saturday!")
  
```

## 4.6 Iteration Simplifies our Turtle Program

To draw a square we’d like to do the same thing four times — move the turtle forward some distance and turn 90 degrees. We previously used 8 lines of Python code to have alex draw the four sides of a square. This next program does exactly the same thing but, with the help of the `for` statement, uses just

three lines (not including the setup code). Remember that the for statement will repeat the forward and left four times, one time for each value in the list.

```
import turtle          # set up alex
wn = turtle.Screen()
alex = turtle.Turtle()

for i in [0, 1, 2, 3]:      # repeat four times
    alex.forward(50)
    alex.left(90)

wn.exitonclick()
```

While “saving some lines of code” might be convenient, it is not the big deal here. What is much more important is that we’ve found a “repeating pattern” of statements, and we reorganized our program to repeat the pattern. Finding the chunks and somehow getting our programs arranged around those chunks is a vital skill when learning *How to think like a computer scientist*.

The values [0,1,2,3] were provided to make the loop body execute 4 times. We could have used any four values. For example, consider the following program.

```
import turtle          # set up alex
wn = turtle.Screen()
alex = turtle.Turtle()

for aColor in ["yellow", "red", "purple", "blue"]:
    repeat four times
    alex.forward(50)
    alex.left(90)

wn.exitonclick()
```

In the previous example, there were four integers in the list. This time there are four strings. Since there are four items in the list, the iteration will still occur four times. `aColor` will take on each color in the list. We can even take this one step further and use the value of `aColor` as part of the computation.

```
import turtle          # set up alex
wn = turtle.Screen()
alex = turtle.Turtle()

for aColor in ["yellow", "red", "purple", "blue"]:
    alex.color(aColor)
    alex.forward(50)
    alex.left(90)

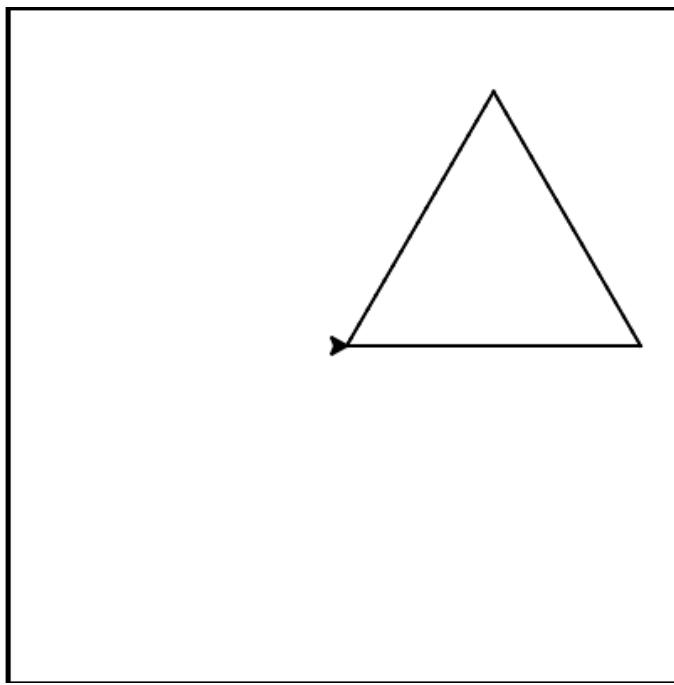
wn.exitonclick()
```

In this case, the value of `aColor` is used to change the color attribute of `alex`. Each iteration causes `aColor` to change to the next value in the list.

The for-loop is our first example of a **compound statement**. Syntactically a compound statement is a statement. The level of indentation of a (whole) compound statement is the indentation of its heading. In the example above there are five statements with the same indentation, executed sequentially: the import, 2 assignments, the *whole* for-loop, and `wn.exitonclick()`. The for-loop compound statement is executed completely before going on to the next sequential statement, `wn.exitonclick()`.

**Mixed up program**

**Checkpoint 4.6.1** The following program uses a turtle to draw a triangle as shown to the left:



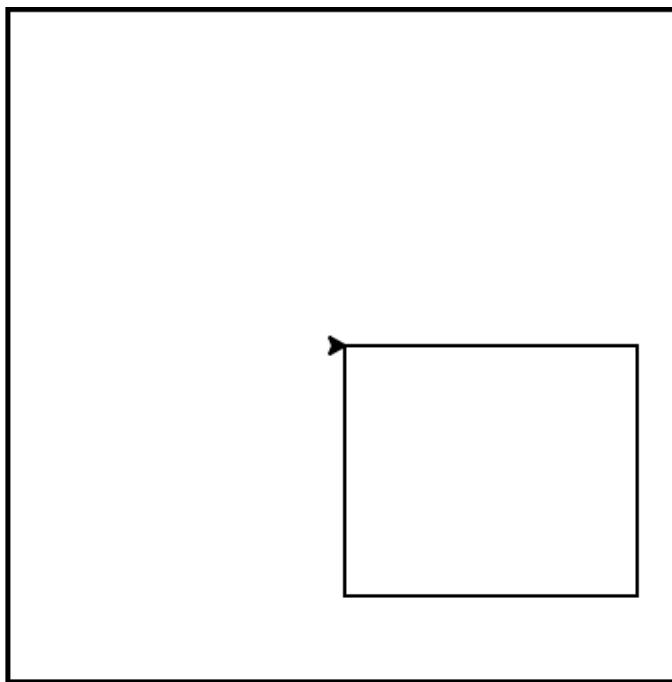
But the lines are mixed up. The program should do all necessary set-up and create the turtle. After that, iterate (loop) 3 times, and each time through the loop the turtle should go forward 175 pixels, and then turn left 120 degrees. After the loop, set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order with the correct indentation. Click on <i>Check Me</i> to see if you are right. You will be told if any of the lines are in the wrong order or are incorrectly indented.

- `#repeat 3 times`
- `for i in [0,1,2]:`
- `marie.forward(175)`
- `marie.left(120)`
- `import turtle`
- `wn=turtle.Screen()`
- `marie=turtle.Turtle()`
- `wn.exitonclick()`

**Mixed up program**

**Checkpoint 4.6.2** The following program uses a turtle to draw a rectangle as shown to the left:



But the lines are mixed up. The program should do all necessary set-up and create the turtle. After that, iterate (loop) 2 times, and each time through the loop the turtle should go forward 175 pixels, turn right 90 degrees, go forward 150 pixels, and turn right 90 degrees. After the loop, set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order with the correct indentation. Click on <i>Check Me</i> to see if you are right. You will be told if any of the lines are in the wrong order or are incorrectly indented.

- wn.exitonclick()
- carlos.forward(175)
- import turtle  
    wn=turtle.Screen()  
    carlos=turtle.Turtle()
- carlos.forward(150)  
    carlos.right(90)
- #repeat 2 times  
    for i in [1,2]:
- carlos.right(90)

#### Check your understanding

**Checkpoint 4.6.3** In the following code, how many lines does this code print?

```
for number in [5, 4, 3, 2, 1, 0]:
    print("I have", number, "cookies. I'm going to eat one.")
```

A. 1

B. 5

C. 6

D. 10

**Checkpoint 4.6.4** How does python know what statements are contained in the loop body?

- A. They are indented to the same degree from the loop header.
- B. There is always exactly one line in the loop body.
- C. The loop body ends with a semi-colon (;) which is not shown in the code above.

**Checkpoint 4.6.5** In the following code, what is the value of number the second time Python executes the loop?

```
for number in [5, 4, 3, 2, 1, 0]:
    print("I have", number, "cookies. I'm going to eat"
          "one.")
```

A. 2

B. 4

C. 5

D. 1

**Checkpoint 4.6.6** Consider the following code:

```
for aColor in ["yellow", "red", "green", "blue"]:
    alex.forward(50)
    alex.left(90)
```

What does each iteration through the loop do?

- A. Draw a square using the same color for each side.
- B. Draw a square using a different color for each side.
- C. Draw one side of a square.

## 4.7 The range Function



In our simple example from the last section (shown again below), we used a list of four integers to cause the iteration to happen four times. We said that we could have used any four values. In fact, we even used four colors.

```

import turtle           # set up alex
wn = turtle.Screen()
alex = turtle.Turtle()

for i in [0, 1, 2, 3]:    # repeat four times
    alex.forward(50)
    alex.left(90)

wn.exitonclick()

```

It turns out that generating lists with a specific number of integers is a very common thing to do, especially when you want to write simple `for` loop controlled iteration. Even though you can use any four items, or any four integers for that matter, the conventional thing to do is to use a list of integers starting with 0. In fact, these lists are so popular that Python gives us special built-in `range` objects that can deliver a sequence of values to the `for` loop. When called with one parameter, the sequence provided by `range` always starts with 0. If you ask for `range(4)`, then you will get 4 values starting with 0. In other words, 0, 1, 2, and finally 3. Notice that 4 is not included since we started with 0. Likewise, `range(10)` provides 10 values, 0 through 9.

```

for i in range(4):
    # Executes the body with i = 0, then 1, then 2, then 3
for x in range(10):
    # sets x to each of ... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

#### Note 4.7.1

Computer scientists like to count from 0!

So to repeat something four times, a good Python programmer would do this:

```

for i in range(4):
    alex.forward(50)
    alex.left(90)

```

The `range`<sup>1</sup> function is actually a very powerful function when it comes to creating sequences of integers. It can take one, two, or three parameters. We have seen the simplest case of one parameter such as `range(4)` which creates `[0, 1, 2, 3]`. But what if we really want to have the sequence `[1, 2, 3, 4]`? We can do this by using a two parameter version of `range` where the first parameter is the starting point and the second parameter is the ending point. The evaluation of `range(1,5)` produces the desired sequence. What happened to the 5? In this case we interpret the parameters of the `range` function to mean `range(start,beyondLast)`, where beyondLast means an index past the last index we want. In the 2-parameter version of `range`, that is the last index included + 1.

**Note 4.7.2** Why in the world would `range` not just work like `range(start, stop)`? Think about it like this. Because computer scientists like to start counting at 0 instead of 1, `range(N)` produces a sequence of things that is N long, but the consequence of this is that the final number of the sequence is N-1. In the case of `start, stop` it helps to simply think that the sequence begins with `start` and continues as long as the number is less than `stop`.

**Note 4.7.3** The `range` function is *lazy*: It produces the next element only when needed. With a regular Python 3 interpreter, printing a `range` does *not* calculate all the elements. To immediately calculate all the elements in a `range`, wrap the `range` in a list, like `list(range(4))`. Activecode is not designed to

---

<sup>1</sup><http://docs.python.org/py3k/library/functions.html?highlight=range#range>

work on very long sequences, and it may allow you to be sloppy, avoiding the `list` function, and *see* the elements in the range with `print(range(4))`.

Here are two examples for you to run. Try them and then add another line below to create a sequence starting at 10 and going up to 20 (including 20).

```
print(list(range(4)))
print(list(range(1, 5)))
```

Codelens will help us to further understand the way range works. In this case, the variable `i` will take on values produced by the `range` function.

```
for i in range(10):
    print(i)
```

Finally, suppose we want to have a sequence of even numbers. How would we do that? Easy, we add another parameter, a step, that tells range what to count by. For even numbers we want to start at 0 and count by 2's. So if we wanted the first 10 even numbers we would use `range(0, 19, 2)`. The most general form of the range is `range(start, beyondLast, step)`. You can also create a sequence of numbers that starts big and gets smaller by using a negative value for the step parameter.

```
print(list(range(0, 19, 2)))
print(list(range(0, 20, 2)))
print(list(range(10, 0, -1)))
```

Try it in codelens. Do you see why the first two statements produce the same result?

```
for i in range(0, 20, 2):
    print(i)
```

### Check your understanding

**Checkpoint 4.7.4** In the command `range(3, 10, 2)`, what does the second argument (10) specify?

- A. Range should generate a sequence that stops before 10 (including 9).
- B. Range should generate a sequence that starts at 10 (including 10).
- C. Range should generate a sequence starting at 3 that stops at 10 (including 10).
- D. Range should generate a sequence using every 10th number between the start and the stopping number.

**Checkpoint 4.7.5** What command correctly generates the sequence 2, 5, 8?

- A. `range(2, 5, 8)`
- B. `range(2, 8, 3)`
- C. `range(2, 10, 3)`
- D. `range(8, 1, -3)`

**Checkpoint 4.7.6** What happens if you give range only one argument? For example: `range(4)`

- A. It will generate a sequence starting at 0, with every number included up to but not including the argument it was passed.
- B. It will generate a sequence starting at 1, with every number up to but

not including the argument it was passed.

- C. It will generate a sequence starting at 1, with every number including the argument it was passed.
- D. It will cause an error: range always takes exactly 3 arguments.

**Checkpoint 4.7.7** Which range function call will produce the sequence 20, 15, 10, 5?

- A. range(5, 25, 5)
- B. range(20, 3, -5)
- C. range(20, 5, 4)
- D. range(20, 5, -5)

**Checkpoint 4.7.8** What could the second parameter (12) in range(2, 12, 4) be replaced with and generate exactly the same sequence?

- A. No other value would give the same sequence.
- B. The only other choice is 14.
- C. 11, 13, or 14

## 4.8 A Few More turtle Methods and Observations

Here are a few more things that you might find useful as you write programs that use turtles.

- Turtle methods can use negative angles or distances. So `tess.forward(-100)` will move tess backwards, and `tess.left(-30)` turns her to the right. Additionally, because there are 360 degrees in a circle, turning 30 to the left will leave you facing in the same direction as turning 330 to the right! (The on-screen animation will differ, though — you will be able to tell if tess is turning clockwise or counter-clockwise!)

This suggests that we don't need both a left and a right turn method — we could be minimalists, and just have one method. There is also a *backward* method. (If you are very nerdy, you might enjoy saying `alex.backward(-100)` to move alex forward!)

Part of *thinking like a scientist* is to understand more of the structure and rich relationships in your field. So reviewing a few basic facts about geometry and number lines, like we've done here is a good start if we're going to play with turtles.

- A turtle's pen can be picked up or put down. This allows us to move a turtle to a different place without drawing a line. The methods are `up` and `down`. Note that the methods `penup` and `pendown` do the same thing.

```

alex.up()
alex.forward(100)      # this moves alex, but no line
                      is drawn
alex.down()
```

- Every turtle can have its own shape. The ones available “out of the box” are `arrow`, `blank`, `circle`, `classic`, `square`, `triangle`, `turtle`.

```
| ...
| alex.shape("turtle")
| ...
|
```

- You can speed up or slow down the turtle’s animation speed. (Animation controls how quickly the turtle turns and moves forward). Speed settings can be set between 1 (slowest) to 10 (fastest). But if you set the speed to 0, it has a special meaning — turn off animation and go as fast as possible.
- | `alex.speed(10)`
- A turtle can “stamp” its footprint onto the canvas, and this will remain after the turtle has moved somewhere else. Stamping works even when the pen is up.

Let’s do an example that shows off some of these new features.

```
import turtle
wn = turtle.Screen()
wn.bgcolor("lightgreen")
tess = turtle.Turtle()
tess.color("blue")
tess.shape("turtle")

print(list(range(5, 60, 2)))
tess.up()                      # this is new
for size in range(5, 60, 2):    # start with size = 5 and
    grow by 2
    tess.stamp()                 # leave an impression on
        the canvas
    tess.forward(size)           # move tess along
    tess.right(24)               # and turn her

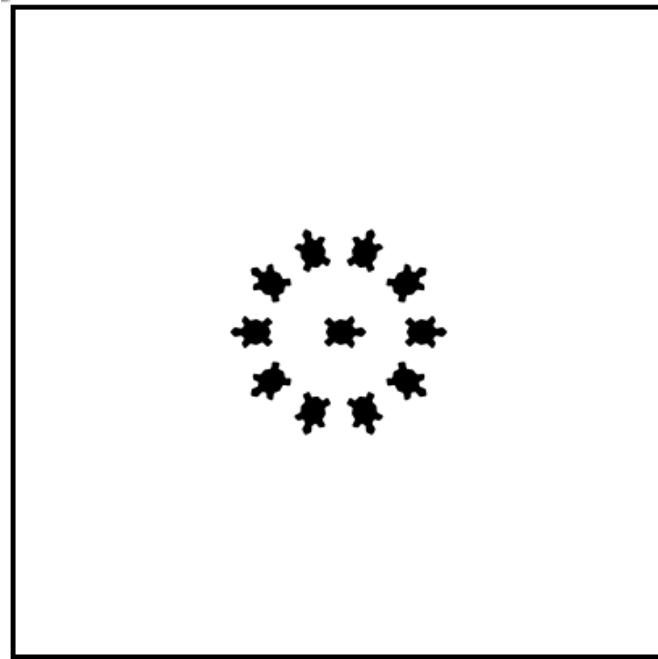
wn.exitonclick()
```

The list of integers printed above for `list(range(5,60,2))` is only displayed to show you the distances being used to move the turtle forward. The actual use appears as part of the `for` loop.

One more thing to be careful about. All except one of the shapes you see on the screen here are footprints created by `stamp`. But the program still only has *one* turtle instance — can you figure out which one is the real tess? (Hint: if you’re not sure, write a new line of code after the `for` loop to change tess’ color, or to put her pen down and draw a line, or to change her shape, etc.)

### Mixed up program

**Checkpoint 4.8.1** The following program uses the `stamp` method to create a circle of turtle shapes as shown to the left:



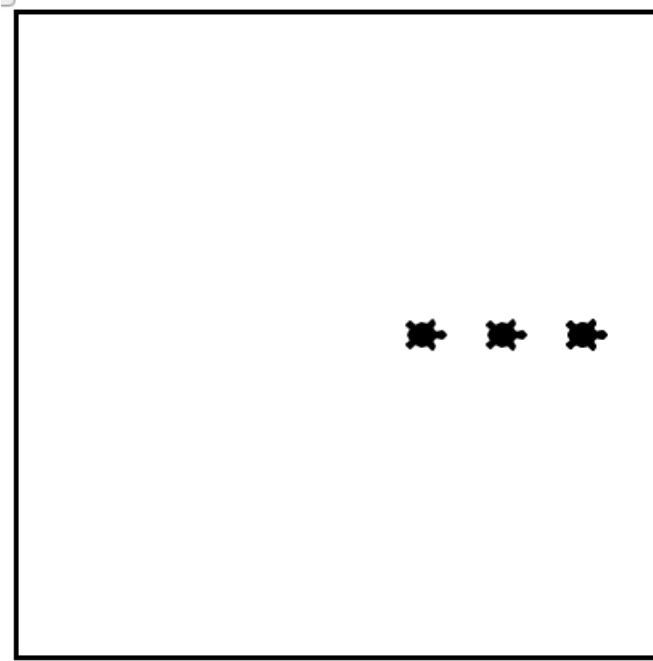
But the lines are mixed up. The program should do all necessary set-up, create the turtle, set the shape to “turtle”, and pick up the pen. Then the turtle should repeat the following ten times: go forward 50 pixels, leave a copy of the turtle at the current position, reverse for 50 pixels, and then turn right 36 degrees. After the loop, set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order with the correct indentation. Click on <i>Check Me</i> to see if you are right. You will be told if any of the lines are in the wrong order or are incorrectly indented.

- wn.exitonclick()
- jose.forward(-50)
- import turtle  
wn=turtle.Screen()  
jose=turtle.Turtle()  
jose.shape("turtle")  
jose.penup()
- jose.forward(50)
- for size in range(10):
- jose.stamp()
- jose.right(36)

#### Mixed up program

**Checkpoint 4.8.2** The following program uses the stamp method to create a line of turtle shapes as shown to the left:



But the lines are mixed up. The program should do all necessary set-up, create the turtle, set the shape to “turtle”, and pick up the pen. Then the turtle should repeat the following three times: go forward 50 pixels and leave a copy of the turtle at the current position. After the loop, set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order with the correct indentation. Click on <i>Check Me</i> to see if you are right. You will be told if any of the lines are in the wrong order or are incorrectly indented.

- `nikea.stamp()`
- `nikea.shape("turtle")`
- `wn.exitonclick()`
- `nikea.forward(50)`
- `nikea=turtle.Turtle()`
- `import turtle`
- `wn=turtle.Screen()`
- `nikea.penup()`
- `for size in range(3):`

#### Note 4.8.3 Lab.

- Turtle Race In this guided lab exercise [Section 21.2](#) we will work through a simple problem solving exercise related to having some turtles race.

## 4.9 Summary of Turtle Methods

**Table 4.9.1**

Method	Parameters	Description
Turtle	None	Creates and returns a new turtle object
forward	distance	Moves the turtle forward
backward	distance	Moves the turtle backward
right	angle	Turns the turtle clockwise
left	angle	Turns the turtle counter clockwise
up	None	Picks up the turtles tail
down	None	Puts down the turtles tail
color	color name	Changes the color of the turtle's tail
fillcolor	color name	Changes the color of the turtle will use to fill a polygon
heading	None	Returns the current heading
position	None	Returns the current position
goto	x,y	Move the turtle to position x,y
begin_fill	None	Remember the starting point for a filled polygon
end_fill	None	Close the polygon and fill with the current fill color
dot	None	Leave a dot at the current position
stamp	None	Leaves an impression of a turtle shape at the current location
shape	shape name	Should be 'arrow', 'classic', 'turtle', 'circle' or 'square'

Once you are comfortable with the basics of turtle graphics you can read about even more options on the [Python Docs Website](#)<sup>1</sup>. Note that we will describe Python Docs in more detail in the next chapter.

**Note 4.9.2** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 4.10 Glossary

### Glossary

**attribute.** Some state or value that belongs to a particular object. For example, tess has a color.

**canvas.** A surface within a window where drawing takes place.

**control flow.** See *flow of execution* in the next chapter.

**for loop.** A statement in Python for convenient repetition of statements in the *body* of the loop.

**instance.** An object that belongs to a class. tess and alex are different instances of the class Turtle

**invoke.** An object has methods. We use the verb invoke to mean *activate the method*. Invoking a method is done by putting parentheses after the method name, with some possible arguments. So wn.exitonclick() is an invocation of the exitonclick method.

**iteration.** A basic building block for algorithms (programs). It allows steps to be repeated. Sometimes called *looping*.

**loop body.** Any number of statements nested inside a loop. The nesting is indicated by the fact that the statements are indented under the for loop statement.

---

<sup>1</sup><http://docs.python.org/dev/py3k/library/turtle.html>

**loop variable.** A variable used as part of a for loop. It is assigned a different value on each iteration of the loop, and is used as part of the terminating condition of the loop, when it can no longer get a further value.

**method.** A function that is attached to an object. Invoking or activating the method causes the object to respond in some way, e.g. `forward` is the method when we say `tess.forward(100)`.

**module.** A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the *import* statement.

**object.** A “thing” to which a variable can refer. This could be a screen window, or one of the turtles you have created.

**range.** A built-in function in Python for generating sequences of integers. It is especially useful when we need to write a for loop that executes a fixed number of times.

**sequential.** The default behavior of a program. Step by step processing of algorithm.

**state.** The collection of attribute values that a specific data object maintains.

**terminating condition.** A condition that occurs which causes a loop to stop repeating its body. In the `for` loops we saw in this chapter, the terminating condition has been when there are no more elements to assign to the loop variable.

**turtle.** A data object used to create pictures (known as turtle graphics).

## 4.11 Exercises

1. Write a program that prints We like Python's turtles! 100 times.
2. Turtle objects have methods and attributes. For example, a turtle has a position and when you move the turtle forward, the position changes. Think about the other methods shown in the summary above. Which attributes, if any, does each method relate to? Does the method change the attribute?

3.

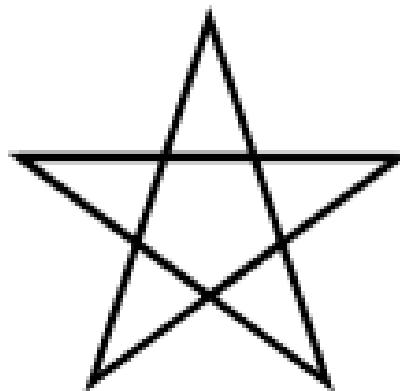
<b>Write a program that uses a for loop to print</b>	One of the months of the year is January One of the months of the year is February One of the months of the year is March etc ...
--	--

4. Assume you have a list of numbers 12, 10, 32, 3, 66, 17, 42, 99, 20
  - a. Write a loop that prints each of the numbers on a new line.
  - b. Write a loop that prints each number and its square on a new line.
5. Use `for` loops to make a turtle draw these regular polygons (regular means all sides the same lengths, all angles the same):
  - An equilateral triangle
  - A square
  - A hexagon (six sides)
  - An octagon (eight sides)

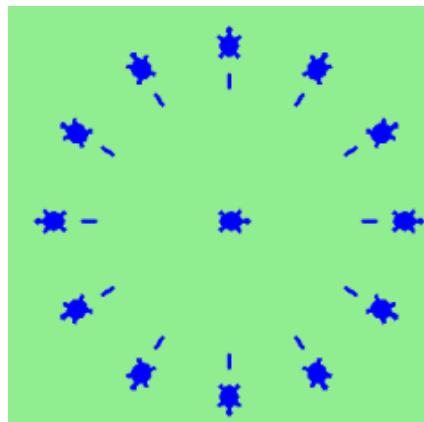
6. Write a program that asks the user for the number of sides, the length of the side, the color, and the fill color of a regular polygon. The program should draw the polygon and then fill it in.
7. A drunk pirate makes a random turn and then takes 100 steps forward, makes another random turn, takes another 100 steps, turns another random amount, etc. A social science student records the angle of each turn before the next 100 steps are taken. Her experimental data is 160, -43, 270, -97, -43, 200, -940, 17, -86. (Positive angles are counter-clockwise.) Use a turtle to draw the path taken by our drunk friend. After the pirate is done walking, print the current heading.
8. On a piece of scratch paper, trace the following program and show the drawing. When you are done, press run and check your answer.

```
import turtle
wn = turtle.Screen()
tess = turtle.Turtle()
tess.right(90)
tess.left(3600)
tess.right(-90)
tess.left(3600)
tess.left(3645)
tess.forward(-100)
```

9. Write a program to draw a shape like this:



10. Write a program to draw a face of a clock that looks something like this:



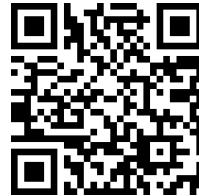
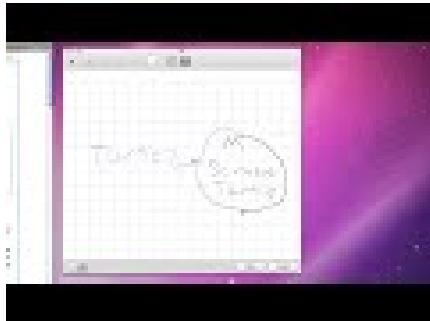
11. Write a program to draw some kind of picture. Be creative and experiment with the turtle methods provided in [Section 4.9](#).

12. Create a turtle and assign it to a variable. When you print its type, what do you get?
13. A sprite is a simple spider shaped thing with n legs coming out from a center point. The angle between each leg is  $360 / n$  degrees.  
Write a program to draw a sprite where the number of legs is provided by the user.

# Chapter 5

## Python Modules

### 5.1 Modules and Getting Help



A **module** is a file containing Python definitions and statements intended for use in other Python programs. There are many Python modules that come with Python as part of the **standard library**. We have already used one of these quite extensively, the **turtle** module. Recall that once we import the module, we can use things that are defined inside.

```
import turtle          # allows us to use the turtles
                      # library

wn = turtle.Screen()    # creates a graphics window
alex = turtle.Turtle()   # create a turtle named alex

alex.forward(150)        # tell alex to move forward by 150
                       # units
alex.left(90)           # turn by 90 degrees
alex.forward(75)         # complete the second side of a
                       # rectangle
wn.exitonclick()
```

Here we are using **Screen** and **Turtle**, both of which are defined inside the **turtle** module.

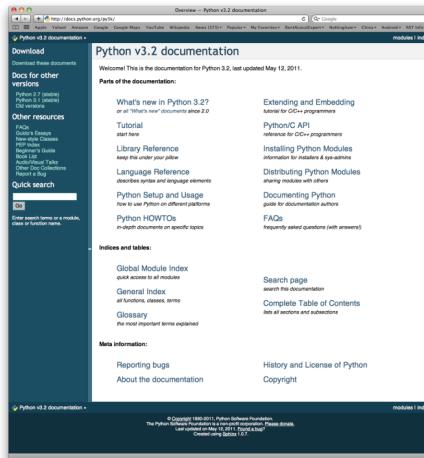
But what if no one had told us about **turtle**? How would we know that it exists. How would we know what it can do for us? The answer is to ask for help and the best place to get help about the Python programming environment is to consult with the Python Documentation.

The [Python Documentation](#)<sup>1</sup> site for Python version 3 (the home page

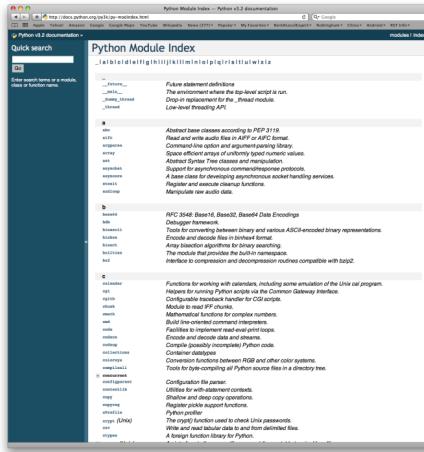
---

<sup>1</sup><http://docs.python.org/py3k/>

is shown below) is an extremely useful reference for all aspects of Python. The site contains a listing of all the standard modules that are available with Python (see [Global Module Index](#)<sup>2</sup>). You will also see that there is a [Language Reference](#)<sup>3</sup> and a [Tutorial](#)<sup>4</sup> (mostly aimed at people who are already familiar with another programming language), as well as installation instructions, how-tos, and frequently asked questions. We encourage you to become familiar with this site and to use it often.



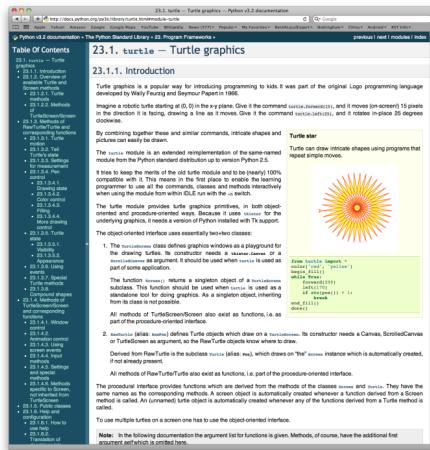
If you have not done so already, take a look at the Global Module Index. Here you will see an alphabetical listing of all the modules that are available as part of the standard library. Find the turtle module.



<sup>2</sup><http://docs.python.org/py3k/py-modindex.html>

<sup>3</sup><http://docs.python.org/py3k/reference/index.html>

<sup>4</sup><http://docs.python.org/py3k/tutorial/index.html>



You can see that all the turtle functionality that we have talked about is there. However, there is so much more. Take some time to read through and familiarize yourself with some of the other things that turtles can do.

**Note 5.1.1 Note: Python modules and limitations with activecode.** Throughout the chapters of this book, activecode windows allow you to practice the Python that you are learning. We mentioned in the first chapter that programming is normally done using some type of development environment and that the activecode used here was strictly to help us learn. It is not the way we write production programs.

To that end, it is necessary to mention that many of the modules available in standard Python will **not** work in the activecode environment. In fact, only turtle, math, and random have been completely ported at this point. If you wish to explore any additional modules, you will need to also explore using a more robust development environment.

### Check your understanding

**Checkpoint 5.1.2** In Python a module is:

- A file containing Python definitions and statements intended for use in other Python programs.
- A separate block of code within a program.
- One line of code in a program.
- A file that contains documentation about functions in Python.

**Checkpoint 5.1.3** To find out information on the standard modules available with Python you should:

- Go to the Python Documentation site.
- Look at the import statements of the program you are working with or writing.
- Ask the professor
- Look in this textbook.

**Checkpoint 5.1.4 True / False:** All standard Python modules will work in activecode.

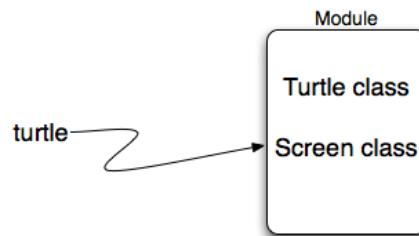
- True

B. False

## 5.2 More About Using Modules

Before we move on to exploring other modules, we should say a bit more about what modules are and how we typically use them. One of the most important things to realize about modules is the fact that they are data objects, just like any other data in Python. Module objects simply contain other Python elements.

The first thing we need to do when we wish to use a module is perform an `import`. In the example above, the statement `import turtle` creates a new name, `turtle`, and makes it refer to a module object. This looks very much like the reference diagrams we saw earlier for simple variables.



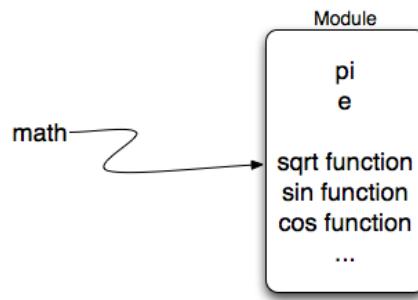
In order to use something contained in a module, we use the dot notation, providing the module name and the specific item joined together with a “dot”. For example, to use the `Turtle` class, we say `turtle.Turtle`. You should read this as: “In the module `turtle`, access the Python element called `Turtle`”.

We will now turn our attention to a few other modules that you might find useful.



## 5.3 The math module

The `math` module contains the kinds of mathematical functions you would typically find on your calculator and some mathematical constants like `pi` and `e`. As we noted above, when we `import math`, we create a reference to a module object that contains these elements.



Here are some items from the math module in action. If you want more information, you can check out the [Math Module<sup>1</sup>](#) Python Documentation.

```
import math

print(math.pi)
print(math.e)

print(math.sqrt(2.0))

print(math.sin(math.radians(90)))    # sin of 90 degrees
```

Notice another difference between this module and our use of turtle. In turtle we create objects (either `Turtle` or `Screen`) and call methods on those objects. Remember that a turtle is a data object (recall `alex` and `tess`). We need to create one in order to use it. When we say `alex = turtle.Turtle()`, we are calling the constructor for the `Turtle` class which returns a single turtle object.

Mathematical functions do not need to be constructed. They simply perform a task. They are all housed together in a module called `math`. Once we have imported the `math` module, anything defined there can be used in our program. Notice that we always use the name of the module followed by a dot followed by the specific item from the module (`math.sqrt`). You can think of this as `lastname.firstname` where the `lastname` is the module family and the `firstname` is the individual entry in the module.

If you have not done so already, take a look at the documentation for the `math` module.

#### Check your understanding

**Checkpoint 5.3.1** Which statement allows you to use the `math` module in your program?

- A. import math
- B. include math
- C. use math
- D. You don't need a statement. You can always use the `math` module

## 5.4 The random module

We often want to use **random numbers** in programs. Here are a few typical uses:

---

<sup>1</sup><http://docs.python.org/py3k/library/math.html#module-math>

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin,
- To shuffle a deck of playing cards randomly,
- To randomly allow a new enemy spaceship to appear and shoot at you,
- To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam,
- For encrypting your banking session on the Internet.

Python provides a module `random` that helps with tasks like this. You can take a look at it in the documentation. Here are the key things we can do with it.

```
import random

prob = random.random()
print(prob)

diceThrow = random.randrange(1, 7) # return an int, one of
                                1,2,3,4,5,6
print(diceThrow)
```

Press the run button a number of times. Note that the values change each time. These are random numbers.

The `randrange` function generates an integer between its lower and upper argument, using the same semantics as `range` — so the lower bound is included, but the upper bound is excluded. All the values have an equal probability of occurring (i.e. the results are *uniformly* distributed).

The `random()` function returns a floating point number in the range [0.0, 1.0) — the square bracket means “closed interval on the left” and the round parenthesis means “open interval on the right”. In other words, 0.0 is possible, but all returned numbers will be strictly less than 1.0. It is usual to *scale* the results after calling this method, to get them into a range suitable for your application.

In the case shown here, we’ve converted the result of the method call to a number in the range [0.0, 5.0). Once more, these are uniformly distributed numbers — numbers close to 0 are just as likely to occur as numbers close to 0.5, or numbers close to 1.0. If you continue to press the run button you will see random values between 0.0 and up to but not including 5.0.

```
import random

prob = random.random()
result = prob * 5
print(result)
```

It is important to note that random number generators are based on a **deterministic** algorithm — repeatable and predictable. So they’re called **pseudo-random** generators — they are not genuinely random. They start with a *seed* value. Each time you ask for another random number, you’ll get one based on the current seed attribute, and the state of the seed (which is one of the attributes of the generator) will be updated. The good news is that each time you run your program, the seed value is likely to be different meaning that even though the random numbers are being created algorithmically, you will likely get random behavior each time you execute.

**Note 5.4.1 Lab.**

- In this guided lab exercise we will have the turtle plot a sine wave. [Section 21.10](#)

**Check your understanding**

**Checkpoint 5.4.2** Which of the following is the correct way to reference the value pi within the math module. Assume you have already imported the math module.

- A. math.pi
- B. math(pi)
- C. pi.math
- D. math->pi

**Checkpoint 5.4.3** Which module would you most likely use if you were writing a function to simulate rolling dice?

- A. the math module
- B. the random module
- C. the turtle module
- D. the game module

**Checkpoint 5.4.4** The correct code to generate a random number between 1 and 100 (inclusive) is:

- A. prob = random.randrange(1, 101)
- B. prob = random.randrange(1, 100)
- C. prob = random.randrange(0, 101)
- D. prob = random.randrange(0, 100)

**Checkpoint 5.4.5** One reason that lotteries don't use computers to generate random numbers is:

- A. There is no computer on the stage for the drawing.
- B. Because computers don't really generate random numbers, they generate pseudo-random numbers.
- C. They would just generate the same numbers over and over again.
- D. The computer can't tell what values were already selected, so it might generate all 5's instead of 5 unique numbers.

**Note 5.4.6** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 5.5 Creating Modules

You've seen how to *use* modules like `random`, `math`, and `turtle`, but how would you *create* a module?

Every time you've written a Python script you've created a module!

A Python module is just a Python source code file. Let's consider the Python file shown below.

```
coffee_shop.py

"""
The coffee shop module contains functions and contains variables
important to implementing a coffee shop.

"""

# Set some variables
shop_name = "Runestone Brew House"
coffee_sizes = ["small", "medium", "large"]
coffee_roasts = ["hot chocolate", "light", "medium",
                 "dark", "espresso"]
```

This is a Python script named `coffee_shop.py` that contains three variables: `shop_name`, `coffee_sizes`, and `coffee_roasts`. The `shop_name` is a string, `coffee_sizes` is a list containing strings, and `coffee_roasts` is also a list containing strings.

**Checkpoint 5.5.1** A module is another name for:

- A. the code inside a function
- B. a file containing Python code
- C. the comments before a function
- D. a small block of Python code

That's so great! We've got the basics of a coffee shop. All you need is some roasted coffee and cups. You're good to go.

If you try to run that code though, it doesn't do much that's visible to a user...

How can we use the `coffee_shop` module? We can import it and use it in other Python source code files. Let's consider the Python file shown below.

```
coffee_customer.py

import coffee_shop

# Output the information we know from the module
print("Welcome to", coffee_shop.shop_name)
print("Available sizes:", coffee_shop.coffee_sizes)
print("Available roasts:", coffee_shop.coffee_roasts)
```

This is a Python script named `coffee_customer.py` that imports our `coffee_shop` module, then prints out the information from that module.

**Note 5.5.2 Note.** The module files must be in the same directory on your computer for Python to know how to import them automatically

**Checkpoint 5.5.3** Create a module by:

- A. writing a new function or class
- B. placing an import statement at the top of a file
- C. placing code in a Python file in the same directory as your other source code
- D. creating a comment block at the beginning of a file

We use **dot notation** to grab the `shop_name`, `coffee_sizes`, and `coffee_roasts` variables from the `coffee_shop` module. Then we print them out as parts of nice messages.

Variables aren't the only thing we can place in modules though... We can put any valid Python code in them.

Let's improve our coffee shop!

`coffee_shop.py`

```
"""
The coffee shop module contains functions and contains
variables
important to implementing a coffee shop.

"""

# Set some variables
shop_name = "Runestone Brew House"
coffee_sizes = ["small", "medium", "large"]
coffee_roasts = ["hot chocolate", "light", "medium",
                 "dark", "espresso"]

def order_coffee(size, roast):
    """
    Take an order from a user
    :param size: a string containing one of the coffee_sizes
    :param roast: a string containing one of the coffee_roasts
    :return: a message about the coffee order
    """
    return "Here's your {} coffee roasted {}".format(size,
                                                      roast)
```

The old file contents are present, but now there's also an `order_coffee` function that takes two arguments, `size` and `roast`.

Also - look at all the awesome comments in there!

**Note 5.5.4 Module Comments.** It is important to include header comments in your module that explain what the module does.

**Note 5.5.5 Function Comments.** Functions are the next chapter, but the comments used here demonstrate a common Python documentation style.

Ok - so we've got a function in our module now, let's use it.

`coffee_customer.py`

```
# Import the module with coffee_shop functionality
import coffee_shop

# Output the information we know from the module
print("Welcome to", coffee_shop.shop_name)
print("Available sizes:", coffee_shop.coffee_sizes)
print("Available roasts:", coffee_shop.coffee_roasts)

# Get some inputs from the user
order_size = input("What size coffee do you want?")
order_roast = input("What roast do you want?")

# Send the order to the coffee shop module
shop_says = coffee_shop.order_coffee(order_size,
                                      order_roast)
# Print out whatever it gave back to us
print(shop_says)
```

**Checkpoint 5.5.6** What determines the name of our import?

- A. the first variable name in the module
- B. a comment early in the module
- C. it's called whatever we name it in the "import" statement
- D. the filename of the module

We added some lines to our `coffee_customer` script... Now after printing data nicely, `coffee_customer` asks the user for a size and a roast. These are the parameters required by our `order_coffee` function over in the `coffee_shop` module!

Call the `order_coffee` function with **dot notation**, just like retrieving variable values. The function call is the line that says `shop_says = coffee_shop.order_coffee(order_size, order_roast)`. The function returns something, so we save that off in `shop_says`. The next line prints out whatever the shop said.

Coffee shops do more than just coffee! Maybe you want some milk. We need to add some functionality to our coffee shop. Check it out below.

*coffee\_shop.py*

```
"""
The coffee shop module contains functions and contains variables
important to implementing a coffee shop.

"""

# Set some variables
shop_name = "Runestone Brew House"
coffee_sizes = ["small", "medium", "large"]
coffee_roasts = ["hot_chocolate", "light", "medium",
                 "dark", "espresso"]

def order_coffee(size, roast):
    """
    Take an order from a user
    :param size: a string containing one of the coffee_sizes
    :param roast: a string containing one of the coffee_roasts
    :return: a message about the coffee order
    """
    return "Here's your {} coffee roasted {}".format(size,
                                                      roast)

def add_milk_please(fat_content):
    """
    Pretend like we're adding some milk to a coffee
    :param fat_content: a string or integer containing the milkfat content
    :return: a message about having added the milk
    """
    return "I've added the {}% milk".format(fat_content)
```

The new function is called `add_milk_please` and it takes one parameter - the `fat_content`. It returns a string explaining what happened.

This is great. But the function isn't going to do anything by itself. We have to call it. Check out the update to our `coffee_customer` script below.

*coffee\_customer.py*

```

# Import the module with coffee_shop functionality
import coffee_shop

# Output the information we know from the module
print("Welcome to", coffee_shop.shop_name)
print("Available sizes:", coffee_shop.coffee_sizes)
print("Available roasts:", coffee_shop.coffee_roasts)

# Get some inputs from the user
order_size = input("What size coffee do you want?")
order_roast = input("What roast do you want?")

# Send the order to the coffee shop module
shop_says = coffee_shop.order_coffee(order_size,
                                      order_roast)
# Print out whatever it gave back to us
print(shop_says)

# See if the user wants to add milk
add_milk_response = input("Do you want to add milk (y/n)?")
# Convert the response to lowercase, then check for a "yes"
answer
if "y" in add_milk_response.lower():
    milk_fat = input("What percent milk do you want added?")
    shop_says = coffee_shop.add_milk_please(milk_fat)
# Print out whatever it gave back to us
print(shop_says)

```

That got fancy! We were just ordering coffee but now the user can choose to add milk! Selection is in a couple chapters, but if you read that code like english you'll see what's going on.

The call to `add_milk_please` happens right in there - it looks just like the other one: `shop_says = coffee_shop.add_milk_please(milk_fat)`.

Let's wrap this coffee shop visit up. But - you better leave a tip. We'll add another function to our coffee shop to enable that.

*coffee\_shop.py*

```

"""
The coffee shop module contains functions and contains variables
important to implementing a coffee shop.

"""

# Set some variables
shop_name = "Runestone Brew House"
coffee_sizes = ["small", "medium", "large"]
coffee_roasts = ["hot chocolate", "light", "medium",
                 "dark", "espresso"]

def order_coffee(size, roast):
    """
    Take an order from a user
    :param size: a string containing one of the coffee_sizes
    :param roast: a string containing one of the coffee_roasts
    :return: a message about the coffee order
    """
    return "Here's your {} coffee roasted {}".format(size,

```

```

        roast)

def add_milk_please(fat_content):
    """
    Pretend like we're adding some milk to a coffee
    :param fat_content: a string or integer containing the
        milkfat content
    :return: a message about having added the milk
    """
    return "I've added the {}% milk".format(fat_content)

def give_tip(tip_amount):
    """
    Take a tip from the user, then be happy about it
    :param tip_amount: the tip amount
    :return: nothing
    """
    print("Thank you so much! We don't make a ton of
        money.")

# Not having a "return" statement causes our function
# to return None

```

We added the `give_tip` function which takes one parameter, the `tip_amount`. We don't actually do anything with that parameter... But if we were getting fancier with the coffee shop we might add it to the customer's bill, we might print it out, or we might berate the customer for being too cheap... Here we just go ahead and blurt out a thanks to the user! Bein' friendly is important.

How do we call this from our `coffee_customer` script?

`coffee_customer.py`

```

# Import the module with coffee_shop functionality
import coffee_shop

# Output the information we know from the module
print("Welcome to", coffee_shop.shop_name)
print("Available sizes:", coffee_shop.coffee_sizes)
print("Available roasts:", coffee_shop.coffee_roasts)

# Get some inputs from the user
order_size = input("What size coffee do you want?")
order_roast = input("What roast do you want?")

# Send the order to the coffee shop module
shop_says = coffee_shop.order_coffee(order_size,
                                      order_roast)
# Print out whatever it gave back to us
print(shop_says)

# See if the user wants to add milk
add_milk_response = input("Do you want to add milk (y/n)?")
# Convert the response to lowercase, then check for a "yes"
answer
if "y" in add_milk_response.lower():
    milk_fat = input("What percent milk do you want added?")
    shop_says = coffee_shop.add_milk_please(milk_fat)
# Print out whatever it gave back to us
print(shop_says)

```

```
# They better give a tip...
print("THAT'S GOOD COFFEE! Very good. Your brain is
      working again.")
print("You better give a tip.")
tip_amount = input("Tip amount? ")
coffee_shop.give_tip(tip_amount)
```

Our function call is there on the last line.

## 5.6 Glossary

### Glossary

**deterministic.** A process that is repeatable and predictable.

**documentation.** A place where you can go to get detailed information about aspects of your programming language.

**module.** A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the *import* statement.

**pseudo-random number.** A number that is not genuinely random but is instead created algorithmically.

**random number.** A number that is generated in such a way as to exhibit statistical randomness.

**random number generator.** A function that will provide you with random numbers, usually between 0 and 1.

**standard library.** A collection of modules that are part of the normal installation of Python.

## 5.7 Exercises

1. Use a `for` statement to print 10 random numbers.
2. Repeat the above exercise but this time print 10 random numbers between 25 and 35, inclusive.
3. The **Pythagorean Theorem** tells us that the length of the hypotenuse of a right triangle is related to the lengths of the other two sides. Look through the `math` module and see if you can find a function that will compute this relationship for you. Once you find it, write a short program to try it out.
4. Search on the internet for a way to calculate an approximation for `pi`. There are many that use simple arithmetic. Write a program to compute the approximation and then print that value as well as the value of `math.pi` from the `math` module.

# Chapter 6

## Functions

### 6.1 Functions



In Python, a **function** is a named sequence of statements that belong together. Their primary purpose is to help us organize programs into chunks that match how we think about the solution to the problem.

The syntax for a **function definition** is:

```
def name( parameters ):  
    statements
```

You can make up any names you want for the functions you create, except that you can't use a name that is a Python keyword, and the names must follow the rules for legal identifiers that were given previously. The parameters specify what information, if any, you have to provide in order to use the new function. Another way to say this is that the parameters specify what the function needs to do its work.

There can be any number of statements inside the function, but they have to be indented from the `def`. In the examples in this book, we will use the standard indentation of four spaces. Function definitions are the second of several **compound statements** we will see, all of which have the same pattern:

- 1 A header line which begins with a keyword and ends with a colon.
- 2 A **body** consisting of one or more Python statements, each indented the same amount – *4 spaces is the Python standard* – from the header line.

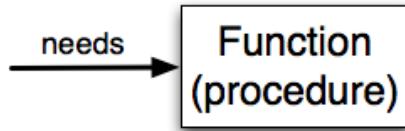
We've already seen the `for` loop which follows this pattern.

In a function definition, the keyword in the header is `def`, which is followed by the name of the function and some *parameters* enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters

separated from one another by commas. In either case, the parentheses are required.

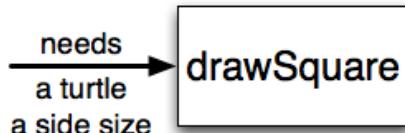
We need to say a bit more about the parameters. In the definition, the parameter list is more specifically known as the **formal parameters**. This list of names describes those things that the function will need to receive from the user of the function. When you use a function, you provide values to the formal parameters.

The figure below shows this relationship. A function needs certain information to do its work. These values, often called **arguments** or **actual parameters**, are passed to the function by the user.



This type of diagram is often called a **black-box diagram** because it only states the requirements from the perspective of the user. The user must know the name of the function and what arguments need to be passed. The details of how the function works are hidden inside the “black-box”.

Suppose we’re working with turtles and a common operation we need is to draw squares. It would make sense if we did not have to duplicate all the steps each time we want to make a square. “Draw a square” can be thought of as an *abstraction* of a number of smaller steps. We will need to provide two pieces of information for the function to do its work: a turtle to do the drawing and a size for the side of the square. We could represent this using the following black-box diagram.



Here is a program containing a function to capture this idea. Give it a try.

```

import turtle

def drawSquare(t, sz):
    """Make turtle t draw a square of side sz."""
    for i in range(4):
        t.forward(sz)
        t.left(90)

wn = turtle.Screen()          # Set up the window and
     its attributes
wn.bgcolor("lightgreen")

alex = turtle.Turtle()         # create alex
drawSquare(alex, 50)          # Call the function to
     draw the square passing the actual turtle and the
     actual side size
  
```

```
| wn.onclick()
```

This function is named `drawSquare`. It has two parameters — one to tell the function which turtle to move around and the other to tell it the size of the square we want drawn. In the function definition they are called `t` and `sz` respectively. Make sure you know where the body of the function ends — it depends on the indentation and the blank lines don't count for this purpose!

**Note 6.1.1 docstrings.** If the first thing after the function header is a string (some tools insist that it must be a triple-quoted string), it is called a **docstring** and gets special treatment in Python and in some of the programming tools.

Another way to retrieve this information is to use the interactive interpreter, and enter the expression `<function_name>.__doc__`, which will retrieve the docstring for the function. So the string you write as documentation at the start of a function is retrievable by python tools *at runtime*. This is different from comments in your code, which are completely eliminated when the program is parsed.

By convention, Python programmers use docstrings for the key documentation of their functions.

Defining a new function does not make the function run. To do that we need a **function call**. This is also known as a **function invocation**. We've already seen how to call some built-in functions like `print`, `range` and `int`. Function calls contain the name of the function to be executed followed by a list of values in parentheses, called *arguments*, which are assigned to the parameters in the function definition. So in the second to the last line of the program, we call the function, and pass `alex` as the turtle to be manipulated, and `50` as the size of the square we want.

Once we've defined a function, we can call it as often as we like and its statements will be executed each time we call it. In this case, we could use it to get one of our turtles to draw a square and then we can move the turtle and have it draw a different square in a different location. Note that we lift the tail so that when `alex` moves there is no trace. We put the tail back down before drawing the next square. Make sure you can identify both invocations of the `drawSquare` function.

```
import turtle

def drawSquare(t, sz):
    """Make turtle t draw a square of size sz."""
    for i in range(4):
        t.forward(sz)
        t.left(90)

wn = turtle.Screen()          # Set up the window and its
                             # attributes
wn.bgcolor("lightgreen")

alex = turtle.Turtle()         # create alex
drawSquare(alex, 50)          # Call the function to draw
                             # the square

alex.penup()
alex.goto(100,100)
alex.pendown()
```

```

    drawSquare(alex,75)           # Draw another square

    wn.exitonclick()

```

In the next example, we've changed the `drawSquare` function a little and we get `tess` to draw 15 squares with some variations. Once the function has been defined, we can call it as many times as we like with whatever actual parameters we like.

```

import turtle

def drawMulticolorSquare(t, sz):
    """Make turtle draw a multi-colour square of sz."""
    for i in ['red','purple','hotpink','blue']:
        t.color(i)
        t.forward(sz)
        t.left(90)

wn = turtle.Screen()          # Set up the window and
     its attributes
wn.bgcolor("lightgreen")

tess = turtle.Turtle()         # create tess and set some
     attributes
tess.pensize(3)

size = 20                      # size of the smallest
     square
for i in range(15):
    drawMulticolorSquare(tess, size)
    size = size + 10            # increase the size for
     next time
    tess.forward(10)             # move tess along a little
    tess.right(18)               # and give her some extra
     turn

wn.exitonclick()

```

**Warning 6.1.2** Even if a function call needs no arguments, the parentheses ( ) after the function name are *required*. This can lead to a difficult bug: A function name without the parenthesis is a legal expression *referring* to the function; for example, `print` and `alex.penup`, but they do not *call* the associated functions.

**Note 6.1.3** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

#### Check your understanding

**Checkpoint 6.1.4** What is a function in Python?

- A. A named sequence of statements.
- B. Any sequence of statements.
- C. A mathematical expression that calculates a value.
- D. A statement of the form  $x = 5 + 4$ .

**Checkpoint 6.1.5** What is one main purpose of a function?

- A. To improve the speed of execution
- B. To help the programmer organize programs into chunks that match how

they think about the solution to the problem.

- C. All Python programs must be written using functions
- D. To calculate values.

**Checkpoint 6.1.6** Which of the following is a valid function header (first line of a function definition)?

- A. def drawCircle(t):
- B. def drawCircle:
- C. drawCircle(t, sz):
- D. def drawCircle(t, sz)

**Checkpoint 6.1.7** What is the name of the following function?

```
def drawSquare(t, sz):
    """Make turtle t draw a square of with side sz."""
    for i in range(4):
        t.forward(sz)
        t.left(90)
```

- A. def drawSquare(t, sz)
- B. drawSquare
- C. drawSquare(t, sz)
- D. Make turtle t draw a square with side sz.

**Checkpoint 6.1.8** What are the parameters of the following function?

```
def drawSquare(t, sz):
    """Make turtle t draw a square of with side sz."""
    for i in range(4):
        t.forward(sz)
        t.left(90)
```

- A. i
- B. t
- C. t, sz
- D. t, sz, i

**Checkpoint 6.1.9** Considering the function below, which of the following statements correctly invokes, or calls, this function (i.e., causes it to run)? Assume we already have a turtle named alex.

```
def drawSquare(t, sz):
    """Make turtle t draw a square of with side sz."""
    for i in range(4):
        t.forward(sz)
        t.left(90)
```

- A. def drawSquare(t, sz)
- B. drawSquare
- C. drawSquare(10)

- D. drawSquare(alex, 10):
- E. drawSquare(alex, 10)

**Checkpoint 6.1.10** True or false: A function can be called several times by placing a function call in the body of a loop.

- A. True
- B. False

## 6.2 Functions that Return Values

Most functions require arguments, values that control how the function does its job. For example, if you want to find the absolute value of a number, you have to indicate what the number is. Python has a built-in function for computing the absolute value:

```
print(abs(5))
print(abs(-5))
```

In this example, the arguments to the `abs` function are 5 and -5.

Some functions take more than one argument. For example the `math` module contains a function called `pow` which takes two arguments, the base and the exponent.

```
import math
print(math.pow(2, 3))

print(math.pow(7, 4))
```

**Note 6.2.1** Of course, we have already seen that raising a base to an exponent can be done with the `**` operator.

Another built-in function that takes more than one argument is `max`.

```
print(max(7, 11))
print(max(4, 1, 17, 2, 12))
print(max(3 * 11, 5 ** 3, 512 - 9, 1024 ** 0))
```

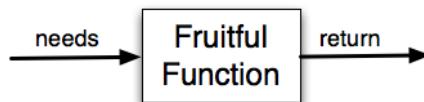
`max` can be sent any number of arguments, separated by commas, and will return the maximum value sent. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1. Note that `max` also works on lists of values.

Furthermore, functions like `range`, `int`, `abs` all return values that can be used to build more complex expressions.

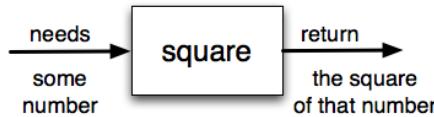
So an important difference between these functions and one like `drawSquare` is that `drawSquare` was not executed because we wanted it to compute a value — on the contrary, we wrote `drawSquare` because we wanted it to execute a sequence of steps that caused the turtle to draw a specific shape.

Functions that return values are sometimes called **fruitful functions**. In many other languages, a chunk that doesn't return a value is called a **procedure**, but we will stick here with the Python way of also calling it a function, or if we want to stress it, a *non-fruitful* function.

Fruitful functions still allow the user to provide information (arguments). However there is now an additional piece of data that is returned from the function.



How do we write our own fruitful function? Let's start by creating a very simple mathematical function that we will call `square`. The `square` function will take one number as a parameter and return the result of squaring that number. Here is the black-box diagram with the Python code following.



```

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

toSquare = 10
result = square(toSquare)
print("The result of", toSquare, "squared is", result)

```

The `return` statement is followed by an expression which is evaluated. Its result is returned to the caller as the “fruit” of calling this function. Because the `return` statement can contain any Python expression we could have avoided creating the **temporary variable** `y` and simply used `return x*x`. Try modifying the `square` function above to see that this works just the same. On the other hand, using **temporary variables** like `y` in the program above makes debugging easier. These temporary variables are examples of **local variables**, pursued further in the next section.

Notice something important here. The name of the variable we pass as an argument — `toSquare` — has nothing to do with the name of the formal parameter — `x`. It is as if `x = toSquare` is executed when `square` is called. It doesn't matter what the value was named in the caller. In `square`, its name is `x`. You can see this very clearly in codelens, where the **global variables** (variables defined outside of any function) and the local variables for the `square` function are in separate boxes.

As you step through the example in codelens notice that the `return` statement not only causes the function to return a value, but it also returns the flow of control back to the place in the program where the function call was made. this is true in general:

**Note 6.2.2** The call to a function *terminates* after the execution of a `return` statement. This is fairly obvious if the `return` statement is the last statement in the function, but we will see later where it makes sense to have a `return` statement even when other statements follow, and the further statements are *not* executed.

```

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

toSquare = 10
squareResult = square(toSquare)
print("The result of", toSquare, "squared is", squareResult)

```

Another important thing to notice as you step through this codelens demon-

stration is the movement of the red and green arrows. Codelens uses these arrows to show you where it is currently executing. Recall that the red arrow always points to the next line of code that will be executed. The light green arrow points to the line that was just executed in the last step.

When you first start running this codelens demonstration you will notice that there is only a red arrow and it points to line 1. This is because line 1 is the next line to be executed and since it is the first line, there is no previously executed line of code.

When you click on the forward button, notice that the red arrow moves to line 5, skipping lines 2 and 3 of the function (and the light green arrow has now appeared on line 1). Why is this? The answer is that function definition is not the same as function execution. Lines 2 and 3 will not be executed until the function is called on line 6. Line 1 defines the function and the name `square` is added to the global variables, but that is all the `def` does at that point. The body of the function will be executed later. Continue to click the forward button to see how the flow of control moves from the call, back up to the body of the function, and then finally back to line 7, after the function has returned its value and the value has been assigned to `squareResult`.

Finally, there is one more aspect of function return values that should be noted. All Python functions return the value `None` unless there is an explicit return statement with a value other than `None`. Consider the following common mistake made by beginning Python programmers. As you step through this example, pay very close attention to the return value in the local variables listing. Then look at what is printed when the function returns.

```
def square(x):
    y = x * x
    print(y)    # Bad! should use return instead!

toSquare = 10
squareResult = square(toSquare)
print("The result of", toSquare, "squared is", squareResult)
```

The problem with this function is that even though it prints the value of the square, that value will not be returned to the place where the call was made. Since line 6 uses the return value as the right hand side of an assignment statement, the evaluation of the function will be `None`. In this case, `squareResult` will refer to that value after the assignment statement and therefore the result printed in line 7 is incorrect. Typically, functions will return values that can be printed or processed in some other way by the caller.

### Check your understanding

**Checkpoint 6.2.3** What is wrong with the following function definition:

```
def addEm(x, y, z):
    return x + y + z
    print('the answer is', x + y + z)
```

- A. You should never use a `print` statement in a function definition.
- B. You should not have any statements in a function after the `return` statement. Once the function gets to the `return` statement it will immediately stop executing the function.
- C. You must calculate the value of  $x+y+z$  before you `return` it.
- D. A function cannot `return` a number.

**Checkpoint 6.2.4** What will the following function return?

```
def addEm(x, y, z):
    print(x + y + z)
```

- A. None
- B. The value of  $x + y + z$
- C. The string ' $x + y + z$ '

## 6.3 Unit Testing

### 6.3.1 Introduction

A **test case** expresses requirements for a program, in a way that can be checked automatically. Specifically, a test asserts something about the state of the program at a particular point in its execution. A **unit test** is an automatic procedure used to validate that individual units of code are working properly. A function is one form of a unit. A collection of these unit tests is called a **test suite**.

We have previously suggested that it's a good idea to first write down comments about what your code is supposed to do, before actually writing the code. It is an even better idea to write down some test cases before writing a program.

There are several reasons why it's a good habit to write test cases.

- Before we write code, we have in mind what it *should* do, but those thoughts may be a little vague. Writing down test cases forces us to be more concrete about what should happen.
- As we write the code, the test cases can provide automated feedback. You've actually been the beneficiary of such automated feedback via test cases throughout this book in some of the activecode windows and almost all of the exercises. We wrote the code for those test cases but kept it hidden, so as not to confuse you and also to avoid giving away the answers. You can get some of the same benefit from writing your own test cases.
- In larger software projects, the set of test cases can be run every time a change is made to the code base. **Unit tests** check that small bits of code are correctly implemented.

One way to implement unit tests in Python is with `assert`.

- Following the word `assert` there will be a python expression.
- If that expression evaluates to the Boolean `False`, then the interpreter will raise a runtime error.
- If the expression evaluates to `True`, then nothing happens and the execution goes on to the next line of code.

Take a look at the way `assert` is used in the following code.

```
assert type(9//5) == int
assert type(9.0//5) == int
```

In the code above, we explicitly state some natural assumptions about how truncated division might work in python. It turns out that the second assumption is wrong: `9.0//5` produces `2.0`, a floating point value!

The python interpreter does not enforce restrictions about the data types of objects that can be bound to particular variables; however, type checking could alert us that something has gone wrong in our program execution. If we are assuming at that `x` is a list, but it's actually an integer, then at some point later in the program execution, there will probably be an error. We can add `assert` statements that will cause an error to be flagged sooner rather than later, which might make it a lot easier to debug.

#### Check your understanding

**Checkpoint 6.3.1** When `assert x==y` is executed and `x` and `y` have the same values, what will happen?

- A. A runtime error will occur
- B. A message is printed out saying that the test failed.
- C. `x` will get the value that `y` currently has
- D. Nothing will happen
- E. A message is printed out saying that the test passed.

#### 6.3.2 assert with for loops

Why would you ever want to write a line of code that can never compute anything useful for you, but sometimes causes a runtime error? For all the reasons we described above about the value of automated tests. You want a test that will alert that you that some condition you assumed was true is not in fact true. It's much better to be alerted to that fact right away than to have some unexpected result much later in your program execution, which you will have trouble tracing to the place where you had an error in your code.

Why doesn't `assert` print out something saying that the test passed? The reason is that you don't want to clutter up your output window with the results of automated tests that pass. You just want to know when one of your tests fails. In larger projects, other testing harnesses are used instead of `assert`, such as the python `unittest` module. Those provide some output summarizing tests that have passed as well as those that failed. In this textbook, we will just use simple `assert` statements for automated tests.

In the code below, `lst` is bound to a list object. In python, not all the elements of a list have to be of the same type. We can check that they all have the same type and get an error if they are not. Notice that with `lst2`, one of the assertions fails.

```
lst = ['a', 'b', 'c']
first_type = type(lst[0])
for item in lst:
    assert type(item) == first_type

lst2 = ['a', 'b', 'c', 17]
first_type = type(lst2[0])
for item in lst2:
    assert type(item) == first_type
```

#### 6.3.3 Return Value Tests

Testing whether a function returns the correct value is the easiest test case to define. You simply check whether the result of invoking the function on a

particular input produces the particular output that you expect. Take a look at the following code.

```
def square(x):
    #raise x to the second power
    return x*x
print('testing square function')
assert square(3) == 9
```

Because each test checks whether a function works properly on specific inputs, the test cases will never be complete: in principle, a function might work properly on all the inputs that are tested in the test cases, but still not work properly on some other inputs. That's where the art of defining test cases comes in: you try to find specific inputs that are representative of all the important kinds of inputs that might ever be passed to the function.

**Checkpoint 6.3.2** For the hangman game, this ‘blanked’ function takes a word and some letters that have been guessed, and returns a version of the word with \_ for all the letters that haven’t been guessed. Which of the following is the correct way to write a test to check that ‘under’ will be blanked as ‘u\_d\_\_’ when the user has guessed letters d and u so far?

- A. assert blanked('under', 'du', 'u\_d\_\_') == True
- B. assert blanked('under', 'u\_d\_\_') == 'du'
- C. assert blanked('under', 'du') == 'u\_d\_\_'

## 6.4 Variables and Parameters are Local

An assignment statement in a function creates a **local variable** for the variable on the left hand side of the assignment operator. It is called local because this variable only exists inside the function and you cannot use it outside. For example, consider again the `square` function:

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

z = square(10)
print(y)
```

If you press the ‘last >>’ button you will see an error message. When we try to use `y` on line 6 (outside the function) Python looks for a global variable named `y` but does not find one. This results in the error: `Name Error: 'y' is not defined.`

The variable `y` only exists while the function is being executed — we call this its **lifetime**. When the execution of the function terminates (returns), the local variables are destroyed. Codelens helps you visualize this because the local variables disappear after the function returns. Go back and step through the statements paying particular attention to the variables that are created when the function is called. Note when they are subsequently destroyed as the function returns.

Formal parameters are also local and act like local variables. For example, the lifetime of `x` begins when `square` is called, and its lifetime ends when the function completes its execution.

So it is not possible for a function to set some local variable to a value, complete its execution, and then when it is called again next time, recover the local variable. Each call of the function creates new local variables, and their lifetimes expire when the function returns to the caller.

On the other hand, it is legal for a function to access a global variable. However, this is considered **bad form** by nearly all programmers and should be avoided. Look at the following, nonsensical variation of the square function.

```
def badsquare(x):
    y = x ** power
    return y

power = 2
result = badsquare(10)
print(result)
```

Although the `badsquare` function works, it is silly and poorly written. We have done it here to illustrate an important rule about how variables are looked up in Python. First, Python looks at the variables that are defined as local variables in the function. We call this the **local scope**. If the variable name is not found in the local scope, then Python looks at the global variables, or **global scope**. This is exactly the case illustrated in the code above. `power` is not found locally in `badsquare` but it does exist globally. The appropriate way to write this function would be to pass `power` as a parameter. For practice, you should rewrite the `badsquare` example to have a second parameter called `power`.

There is another variation on this theme of local versus global variables. Assignment statements in the local function cannot change variables defined outside the function, without further (discouraged) special syntax. Consider the following codelens example:

```
def powerof(x, p):
    power = p      # Another dumb mistake
    y = x ** power
    return y

power = 3
result = powerof(10, 2)
print(result)
```

Now step through the code. What do you notice about the values of variable `power` in the local scope compared to the variable `power` in the global scope?

The value of `power` in the local scope was different than the global scope. That is because in this example `power` was used on the left hand side of the assignment statement `power = p`. When a variable name is used on the left hand side of an assignment statement Python creates a local variable. When a local variable has the same name as a global variable we say that the local shadows the global. A **shadow** means that the global variable cannot be accessed by Python because the local variable will be found first. This is another good reason not to use global variables. As you can see, it makes your code confusing and difficult to understand.

To cement all of these ideas even further lets look at one final example. Inside the `square` function we are going to make an assignment to the parameter `x`. There's no good reason to do this other than to emphasize the fact that the parameter `x` is a local variable. If you step through the example in codelens you will see that although `x` is 0 in the local variables for `square`, the `x` in the global scope remains 2. This is confusing to many beginning programmers who think

that an assignment to a formal parameter will cause a change to the value of the variable that was used as the actual parameter, especially when the two share the same name. But this example demonstrates that that is clearly not how Python operates.

```
def square(x):
    y = x * x
    x = 0          # assign a new value to the parameter x
    return y

x = 2
z = square(x)
print(z)
```

### Check your understanding

**Checkpoint 6.4.1** What is a variable's scope?

- A. Its value
- B. The range of statements in the code where a variable can be accessed.
- C. Its name

**Checkpoint 6.4.2** What is a local variable?

- A. A temporary variable that is only used inside a function
- B. The same as a parameter
- C. Another name for any variable

**Checkpoint 6.4.3** Can you use the same name for a local variable as a global variable?

- A. Yes, and there is no reason not to.
- B. Yes, but it is considered bad form.
- C. No, it will cause an error.

## 6.5 The Accumulator Pattern

### 6.5.1 Introduction



In the previous example, we wrote a function that computes the square of a number. The algorithm we used in the function was simple: multiply the number by itself. In this section we will reimplement the square function and use a different algorithm, one that relies on addition instead of multiplication.

If you want to multiply two numbers together, the most basic approach is to think of it as repeating the process of adding one number to itself. The number of repetitions is where the second number comes into play. For example, if we wanted to multiply three and five, we could think about it as adding three to itself five times. Three plus three is six, plus three is nine, plus three is 12, and finally plus three is 15. Generalizing this, if we want to implement the idea of squaring a number, call it  $n$ , we would add  $n$  to itself  $n$  times.

Do this by hand first and try to isolate exactly what steps you take. You'll find you need to keep some "running total" of the sum so far, either on a piece of paper, or in your head. Remembering things from one step to the next is precisely why we have variables in a program. This means that we will need some variable to remember the "running total". It should be initialized with a value of zero. Then, we need to **update** the "running total" the correct number of times. For each repetition, we'll want to update the running total by adding the number to it.

In words we could say it this way. To square the value of  $n$ , we will repeat the process of updating a running total  $n$  times. To update the running total, we take the old value of the "running total" and add  $n$ . That sum becomes the new value of the "running total".

Here is the program in activecode. Note that the heading of the function definition is the same as it was before. All that has changed is the details of how the squaring is done. This is a great example of "black box" design. We can change out the details inside of the box and still use the function exactly as we did before.

```
def square(x):
    runningtotal = 0
    for counter in range(x):
        runningtotal = runningtotal + x

    return runningtotal

toSquare = 10
squareResult = square(toSquare)
print("The result of", toSquare, "squared is", squareResult)
```

In the program above, notice that the variable `runningtotal` starts out with a value of 0. Next, the iteration is performed  $x$  times. Inside the for loop, the update occurs. `runningtotal` is reassigned a new value which is the old value plus the value of  $x$ .

This pattern of iterating the updating of a variable is commonly referred to as the **accumulator pattern**. We refer to the variable as the **accumulator**. This pattern will come up over and over again. Remember that the key to making it work successfully is to be sure to initialize the variable before you start the iteration. Once inside the iteration, it is required that you update the accumulator.

**Note 6.5.1** What would happen if we put the assignment `runningTotal = 0` inside the for statement? Not sure? Try it and find out.

Here is the same program in codelens. Step through the function and watch the "running total" accumulate the result.

```
def square(x):
    runningtotal = 0
    for counter in range(x):
        runningtotal = runningtotal + x
```

```

    return runningtotal

toSquare = 10
squareResult = square(toSquare)
print("The result of", toSquare, "squared is", squareResult)

```

### 6.5.2 The General Accumulator Pattern

```

initialize the accumulator variable
repeat:
    modify the accumulator variable

# when the loop terminates the accumulator has the correct
    value

```

**Note 6.5.2** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

#### Check your understanding

**Checkpoint 6.5.3** Consider the following code:

```

def square(x):
    for counter in range(x):
        runningtotal = 0
        runningtotal = runningtotal + x
    return runningtotal

```

What happens if you put the initialization of runningtotal (the line runningtotal = 0) inside the for loop as the first instruction in the loop?

- A. The square function will return x instead of  $x * x$
- B. The square function will cause an error
- C. The square function will work as expected and return  $x * x$
- D. The square function will return 0 instead of  $x * x$

**Checkpoint 6.5.4** Rearrange the code statements so that the program will add up the first n odd numbers where n is provided by the user.

- $\text{thesum} = \text{thesum} + \text{oddnumber}$   
 $\text{oddnumber} = \text{oddnumber} + 2$
- `for counter in range(n):`
- `print(thesum)`
- `n = int(input('How many odd numbers would you like to add together?'))`  
`thesum = 0`  
`oddnumber = 1`

### 6.5.3 A Variation on the Accumulator Pattern

```

def square(x):
    '''raise x to the second power'''
    runningtotal = 0
    for counter in range(x):
        runningtotal = runningtotal + x

```

```

        return runningtotal

    toSquare = 10
    squareResult = square(toSquare)
    print("The result of", toSquare, "squared is", squareResult)

```

**Note 6.5.5 Modify the program ....** Change the value of `toSquare` in line 9 to `-10` and run.

We now see that our function has a semantic error. Remember when we first introduced the `square` function, unit testing and equivalence classes?

Change the value of `toSquare` in line 9 back to `10` and run.

What would happen if we change `runningtotal = runningtotal + x` to use **multiplication** instead of addition? Make this change to the program and look at the output.

It is very important to properly initialize the accumulator variable. Do a web search on **additive identity** and **multiplicative identity**. **Properly initialize the accumulator variable** and run the program.

Now we get an answer other than 0. However, the answer is not the square of `x`. It is also important that the loop repeat the proper number of times. How many times do we need to execute line 5 to get the square of `x`? **Change line 4 to repeat the correct number of times.** Now the program should produce the correct result.

Change the value of `toSquare` in line 9 to `-10` and run. Now negative inputs also work!

Remember that the boundary between our equivalence classes is 0. Try that value for `toSquare` also.

## 6.6 Functions can Call Other Functions

It is important to understand that each of the functions we write can be used and called from other functions we write. This is one of the most important ways that computer scientists take a large problem and break it down into a group of smaller problems. This process of breaking a problem into smaller subproblems is called **functional decomposition**.

Here's a simple example of functional decomposition using two functions. The first function called `square` simply computes the square of a given number. The second function called `sum_of_squares` makes use of `square` to compute the sum of three numbers that have been squared.

```

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

def sum_of_squares(x, y, z):
    a = square(x)
    b = square(y)
    c = square(z)

    return a + b + c

a = -5
b = 2
c = 10
result = sum_of_squares(a, b, c)
print(result)

```

Even though this is a pretty simple idea, in practice this example illustrates many very important Python concepts, including local and global variables along with parameter passing. Note that when you step through this example, codelens bolds line 1 and line 5 as the functions are defined. The body of `square` is not executed until it is called from the `sum_of_squares` function for the first time on line 6. Also notice that when `square` is called there are two groups of local variables, one for `square` and one for `sum_of_squares`. As you step through you will notice that `x`, and `y` are local variables in both functions and may even have different values. This illustrates that even though they are named the same, they are in fact, very different.

Now we will look at another example that uses two functions. This example illustrates an important computer science problem solving technique called **generalization**. Assume we want to write a function to draw a square. The generalization step is to realize that a square is just a special kind of rectangle.

To draw a rectangle we need to be able to call a function with different arguments for width and height. Unlike the case of the square, we cannot repeat the same thing 4 times, because the four sides are not equal. However, it is the case that drawing the bottom and right sides are the same sequence as drawing the top and left sides. So we eventually come up with this rather nice code that can draw a rectangle.

```
def drawRectangle(t, w, h):
    """Get turtle t to draw a rectangle of width w and height h."""
    for i in range(2):
        t.forward(w)
        t.left(90)
        t.forward(h)
        t.left(90)
```

The parameter names are chosen as single letters for conciseness. In real programs, we will insist on better variable names than this. The point is that the program doesn't "understand" that you're drawing a rectangle or that the parameters represent the width and the height. Concepts like rectangle, width, and height are meaningful for humans. They are not concepts that the program or the computer understands.

*Thinking like a computer scientist* involves looking for patterns and relationships. In the code above, we've done that to some extent. We did not just draw four sides. Instead, we spotted that we could draw the rectangle as two halves and used a loop to repeat that pattern twice.

But now we might spot that a square is a special kind of rectangle. A square simply uses the same value for both the height and the width. We already have a function that draws a rectangle, so we can use that to draw our square.

```
def drawSquare(tx, sz):          # a new version of drawSquare
    drawRectangle(tx, sz, sz)
```

Here is the entire example with the necessary set up code.

```
import turtle

def drawRectangle(t, w, h):
    """Get turtle t to draw a rectangle of width w and height h."""
    for i in range(2):
        t.forward(w)
        t.left(90)
```

```

        t.forward(h)
        t.left(90)

def drawSquare(tx, sz):      # a new version of drawSquare
    drawRectangle(tx, sz, sz)

wn = turtle.Screen()          # Set up the window
wn.bgcolor("lightgreen")

tess = turtle.Turtle()         # create tess

drawSquare(tess, 50)

wn.exitonclick()

```

There are some points worth noting here:

- Functions can call other functions.
- Rewriting `drawSquare` like this captures the relationship that we've spotted.
- A caller of this function might say `drawSquare(tess, 50)`. The parameters of this function, `tx` and `sz`, are assigned the values of the `tess` object, and the integer `50` respectively.
- In the body of the function, `tz` and `sz` are just like any other variable.
- When the call is made to `drawRectangle`, the values in variables `tx` and `sz` are fetched first, then the call happens. So as we enter the top of function `drawRectangle`, its variable `t` is assigned the `tess` object, and `w` and `h` in that function are both given the value `50`.

So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

- 1 Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command. The function (including its name) can capture your mental chunking, or *abstraction*, of the problem.
- 2 Creating a new function can make a program smaller by eliminating repetitive code.
- 3 Sometimes you can write functions that allow you to solve a specific problem using a more general solution.

#### Note 6.6.1 Lab.

- Drawing a Circle In this guided lab exercise [Section 21.3](#) we will work through a simple problem solving exercise related to drawing a circle with the turtle.

## 6.7 Flow of Execution Summary

When you are working with functions it is really important to know the order in which statements are executed. This is called the **flow of execution** and we've already talked about it a number of times in this chapter.

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. Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called 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 adept 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, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the `def` statements as you are scanning from top to bottom, but you should skip the body of the function until you reach a point where that function is called.

### Check your understanding

**Checkpoint 6.7.1** Consider the following Python code. Note that line numbers are included on the left.

```
def pow(b, p):
    y = b ** p
    return y

def square(x):
    a = pow(x, 2)
    return a

n = 5
result = square(n)
print(result)
```

Which of the following best reflects the order in which these lines of code are processed in Python?

- A. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
- B. 1, 2, 3, 5, 6, 7, 9, 10, 11
- C. 9, 10, 11, 1, 2, 3, 5, 6, 7
- D. 9, 10, 5, 6, 1, 2, 3, 6, 7, 10, 11
- E. 1, 5, 9, 10, 5, 6, 1, 2, 3, 6, 7, 10, 11

**Checkpoint 6.7.2** Consider the following Python code. Note that line numbers are included on the left.

```

def pow(b, p):
    y = b ** p
    return y

def square(x):
    a = pow(x, 2)
    return a

n = 5
result = square(n)
print(result)

```

What does this function print?

- A. 25
- B. 5
- C. 125
- D. 32

## 6.8 Using a Main Function

Using functions is a good idea. It helps us to modularize our code by breaking a program into logical parts where each part is responsible for a specific task. For example, in one of our first programs there was a function called `drawSquare` that was responsible for having some turtle draw a square of some size. The actual turtle and the actual size of the square were defined to be provided as parameters. Here is that original program.

```

import turtle

def drawSquare(t, sz):
    """Make turtle draw a square of with side sz."""

    for i in range(4):
        t.forward(sz)
        t.left(90)

wn = turtle.Screen()          # Set up the window and its
                             # attributes
wn.bgcolor("lightgreen")

alex = turtle.Turtle()         # create alex
drawSquare(alex, 50)          # Call the function to draw
                             # the square

wn.exitonclick()

```

If you look closely at the structure of this program, you will notice that we first perform all of our necessary `import` statements, in this case to be able to use the `turtle` module. Next, we define the function `drawSquare`. At this point, we could have defined as many functions as were needed. Finally, there are five statements that set up the window, create the turtle, perform the function invocation, and wait for a user click to terminate the program.

These final five statements perform the main processing that the program will do. Notice that much of the detail has been pushed inside the `drawSquare`

function. However, there are still these five lines of code that are needed to get things done.

In many programming languages (e.g. Java and C++), it is not possible to simply have statements sitting alone like this at the bottom of the program. They are required to be part of a special function that is automatically invoked by the operating system when the program is executed. This special function is called **main**. Although this is not required by the Python programming language, it is actually a good idea that we can incorporate into the logical structure of our program. In other words, these five lines are logically related to one another in that they provide the main tasks that the program will perform. Since functions are designed to allow us to break up a program into logical pieces, it makes sense to call this piece **main**.

The following activecode shows this idea. In line 11 we have defined a new function called **main** that doesn't need any parameters. The five lines of main processing are now placed inside this function. Finally, in order to execute that main processing code, we need to invoke the **main** function (line 20). When you push run, you will see that the program works the same as it did before.

```
import turtle

def drawSquare(t, sz):
    """Make turtle t draw a square of side sz."""

    for i in range(4):
        t.forward(sz)
        t.left(90)

def main():                      # Define the main function
    wn = turtle.Screen()          # Set up the window and
        its attributes
    wn.bgcolor("lightgreen")

    alex = turtle.Turtle()         # create alex
    drawSquare(alex, 50)           # Call the function to
        draw the square

    wn.exitonclick()

main()                           # Invoke the main function
```

Now our program structure is as follows. First, import any modules that will be required. Second, define any functions that will be needed. Third, define a **main** function that will get the process started. And finally, invoke the **main** function (which will in turn call the other functions as needed).

**Note 6.8.1** In Python there is nothing special about the name **main**. We could have called this function anything we wanted. We chose **main** just to be consistent with some of the other languages.

### Advanced Topic

Before the Python interpreter executes your program, it defines a few special variables. One of those variables is called `__name__` and it is automatically set to the string value "`__main__`" when the program is being executed by itself in a standalone fashion. On the other hand, if the program is being imported by another program, then the `__name__` variable is set to the name of that module. This means that we can know whether the program is being run by itself or whether it is being used by another program and based on that observation,

we may or may not choose to execute some of the code that we have written.

For example, assume that we have written a collection of functions to do some simple math. We can include a `main` function to invoke these math functions. It is much more likely, however, that these functions will be imported by another program for some other purpose. In that case, we would not want to execute our `main` function.

The activecode below defines two simple functions and a `main`.

```
def squareit(n):
    return n * n

def cubeit(n):
    return n*n*n

def main():
    anum = int(input("Please enter a number"))
    print(squareit(anum))
    print(cubeit(anum))

if __name__ == "__main__":
    main()
```

Line 12 uses an `if` statement to ask about the value of the `__name__` variable. If the value is "`__main__`", then the `main` function will be called. Otherwise, it can be assumed that the program is being imported into another program and we do not want to call `main` because that program will invoke the functions as needed. This ability to conditionally execute our `main` function can be extremely useful when we are writing code that will potentially be used by others. It allows us to include functionality that the user of the code will not need, most often as part of a testing process to be sure that the functions are working correctly.

**Note 6.8.2** In order to conditionally execute the `main` function, we used a structure called an `if` statement to create what is known as selection. This topic will be studied in much more detail later.

## 6.9 Program Development

At this point, you should be able to look at complete functions and tell what they do. Also, if you have been doing the exercises, you have written some small functions. As you write larger functions, you might start to have more difficulty, especially with runtime and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$ . By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `distance` function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function that captures our thinking so far.

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated.

We import the test module to enable us to write a unit test for the function.

```
import test
def distance(x1, y1, x2, y2):
    return 0.0

test.testEqual(distance(1, 2, 1, 2), 0)
```

The `testEqual` function from the test module calls the `distance` function with sample inputs: (1,2, 1,2). The first 1,2 are the coordinates of the first point and the second 1,2 are the coordinates of the second point. What is the distance between these two points? Zero. `testEqual` compares what is returned by the `distance` function and the 0 (the correct answer).

**Note 6.9.1 Extend the program ....** On line 6, write another unit test. Use (1,2, 4,6) as the parameters to the `distance` function. How far apart are these two points? Use that value (instead of 0) as the correct answer for this unit test.

On line 7, write another unit test. Use (0,0, 1,1) as the parameters to the `distance` function. How far apart are these two points? Use that value as the correct answer for this unit test.

The first test passes but the others fail since the `distance` function does not yet contain all the necessary steps.

**When testing a function, it is essential to know the right answer.**

For the second test the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). For the third test, we have a 1-1-sqrt(2) triangle.

At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be — in the last line we added.

A logical first step in the computation is to find the differences  $x_2 - x_1$  and  $y_2 - y_1$ . We will store those values in temporary variables named `dx` and `dy`.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    return 0.0
```

Next we compute the sum of squares of `dx` and `dy`.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    return 0.0
```

Again, we could run the program at this stage and check the value of `dsquared` (which should be 25).

Finally, using the fractional exponent `0.5` to find the square root, we compute and return the result.

```
import test
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result

test.assertEqual(distance(1,2, 1,2), 0)
test.assertEqual(distance(1,2, 4,6), 5)
test.assertEqual(distance(0,0, 1,1), 1.41)
```

**Note 6.9.2 Fix the error ....** Two of the tests pass but the last one fails. Is there still an error in the function?

Frequently we discover errors in the functions that we are writing. However, it is possible that there is an error in a test. Here the error is in the precision of the correct answer.

The third test fails because by default `testEqual` checks 5 digits to the right of the decimal point.

- Change `1.41` to `1.41421` and run. The test will pass.

There are circumstances where 2 digits to the right of the decimal point is sufficiently precise.

- Copy line 11 on to line 12. On line 12, change `1.41421` to `1.41`. Run. The test fails.
- Type `, 2` after `1.41`. (The 2 represents the precision of the test – how many digits to the right of the decimal that must be correct.) Run.

Now all four the tests pass! Wonderful! However, you may still need to perform additional tests.

When you start out, you might add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger conceptual chunks. As you improve your programming skills you should find yourself managing bigger and bigger chunks: this is very similar to the way we learned to read letters, syllables, words, phrases, sentences, paragraphs, etc., or the way we learn to chunk music — from individual notes to chords, bars, phrases, and so on.

The key aspects of the process are:

- 1 Make sure you know what you are trying to accomplish. Then you can write appropriate unit tests.
- 2 Start with a working skeleton program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
- 3 Use temporary variables to hold intermediate values so that you can easily inspect and check them.
- 4 Once the program is working, you might want to consolidate multiple statements into compound expressions, but only do this if it does not make the program more difficult to read.

## 6.10 Composition

As we have already seen, you can call one function from within another. This ability to build functions by using other functions is called **composition**.

As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we've just written a function, `distance`, that does just that, so now all we have to do is use it:

```
| radius = distance(xc, yc, xp, yp)
```

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier functions:

```
| result = area(radius)
| return result
```

Wrapping that up in a function, we get:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result

def area(radius):
    b = 3.14159 * radius**2
    return b

def area2(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result

print(area2(0,0,1,1))
```

We called this function `area2` to distinguish it from the `area` function defined earlier. There can only be one function with a given name within a module.

Note that we could have written the composition without storing the intermediate results.

```
| def area2(xc, yc, xp, yp):
|     return area(distance(xc, yc, xp, yp))
```

## 6.11 A Turtle Bar Chart

Recall from our discussion of modules that there were a number of things that turtles can do. Here are a couple more tricks (remember that they are all described in the module documentation).

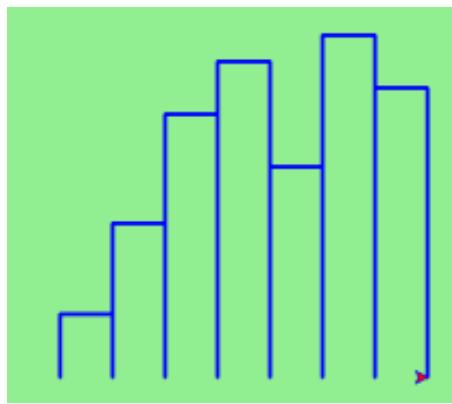
- We can get a turtle to display text on the canvas at the turtle's current position. The method is called `write`. For example, `alex.write("Hello")` would write the string hello at the current position.

- One can fill a shape (circle, semicircle, triangle, etc.) with a fill color. It is a two-step process. First you call the method `begin_fill`, for example `alex.begin_fill()`. Then you draw the shape. Finally, you call `end_fill` (`alex.end_fill()`).
- We've previously set the color of our turtle - we can now also set its fill color, which need not be the same as the turtle and the pen color. To do this, we use a method called `fillcolor`, for example, `alex.fillcolor("red")`.

Ok, so can we get tess to draw a bar chart? Let us start with some data to be charted,

```
xs = [48, 117, 200, 240, 160, 260, 220]
```

Corresponding to each data measurement, we'll draw a simple rectangle of that height, with a fixed width. Here is a simplified version of what we would like to create.



We can quickly see that drawing a bar will be similar to drawing a rectangle or a square. Since we will need to do it a number of times, it makes sense to create a function, `drawBar`, that will need a turtle and the height of the bar. We will assume that the width of the bar will be 40 units. Once we have the function, we can use a basic for loop to process the list of data values.

```
def drawBar(t, height):
    """Get turtle t to draw one bar, of height."""
    t.left(90)           # Point up
    t.forward(height)     # Draw up the left side
    t.right(90)
    t.forward(40)         # width of bar, along the top
    t.right(90)
    t.forward(height)     # And down again!
    t.left(90)           # put the turtle facing the
                         # way we found it.

    ...
for v in xs:                  # assume xs and tess are ready
    drawBar(tess, v)
```

It is a nice start! The important thing here was the mental chunking. To solve the problem we first broke it into smaller pieces. In particular, our chunk is to draw one bar. We then implemented that chunk with a function. Then, for the whole chart, we repeatedly called our function.

Next, at the top of each bar, we'll print the value of the data. We will do this in the body of `drawBar` by adding `t.write(str(height))` as the new fourth line of the body. Note that we had to turn the number into a string. Finally, we'll add the two methods needed to fill each bar.

The one remaining problem is related the fact that our turtle lives in a world where position (0,0) is at the center of the drawing canvas. In this problem, it would help if (0,0) were in the lower left hand corner. To solve this we can use our `setworldcoordinates` method to rescale the window. While we are at it, we should make the window fit the data. The tallest bar will correspond to the maximum data value. The width of the window will need to be proportional to the number of bars (the number of data values) where each has a width of 40. Using this information, we can compute the coordinate system that makes sense for the data set. To make it look nice, we'll add a 10 unit border around the bars.

Here is the complete program. Try it and then change the data to see that it can adapt to the new values. Note also that we have stored the data values in a list and used a few list functions. We will have much more to say about lists in a later chapter.

```

import turtle

def drawBar(t, height):
    """Get turtle t to draw one bar, of height."""
    t.begin_fill()                      # start filling this shape
    t.left(90)
    t.forward(height)
    t.write(str(height))
    t.right(90)
    t.forward(40)
    t.right(90)
    t.forward(height)
    t.left(90)
    t.end_fill()                       # stop filling this shape


xs = [48, 117, 200, 240, 160, 260, 220]  # here is the data
maxheight = max(xs)
numbars = len(xs)
border = 10

wn = turtle.Screen()                  # Set up the window and
    its attributes
wn.setworldcoordinates(0-border, 0-border,
    40*numbars+border, maxheight+border)
wn.bgcolor("lightgreen")

tess = turtle.Turtle()                # create tess and set some
    attributes
tess.color("blue")
tess.fillcolor("red")
tess.pensize(3)

for a in xs:
    drawBar(tess, a)

wn.exitonclick()

```

This code is quite concise, but each height label is partly covered by the top segment of its bar. Can you modify the `drawBar` code, moving the label

up slightly but not changing the bar? Hint: The label cannot be drawn during the polygon fill sequence.

**Note 6.11.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 6.12 Glossary

### Glossary

**chatterbox function.** A function which interacts with the user (using `input` or `print`) when it should not. Silent functions that just convert their input arguments into their output results are usually the most useful ones.

**composition (of functions).** Calling one function from within the body of another, or using the return value of one function as an argument to the call of another.

**dead code.** Part of a program that can never be executed, often because it appears after a `return` statement.

**fruitful function.** A function that yields a return value instead of `None`.

**incremental development.** A program development plan intended to simplify debugging by adding and testing only a small amount of code at a time.

**None.** A special Python value. One use in Python is that it is returned by functions that do not execute a `return` statement with a return argument.

**return value.** The value provided as the result of a function call.

**scaffolding.** Code that is used during program development to assist with development and debugging. The unit test code that we added in this chapter are examples of scaffolding.

**temporary variable.** A variable used to store an intermediate value in a complex calculation.

## 6.13 Exercises

1. Use the `drawsquare` function we wrote in this chapter in a program to draw the image shown below. Assume each side is 20 units. (Hint: notice that the turtle has already moved away from the ending point of the last square when the program ends.)



```

import turtle

def drawSquare(t, sz):
    """Get turtle t to draw a square of sz side"""

    for i in range(4):
        t.forward(sz)
        t.left(90)

wn = turtle.Screen()
wn.bgcolor("lightgreen")

alex = turtle.Turtle()
alex.color("pink")

drawSquare(alex, 20)

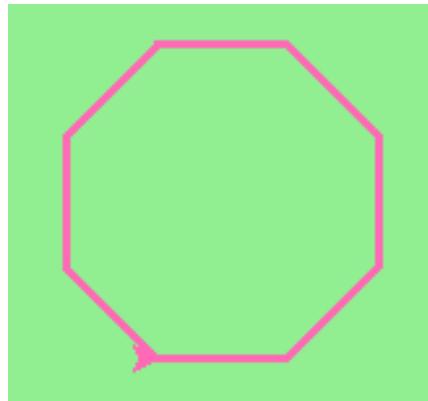
wn.exitonclick()

```

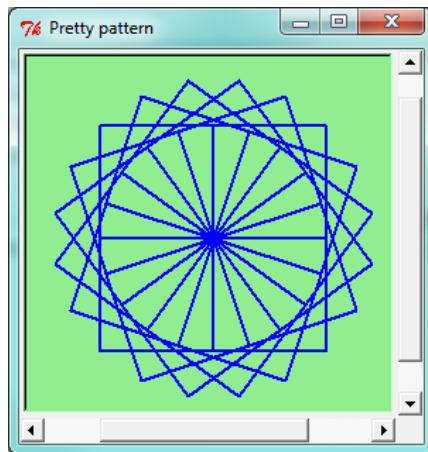
2. Write a program to draw this. Assume the innermost square is 20 units per side, and each successive square is 20 units bigger, per side, than the one inside it.



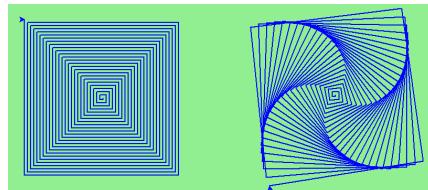
3. Write a non-fruitful function `drawPoly(someturtle, somesides, somesize)` which makes a turtle draw a regular polygon. When called with `drawPoly(tess, 8, 50)`, it will draw a shape like this:



4. Draw this pretty pattern.



5. The two spirals in this picture differ only by the turn angle. Draw both.



6. Write a non-fruitful function `drawEquitriangle(someturtle, somesize)` which calls `drawPoly` from the previous question to have its turtle draw a equilateral triangle.
7. Write a fruitful function `sumTo(n)` that returns the sum of all integer numbers up to and including n. So `sumTo(10)` would be  $1+2+3\dots+10$  which would return the value 55. Use the equation  $(n * (n + 1)) / 2$ .

```
def sumTo(n):
    # your code here ====
    from unittest.gui import TestCaseGui

    class myTests(TestCaseGui):

        def testOne(self):
            self.assertAlmostEqual(sumTo(15), 120.0, 0, "Tested\u
                sumTo\uon\uinput\u15")
            self.assertAlmostEqual(sumTo(0), 0.0, 0, "Tested\u
                sumTo\uon\uinput\u0")
            self.assertAlmostEqual(sumTo(25), 325.0, 0, "Tested\u
                sumTo\uon\uinput\u25")
            self.assertAlmostEqual(sumTo(7), 28.0, 0, "Tested\u
                sumTo\uon\uinput\u7")

    myTests().main()
```

8. Write a function `areaOfCircle(r)` which returns the area of a circle of radius r. Make sure you use the `math` module in your solution.

```

def areaOfCircle(r):
    # your code here ====
from unittest.gui import TestCaseGui

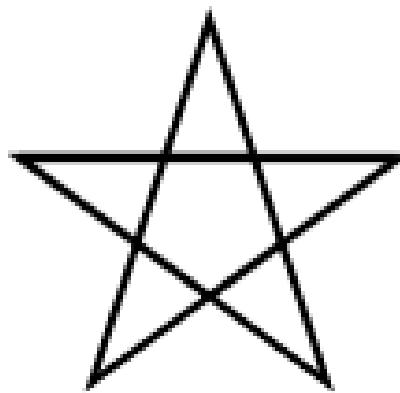
class myTests(TestCaseGui):

    def testOne(self):
        self.assertAlmostEqual(areaOfCircle(5.0), 78.53981633974483, 5, "Tested"
                           input:areaOfCircle(5.0)")
        self.assertEqual(areaOfCircle(5.0), 78.53981633974483, "Tested"
                           input:areaOfCirlce(5.0)")
        self.assertEqual(areaOfCircle(0), 0.0, "Tested"
                           input:areaOfCirlce(0))")
        self.assertAlmostEqual(areaOfCircle(31415.926535897932), 3100627668.0299816
                           input:areaOfCirlce(31415.926535897932)")

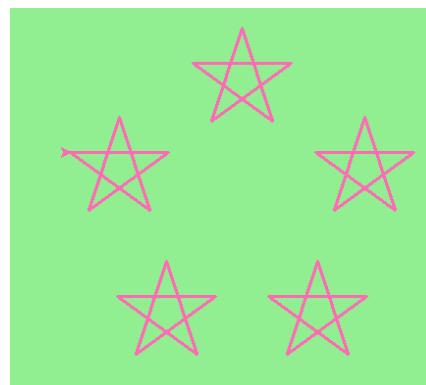
myTests().main()

```

9. Write a non-fruitful function to draw a five pointed star, where the length of each side is 100 units.



10. Extend your program above. Draw five stars, but between each, pick up the pen, move forward by 350 units, turn right by 144, put the pen down, and draw the next star. You'll get something like this (note that you will need to move to the left before drawing your first star in order to fit everything in the window):



What would it look like if you didn't pick up the pen?

11. Extend the star function to draw an n pointed star. (Hint: n must be an odd number greater or equal to 3).

12. Write a function called `drawSprite` that will draw a sprite. The function will need parameters for the turtle, the number of legs, and the length of the legs. Invoke the function to create a sprite with 15 legs of length 120.
13. Rewrite the function `sumTo(n)` that returns the sum of all integer numbers up to and including n. This time use the accumulator pattern.

```
def sumTo(n):
    # your code here ====
from unittest.gui import TestCaseGui
class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(sumTo(15),120,"Tested\u2022sumTo\u2022on\u
            input\u202215")
        self.assertEqual(sumTo(0),0,"Tested\u2022sumTo\u2022on\u
            input\u20220")
        self.assertEqual(sumTo(25),325,"Tested\u2022sumTo\u2022on\u
            input\u202225")
        self.assertEqual(sumTo(7),28,"Tested\u2022sumTo\u2022on\u
            input\u20227")

myTests().main()
```

14. Write a function called `mySqrt` that will approximate the square root of a number, call it n, by using Newton's algorithm. Newton's approach is an iterative guessing algorithm where the initial guess is  $n/2$  and each subsequent guess is computed using the formula:  $\text{newguess} = (1/2) * (\text{oldguess} + (n/\text{oldguess}))$ .

```
def mySqrt(n):
    # your code here ====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        self.assertAlmostEqual(mySqrt(4.0),2.0,0,"Tested\u
            mySqrt\u2022on\u2022input\u20224.0")
        self.assertAlmostEqual(mySqrt(9.0),3.0,4,"Tested\u
            accuracy\u2022of\u2022mySqrt\u2022on\u2022input\u20223.0")
        self.assertAlmostEqual(mySqrt(36.0),6.0,5,"Tested\u
            accuracy\u2022of\u2022mySqrt\u2022on\u2022input\u20226.0")
        self.assertAlmostEqual(mySqrt(100.0),10.0,4,"Tested\u
            accuracy\u2022of\u2022mySqrt\u2022on\u2022input\u202210.0.\u2022Try\u
            iterating\u2022more\u2022times.")

myTests().main()
```

15. Write a function called `myPi` that will return an approximation of PI ( $3.14159\dots$ ). Use the [Leibniz](#)<sup>1</sup> approximation.

```
def myPi(iters):
    # Calculate an approximation of PI using the Leibniz
    # approximation with iters number of iterations

    # your code here
```

16. Write a function called `myPi` that will return an approximation of PI ( $3.14159\dots$ ). Use the [Madhava](#)<sup>2</sup> approximation.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Leibniz\\_formula\\_for\\_\pi](http://en.wikipedia.org/wiki/Leibniz_formula_for_\pi)

```
def myPi(iters):
    # Calculate an approximation of PI using the Madhava
    # approximation with iters number of iterations

    #your code here
```

17. Write a function called fancySquare that will draw a square with fancy corners (sprites on the corners). You should implement and use the drawSprite function from above. For an even more interesting look, how about adding small triangles to the ends of the sprite legs.
18. There was a whole program in [Section 6.11](#) to create a bar chart with specific data. Creating a bar chart is a useful idea in general. Write a non-fruitful function called barChart, that takes the numeric list of data as a parameter, and draws the bar chart. Write a full program calling this function. The current version of the drawBar function unfortunately draws the top of the bar through the bottom of the label. A nice elaboration is to make the label appear completely above the top line. To keep the spacing consistent you might pass an extra parameter to drawBar for the distance to move up. For the barChart function make that parameter be some small fraction of maxheight+border. The fill action makes this modification particularly tricky: You will want to move past the top of the bar and write before or after drawing and filling the bar.

---

<sup>2</sup>[http://en.wikipedia.org/wiki/Madhava\\_of\\_Sangamagrama](http://en.wikipedia.org/wiki/Madhava_of_Sangamagrama)

```
import turtle

def drawBar(t, height):
    """Get turtle t to draw one bar, of height."""
    t.begin_fill()                      # start filling this
        shape
    t.left(90)
    t.forward(height)
    t.write(str(height))
    t.right(90)
    t.forward(40)
    t.right(90)
    t.forward(height)
    t.left(90)
    t.end_fill()                       # stop filling this
        shape

xs = [48, 117, 200, 240, 160, 260, 220] # here is the
    data
maxheight = max(xs)
numbars = len(xs)
border = 10

wn = turtle.Screen()                  # Set up the window
    and its attributes
wn.setworldcoordinates(0-border, 0-border,
    40*numbars+border, maxheight+border)
wn.bgcolor("lightgreen")

tess = turtle.Turtle()                # create tess and set
    some attributes
tess.color("blue")
tess.fillcolor("red")
tess.pensize(3)

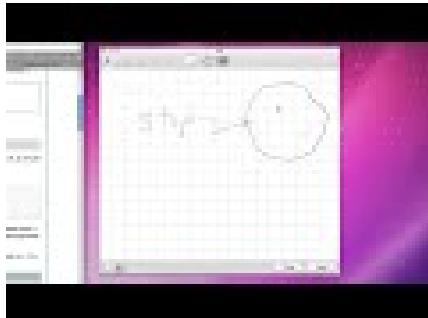
for a in xs:
    drawBar(tess, a)

wn.exitonclick()
```

# Chapter 7

## Selection

### 7.1 Boolean Values and Boolean Expressions



The Python type for storing true and false values is called `bool`, named after the British mathematician, George Boole. George Boole created *Boolean Algebra*, which is the basis of all modern computer arithmetic.

There are only two **boolean values**. They are `True` and `False`. Capitalization is important, since `true` and `false` are not boolean values (remember Python is case sensitive).

```
print(True)
print(type(True))
print(type(False))
```

**Note 7.1.1** Boolean values are not strings!

It is extremely important to realize that `True` and `False` are not strings. They are not surrounded by quotes. They are the only two values in the data type `bool`. Take a close look at the types shown below.

```
print(type(True))
print(type("True"))
```

A **boolean expression** is an expression that evaluates to a boolean value. The equality operator, `==`, compares two values and produces a boolean value related to whether the two values are equal to one another.

```
print(5 == 5)

print(5 == 6)

j = "hel"
print(j + "lo" == "hello")
```

In the first statement, the two operands are equal, so the expression evaluates to `True`. In the second statement, 5 is not equal to 6, so we get `False`.

The `==` operator is one of six common **comparison operators**; the others are:

<code>x != y</code>	<code># x is not equal to y</code>
<code>x &gt; y</code>	<code># x is greater than y</code>
<code>x &lt; y</code>	<code># x is less than y</code>
<code>x &gt;= y</code>	<code># x is greater than or equal to y</code>
<code>x &lt;= y</code>	<code># x is less than or equal to y</code>

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 comparison operator. Also, there is no such thing as `=<` or `=>`.

Note too that an equality test is symmetric, but assignment is not. For example, if `a == 7` then `7 == a`. But in Python, the statement `a = 7` is legal and `7 = a` is not. (Can you explain why?)

#### Check your understanding

**Checkpoint 7.1.2** Which of the following is a Boolean expression? Select all that apply.

- A. True
- B. `3 == 4`
- C. `3 + 4`
- D. `3 + 4 == 7`
- E. "False"

## 7.2 Logical operators

### 7.2.1 Introduction

There are three **logical operators**: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example, `x > 0` and `x < 10` is true only if `x` is greater than 0 *and* at the same time, `x` is less than 10. How would you describe this in words? You would say that `x` is between 0 and 10, not including the endpoints.

`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* divisible by 3. In this case, one, or the other, or both of the parts has to be true for the result to be true.

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`.

<code>x = 5</code>	
<code>print(x &gt; 0 and x &lt; 10)</code>	
<code>n = 25</code>	
<code>print(n % 2 == 0 or n % 3 == 0)</code>	

When trying to show how logical operators work, computer scientists and mathematicians alike will use **truth tables**. A truth table is a small table that lists all possible inputs on its left columns and then will display the output of

its particular logical operator in the right column. Take the logical operator and for example:

**Table 7.2.1**

a	b	a and b
T	T	T
T	F	F
F	T	F
F	F	F

The *T* in the table stands for **True** while the *F* stands for **False**. Notice that when **a** and **b** are both **True**, the logical operator **and** outputs **True**. This is exactly how we normally use “and” in everyday conversation. Here are the rest of the operators:

**Table 7.2.2**

a	b	a and b	a or b	not a	not b
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Also, Google has provided this short video showing different logical operators:



In the video, each letter is representative of a logical operator and only shows its color when the corresponding **x** or **y** is showing in the second **G**. If you take a look at xor, you will notice it is only colorful when either **x** or **y** is showing, but not both. This is called **exclusive or**, which we will not be using.

**Note 7.2.3 WARNING!** There is a very common mistake that occurs when programmers try to write boolean expressions. For example, what if we have a variable **number** and we want to check to see if its value is 5,6, or 7. In words we might say: “number equal to 5 or 6 or 7”. However, if we translate this into Python, `number == 5 or 6 or 7`, it will not be correct. The **or** operator must join the results of three equality checks. The correct way to write this is `number == 5 or number == 6 or number == 7`. This may seem like a lot of typing but it is absolutely necessary. You cannot take a shortcut.

#### Check your understanding

**Checkpoint 7.2.4** What is a correct Python expression for checking to see if a number stored in a variable **x** is between 0 and 5?

- A.  $x > 0$  and  $< 5$
- B.  $x > 0$  or  $x < 5$
- C.  $x > 0$  and  $x < 5$

**Checkpoint 7.2.5** Say you are registering for next semester's classes. You have choice A, which is your art class, and choice B, which is your math class. You need both of them, but it's a race between time and luck. If you end up registering on time for choice A, but you don't get your choice B, which logical operators would be true?

- A. A and B
- B. A or B
- C. not A
- D. not B

### 7.2.2 Logical Opposites

Each of the six relational operators has a logical opposite: for example, suppose we can get a driving licence when our age is greater or equal to 17, we can *not* get the driving licence when we are less than 17.

**Table 7.2.6**

Operator	Definiton	Logical Opposites
<code>==</code>	Equals to	<code>!=</code>
<code>!=</code>	Not Equals to	<code>==</code>
<code>&lt;</code>	Less than	<code>&gt;=</code>
<code>&lt;=</code>	Less Than or Equal to	<code>&gt;</code>
<code>&gt;</code>	Greater Than	<code>&lt;=</code>
<code>&gt;=</code>	Greater Than or Equal to	<code>&lt;</code>

Understanding these logical opposites allows us to sometimes get rid of `not` operators. `not` operators are often quite difficult to read in computer code, and our intentions will usually be clearer if we can eliminate them.

For example, if we wrote this Python:

```
| if not (age >= 17):
|     print("Hey, you're too young to get a driving licence!")
```

it would probably be clearer to use the simplification laws, and to write instead:

```
| if age < 17:
|     print("Hey, you're too young to get a driving licence!")
```

Two powerful simplification laws (called de Morgan's laws) that are often helpful when dealing with complicated Boolean expressions are:

```
| not (x and y) == (not x) or (not y)
| not (x or y) == (not x) and (not y)
```

For example, suppose you want to update your phone; however, your phone will only update if it has at least 50% battery life and 15% of its storage available. As we look at the Python code for this, we see:

```
| if not ((phone_charge >= 0.50) and (phone_storage >= .15)):
|     print("You cannot restart your phone. Battery too low or")
```

```

        notEnoughFreeSpace.")
else:
    print("Updating now... Several restarts may be required.")

```

Applying rules of logical opposites would let us rework the condition in a (perhaps) easier to understand way like this:

```

if (phone_charge < 0.50) or (phone_storage < .15):
    print("You cannot restart your phone. Battery too low or"
        notEnoughFreeSpace.")
else:
    print("Updating now... Several restarts may be required.")

```

We could also get rid of the `not` by swapping around the `then` and `else` parts of the conditional. So here is a third version, also equivalent:

```

if (phone_charge >= 0.50) and (phone_storage >= .15):
    print("Updating now... Several restarts may be required.")
else:
    print("You cannot restart your phone. Battery too low or"
        notEnoughFreeSpace.")

```

This version is probably the best of the three, because it very closely matches the initial English statement. Clarity of our code (for other humans), and making it easy to see that the code does what was expected should always be a high priority.

As our programming skills develop we'll find we have more than one way to solve any problem. So good programs are *designed*. We make choices that favour clarity, simplicity, and elegance. The job title *software architect* says a lot about what we do — we are *architects* who engineer our products to balance beauty, functionality, simplicity and clarity in our creations.

## 7.3 Precedence of Operators

We have now added a number of additional operators to those we learned in the previous chapters. It is important to understand how these operators relate to the others with respect to operator precedence. Python will always evaluate the arithmetic operators first (`**` is highest, then multiplication/division, then addition/subtraction). Next comes the relational operators. Finally, the logical operators are done last. This means that the expression `x*5 >= 10 and y-6 <= 20` will be evaluated so as to first perform the arithmetic and then check the relationships. The `and` will be done last. Although many programmers might place parenthesis around the two relational expressions, it is not necessary.

The following table summarizes the precedence discussed so far from highest to lowest. See [Section 21.11](#) for *all* the operators introduced in this book.

**Table 7.3.1**

Level	Category	Operators
7(high)	exponent	<code>**</code>
6	multiplication	<code>*, /, //, %</code>
5	addition	<code>+, -</code>
4	relational	<code>==, !=, &lt;=, &gt;=, &gt;, &lt;</code>
3	logical	<code>not</code>
2	logical	<code>and</code>
1(low)	logical	<code>or</code>

**Note 7.3.2** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

#### Check your understanding

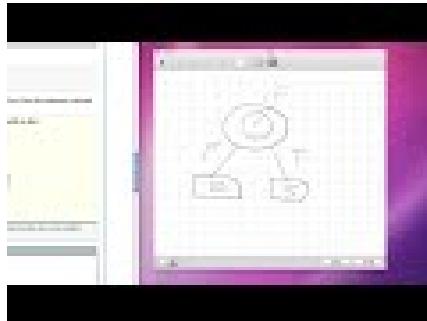
**Checkpoint 7.3.3** Which of the following properly expresses the precedence of operators (using parentheses) in the following expression:  $5*3 > 10$  and  $4+6==11$

- A.  $((5*3) > 10) \text{ and } ((4+6) == 11)$
- B.  $(5*(3 > 10)) \text{ and } (4 + (6 == 11))$
- C.  $((((5*3) > 10) \text{ and } 4)+6) == 11$
- D.  $((5*3) > (10 \text{ and } (4+6))) == 11$

Here is an animation for the above expression:

**Checkpoint 7.3.4** An interactive Runestone problem goes here, but there is not yet a static representation.

## 7.4 Conditional Execution: Binary Selection



In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Selection statements**, sometimes also referred to as **conditional statements**, give us this ability. The simplest form of selection is the **if statement**. This is sometimes referred to as **binary selection** since there are two possible paths of execution.

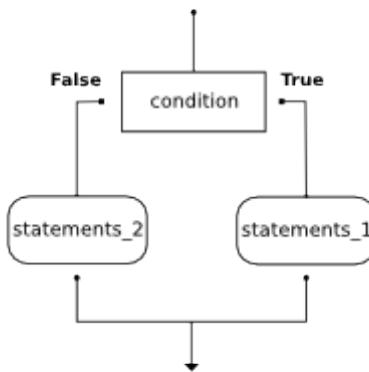
```
x = 15

if x % 2 == 0:
    print(x, "is even")
else:
    print(x, "is odd")
```

The syntax for an if statement looks like this:

```
if BOOLEAN EXPRESSION:
    STATEMENTS_1          # executed if condition evaluates
    to True
else:
    STATEMENTS_2          # executed if condition evaluates
    to False
```

The boolean expression after the if statement is called the **condition**. If it is true, then the immediately following indented statements get executed. If not, then the statements indented under the else clause get executed.



As with the function definition from the last chapter and other compound statements like `for`, the `if` statement consists of a header line and a body. The header line begins with the keyword `if` followed by a *boolean expression* and ends with a colon (:).

The more indented statements that follow are called a **block**.

Each of the statements inside the first block of statements is executed in order if the boolean expression evaluates to `True`. The entire first block of statements is skipped if the boolean expression evaluates to `False`, and instead all the statements under the `else` clause are executed.

There is no limit on the number of statements that can appear under the two clauses of an `if` statement, but there has to be at least one statement in each block.

Each compound statement includes a heading and all the following further-indented statements in the block after the heading. The `if - else` statement is an unusual compound statement because it has more than one part at the same level of indentation as the `if` heading, (the `else` clause, with its own indented block).

#### Note 7.4.1 Lab.

- Approximating Pi with Simulation [Section 21.7](#) In this guided lab exercise we will work through a problem solving exercise related to approximating the value of pi using random numbers.

#### Check your understanding

**Checkpoint 7.4.2** How many statements can appear in each block (the `if` and the `else`) in a conditional statement?

- Just one.
- Zero or more.
- One or more.
- One or more, and each must contain the same number.

**Checkpoint 7.4.3** What does the following code print (choose from output a, b, c or nothing)?

```
if 4 + 5 == 10:  
    print("TRUE")  
else:  
    print("FALSE")
```

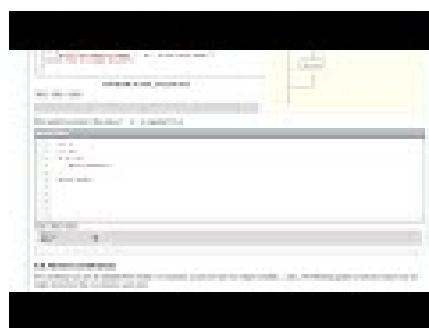
- A. TRUE
- B. FALSE
- C. TRUE on one line and FALSE on the next
- D. Nothing will be printed

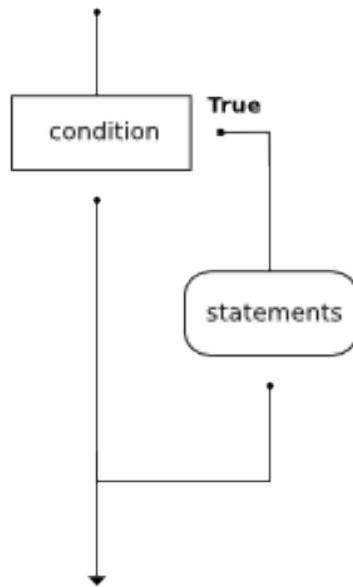
**Checkpoint 7.4.4** What does the following code print?

```
if 4 + 5 == 10:  
    print("TRUE")  
else:  
    print("FALSE")  
print("TRUE")
```

- a. TRUE
  - b.
    - TRUE
    - FALSE
  - c.
    - FALSE
    - TRUE
  - d.
    - TRUE
    - FALSE
    - TRUE
- A. Output a
  - B. Output b
  - C. Output c
  - D. Output d

## 7.5 Omitting the else Clause: Unary Selection





Another form of the `if` statement is one in which the `else` clause is omitted entirely. This creates what is sometimes called **unary selection**. In this case, when the condition evaluates to `True`, the statements are executed. Otherwise the flow of execution continues to the statement after the body of the `if`.

```

x = 10
if x < 0:
    print("The negative number", x, "is not valid here.")
print("This is always printed")

```

What would be printed if the value of `x` is negative? Try it.

#### Check your understanding

**Checkpoint 7.5.1** What does the following code print?

```

x = -10
if x < 0:
    print("The negative number", x, "is not valid here.")
print("This is always printed")

```

a.

This is always printed

b.

The negative number -10 is not valid here  
This is always printed

c.

The negative number -10 is not valid here

A. Output a

B. Output b

C. Output c

D. It will cause an error because every if must have an else clause.

**Checkpoint 7.5.2** Will the following code cause an error?

```

x = -10
if x < 0:
    print("The negative number", x, "is not valid here.")
else:
    print(x, "is a positive number")
else:
    print("This is always printed")

```

A. No

B. Yes

## 7.6 Nested conditionals

One conditional can also be **nested** within another. For example, assume we have two integer variables, *x* and *y*. The following pattern of selection shows how we might decide how they are related to each other.

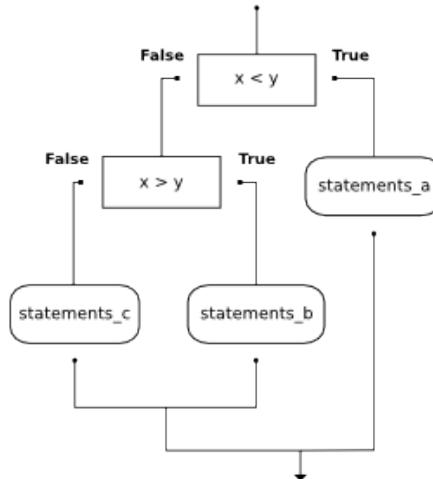
```

if x < y:
    print("x is less than y")
else:
    if x > y:
        print("x is greater than y")
    else:
        print("x and y must be equal")

```

The outer conditional contains two branches. The second branch (the else from the outer) contains another if statement, which has two branches of its own. Those two branches could contain conditional statements as well.

The flow of control for this example can be seen in this flowchart illustration.



Here is a complete program that defines values for *x* and *y*. Run the program and see the result. Then change the values of the variables to change the flow of control.

```

x = 10
y = 10

if x < y:
    print("x is less than y")

```

```

    else:
        if x > y:
            print("x is greater than y")
        else:
            print("x and y must be equal")

```

**Note 7.6.1** In some programming languages, matching the if and the else is a problem. However, in Python this is not the case. The indentation pattern tells us exactly which else belongs to which if.

If you are still a bit unsure, here is the same selection as part of a codelens example. Step through it to see how the correct print is chosen.

```

x = 10
y = 10

if x < y:
    print("x is less than y")
else:
    if x > y:
        print("x is greater than y")
    else:
        print("x and y must be equal")

```

#### Check your understanding

**Checkpoint 7.6.2** Will the following code cause an error?

```

x = -10
if x < 0:
    print("The negative number", x, "is not valid here.")
else:
    if x > 0:
        print(x, "is a positive number")
    else:
        print(x, "is 0")

```

- A. No
- B. Yes

## 7.7 Chained conditionals

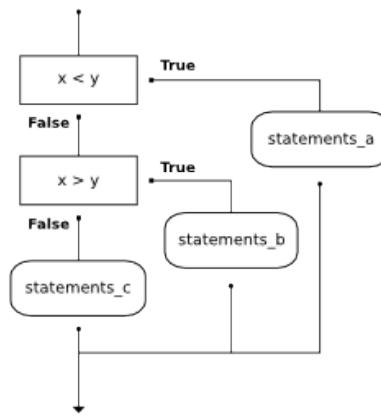
Python provides an alternative way to write nested selection such as the one shown in the previous section. This is sometimes referred to as a **chained conditional**.

```

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x and y must be equal")

```

The flow of control can be drawn in a different orientation but the resulting pattern is identical to the one shown above.



`elif` is an abbreviation of `else if`. Again, exactly one branch will be executed. There is no limit of the number of `elif` statements but only a single (and optional) final `else` statement is allowed and it must be the last branch in the statement.

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.

Here is the same program using `elif`.

```

x = 10
y = 10

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x and y must be equal")

```

**Note 7.7.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

### Check your understanding

**Checkpoint 7.7.2** Which of I, II, and III below gives the same result as the following nested `if`?

```

# nested if-else statement
x = -10
if x < 0:
    print("The negative number", x, "is not valid here.")
else:
    if x > 0:
        print(x, "is a positive number")
    else:
        print(x, "is 0")

```

```

I.

if x < 0:
    print("The negative number", x, "is not valid here.")
else x > 0:
    print(x, "is a positive number")
else:
    print(x, "is 0")

II.

if x < 0:
    print("The negative number", x, "is not valid here.")
elif x > 0:
    print(x, "is a positive number")
else:
    print(x, "is 0")

III.

if x < 0:
    print("The negative number", x, "is not valid here.")
if x > 0:
    print(x, "is a positive number")
else:
    print(x, "is 0")

```

- A. I only
- B. II only
- C. III only
- D. II and III
- E. I, II, and III

**Checkpoint 7.7.3** What will the following code print if  $x = 3$ ,  $y = 5$ , and  $z = 2$ ?

```

if x < y and x < z:
    print("a")
elif y < x and y < z:
    print("b")
else:
    print("c")

```

- A. a
- B. b
- C. c

## 7.8 Boolean Functions

### 7.8.1 Introduction

We have already seen that boolean values result from the evaluation of boolean expressions. Since the result of any expression evaluation can be returned by a function (using the `return` statement), functions can return boolean values. This turns out to be a very convenient way to hide the details of complicated

tests. For example:

```
def isDivisible(x, y):
    if x % y == 0:
        result = True
    else:
        result = False

    return result

print(isDivisible(10, 5))
```

The name of this function is `isDivisible`. It is common to give **boolean functions** names that sound like yes/no questions. `isDivisible` returns either `True` or `False` to indicate whether the `x` is or is not divisible by `y`.

We can make the function more concise by taking advantage of the fact that the condition of the `if` statement is itself a boolean expression. We can return it directly, avoiding the `if` statement altogether:

```
def isDivisible(x, y):
    return x % y == 0
```

Boolean functions are often used in conditional statements:

```
if isDivisible(x, y):
    ... # do something ...
else:
    ... # do something else ...
```

It might be tempting to write something like `if isDivisible(x, y) == True:` but the extra comparison is redundant. You only need an `==` expression if you are comparing some other type than boolean. (`isDivisible(x, y) == False` can also be made more concise as `not isDivisible(x, y)`). The following example shows the `isDivisible` function at work. Notice how descriptive the code is when we move the testing details into a boolean function. Try it with a few other actual parameters to see what is printed.

```
def isDivisible(x, y):
    return x % y == 0

if isDivisible(10, 5):
    print("That works")
else:
    print("Those values are no good")
```

Here is the same program in codelens. When we evaluate the `if` statement in the main part of the program, the evaluation of the boolean expression causes a call to the `isDivisible` function. This is very easy to see in codelens.

```
def isDivisible(x, y):
    return x % y == 0

if isDivisible(10, 5):
    print("That works")
else:
    print("Those values are no good")
```

### Check your understanding

**Checkpoint 7.8.1** What is a Boolean function?

- A. A function that returns True or False

- B. A function that takes True or False as an argument
- C. The same as a Boolean expression

**Checkpoint 7.8.2** Is the following statement legal in a Python function (assuming x, y and z are defined to be numbers)?

```
| return x + y < z
```

- A. Yes
- B. No

### 7.8.2 More Unit Testing

When we write unit tests, we should also consider **output equivalence classes** that result in significantly different results.

The `isDivisible` function can return either `True` or `False`. These two different outputs give us two equivalence classes. We then choose inputs that should give each of the different results. **It is important to have at least one test for each output equivalence class.**

```
def isDivisible(x, y):
    '''is x evenly divisible by y?'''
    return x % y == 0

if __name__ == "__main__":
    import test
```

**Note 7.8.3 Extend the program ....** Starting on line 7, write two unit tests (that should pass), one for each output equivalence class.

**Note 7.8.4** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 7.9 Glossary

### Glossary

- block.** A group of consecutive statements with the same indentation.
- body.** The block of statements in a compound statement that follows the header.
- boolean expression.** An expression that is either true or false.
- boolean function.** A function that returns a boolean value. The only possible values of the `bool` type are `False` and `True`.
- boolean value.** There are exactly two boolean values: `True` and `False`. Boolean values result when a boolean expression is evaluated by the Python interpreter. They have type `bool`.
- branch.** One of the possible paths of the flow of execution determined by conditional execution.
- chained conditional.** A conditional branch with more than two possible flows of execution. In Python chained conditionals are written with `if ... elif ... else` statements.
- comparison operator.** One of the operators that compares two values: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

**condition.** The boolean expression in a conditional statement that determines which branch is executed.

**conditional statement.** A statement that controls the flow of execution depending on some condition. In Python the keywords `if`, `elif`, and `else` are used for conditional statements.

**logical operator.** One of the operators that combines boolean expressions: `and`, `or`, and `not`.

**modulus operator.** An operator, denoted with a percent sign ( `%` ), that works on integers and yields the remainder when one number is divided by another.

**nesting.** One program structure within another, such as a conditional statement inside a branch of another conditional statement.

## 7.10 Exercises

- What do these expressions evaluate to?

1 `3 == 3`

2 `3 != 3`

3 `3 >= 4`

4 `not (3 < 4)`

- Give the **logical opposites** of these conditions. You are not allowed to use the `not` operator.

1 `a > b`

2 `a >= b`

3 `a >= 18 and day == 3`

4 `a >= 18 or day != 3`

- Write a function which is given an exam mark, and it returns a string — the grade for that mark — according to this scheme:

**Table 7.10.1**

Mark	Grade
$\geq 90$	A
[80-90)	B
[70-80)	C
[60-70)	D
$< 60$	F

The square and round brackets denote closed and open intervals. A closed interval includes the number, and open interval excludes it. So 79.99999 gets grade C , but 80 gets grade B.

Test your function by printing the mark and the grade for a number of different marks.

```

def getGrade(grade):
    #your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(getGrade(95), 'A', 'Tested\u
            getGrade\uon\uinput\uof\u95')
        self.assertEqual(getGrade(85), 'B', 'Tested\u
            getGrade\uon\uinput\uof\u85')
        self.assertEqual(getGrade(65), 'D', 'Tested\u
            getGrade\uon\uinput\uof\u65')
        self.assertEqual(getGrade(79.99999), 'C', 'Tested\u
            getGrade\uon\uinput\uof\u79.9999')
        self.assertEqual(getGrade(80), 'B', 'Tested\u
            getGrade\uon\uinput\uof\u80')

myTests().main()

```

4. Modify the turtle bar chart program from the previous chapter so that the bar for any value of 200 or more is filled with red, values between [100 and 200) are filled yellow, and bars representing values less than 100 are filled green.
5. In the turtle bar chart program, what do you expect to happen if one or more of the data values in the list is negative? Go back and try it out. Change the program so that when it prints the text value for the negative bars, it puts the text above the base of the bar (on the 0 axis).
6. Write a function `findHypot`. The function will be given the length of two sides of a right-angled triangle and it should return the length of the hypotenuse. (Hint: `x ** 0.5` will return the square root, or use `sqrt` from the math module)

```

def findHypot(a,b):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        self.assertEqual(findHypot(12.0,5.0),13.0,"Tested\u
            findHypot\uon\uinputs\uof\u12.0\uand\u5.0")
        self.assertEqual(findHypot(14.0,48.0),50.0,"Tested\u
            findHypot\uon\uinputs\uof\u14.0\uand\u48.0")
        self.assertEqual(findHypot(21.0,72.0),75.0,"Tested\u
            findHypot\uon\uinputs\uof\u21.0\uand\u72.0")
        self.assertAlmostEqual(findHypot(1,1.73205),1.999999,2,"Tested\u
            findHypot\uon\uinputs\uof\u1\uand\u1.73205")

myTests().main()

```

7. Write a function called `is_even(n)` that takes an integer as an argument and returns `True` if the argument is an **even number** and `False` if it is **odd**.

```
def is_even(n):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        self.assertEqual(is_even(10), True, "Tested\u2022
            is_even\u2022on\u2022input\u2022of\u202210")
        self.assertEqual(is_even(5), False, "Tested\u2022
            is_even\u2022on\u2022input\u2022of\u20225")
        self.assertEqual(is_even(1), False, "Tested\u2022
            is_even\u2022on\u2022input\u2022of\u20221")
        self.assertEqual(is_even(0), True, "Tested\u2022
            is_even\u2022on\u2022input\u2022of\u20220")

myTests().main()
```

8. Now write the function `is_odd(n)` that returns `True` when `n` is odd and `False` otherwise.

```
def is_odd(n):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        self.assertEqual(is_odd(10), False, "Tested\u2022
            is_odd\u2022on\u2022input\u2022of\u202210")
        self.assertEqual(is_odd(5), True, "Tested\u2022is\u2022odd\u2022
            on\u2022input\u2022of\u20225")
        self.assertEqual(is_odd(1), True, "Tested\u2022is\u2022odd\u2022
            on\u2022input\u2022of\u20221")
        self.assertEqual(is_odd(0), False, "Tested\u2022
            is_odd\u2022on\u2022input\u2022of\u20220")

myTests().main()
```

9. Modify `is_odd` so that it uses a call to `is_even` to determine if its argument is an odd integer.

```

def is_odd(n):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        self.assertEqual(is_odd(10), False, "Tested\u
            is_odd\uon\uinput\uof\u10")
        self.assertEqual(is_odd(5), True, "Tested\uis_odd\u
            on\uinput\uof\u5")
        self.assertEqual(is_odd(1), True, "Tested\uis_odd\u
            on\uinput\uof\u1")
        self.assertEqual(is_odd(0), False, "Tested\u
            is_odd\uon\uinput\uof\u0")

myTests().main()

```

10. Write a function `is_rightangled` which, given the length of three sides of a triangle, will determine whether the triangle is right-angled. Assume that the third argument to the function is always the longest side. It will return `True` if the triangle is right-angled, or `False` otherwise.

Hint: floating point arithmetic is not always exactly accurate, so it is not safe to test floating point numbers for equality. If a good programmer wants to know whether `x` is equal or close enough to `y`, they would probably code it up as

```

if abs(x - y) < 0.001:      # if x is approximately
    equal to y
    ...

def is_rightangled(a, b, c):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        self.assertEqual(is_rightangled(1.5, 2.0, 2.5), True, "Tested\u
            is_rightangled\uon\uinputs\uof\u1.5,\u2.0\uand\u
            2.5")
        self.assertEqual(is_rightangled(4.0, 8.0, 16.0), False, "Tested\u
            is_rightangled\uon\uinputs\uof\u4.0,\u8.0\uand\u
            16.0")
        self.assertEqual(is_rightangled(4.1, 8.2, 9.1678787077), True, "Tested\u
            is_rightangled\uon\uinputs\uof\u4.1,\u8.2\uand\u
            9.1678787077")
        self.assertEqual(is_rightangled(4.1, 8.2, 9.16787), True, "Tested\u
            is_rightangled\uon\uinputs\uof\u4.1,\u8.2\uand\u
            9.16787")
        self.assertEqual(is_rightangled(4.1, 8.2, 9.168), False, "Tested\u
            is_rightangled\uon\uinputs\uof\u4.1,\u8.2\uand\u
            9.168")
        self.assertEqual(is_rightangled(0.5, 0.4, 0.64031), True, "Tested\u
            is_rightangled\uon\uinputs\uof\u0.5,\u0.4\uand\u
            0.64031")

myTests().main()

```

11. Extend the above program so that the sides can be given to the function in any order.

```

def is_rightangled(a, b, c):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        self.assertEqual(is_rightangled(1.5,2.5,2.0),True,"Tested\u
            is_rightangled\uon\uinputs\uof\u1.5,\u2.5\uand\u
            2.0")
        self.assertEqual(is_rightangled(16.0,4.0,8.0),False,"Tested\u
            is_rightangled\uon\uinputs\uof\u16.0,\u4.0\uand\u
            8.0")
        self.assertEqual(is_rightangled(4.1,8.2,9.1678787077),True,"Tested\u
            is_rightangled\uon\uinputs\uof\u4.1,\u8.2\uand\u
            9.1678787077")
        self.assertEqual(is_rightangled(4.1,9.16787,8.2),True,"Tested\u
            is_rightangled\uon\uinputs\uof\u4.1,\u9.16787\u
            and\u8.2")
        self.assertEqual(is_rightangled(4.1,8.2,9.168),False,"Tested\u
            is_rightangled\uon\uinputs\uof\u4.1,\u8.2\uand\u
            9.168")
        self.assertEqual(is_rightangled(0.5,0.64031,0.4),True,"Tested\u
            is_rightangled\uon\uinputs\uof\u0.5,\u0.64031\u
            and\u0.4")

myTests().main()

```

12. 3 criteria must be taken into account to identify leap years:

The year is evenly divisible by 4;

If the year can be evenly divided by 100, it is NOT a leap year, unless;

The year is also evenly divisible by 400. Then it is a leap year.

Write a function that takes a year as a parameter and returns True if the year is a leap year, False otherwise.

```

def isLeap(year):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        self.assertEqual(isLeap(1944), True, "Tested\u
            isLeap\uon\uan\uinput\uof\u1944")
        self.assertEqual(isLeap(2011), False, "Tested\u
            isLeap\uon\uan\uinput\uof\u2011")
        self.assertEqual(isLeap(1986), False, "Tested\u
            isLeap\uon\uan\uinput\uof\u1986")
        self.assertEqual(isLeap(1800), False, "Tested\u
            isLeap\uon\uan\uinput\uof\u1800")
        self.assertEqual(isLeap(1900), False, "Tested\u
            isLeap\uon\uan\uinput\uof\u1900")
        self.assertEqual(isLeap(2000), True, "Tested\u
            isLeap\uon\uan\uinput\uof\u2000")
        self.assertEqual(isLeap(2056), True, "Tested\u
            isLeap\uon\uan\uinput\uof\u2056")

myTests().main()

```

13. Implement the calculator for the date of Easter.

The following algorithm computes the date for Easter Sunday for any year between 1900 to 2099.

Ask the user to enter a year. Compute the following:

- 1 a = year % 19
- 2 b = year % 4
- 3 c = year % 7
- 4 d = (19 \* a + 24) % 30
- 5 e = (2 \* b + 4 \* c + 6 \* d + 5) % 7
- 6 dateofeaster = 22 + d + e

Special note: The algorithm can give a date in April. Also, if the year is one of four special years (1954, 1981, 2049, or 2076) then subtract 7 from the date.

Your program should print an error message if the user provides a date that is out of range.

# Chapter 8

# More About Iteration

## 8.1 Iteration Revisited

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.

Repeated execution of a sequence of statements is called **iteration**. Because iteration is so common, Python provides several language features to make it easier. We've already seen the `for` statement in a previous chapter. This is a very common form of iteration in Python. In this chapter we are going to look at the `while` statement — another way to have your program do iteration.

## 8.2 The **for** loop revisited

Recall that the `for` loop processes each item in a list. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed. We saw this example in an earlier chapter.

```
for f in ["Joe", "Amy", "Brad", "Angelina", "Zuki",  
    "Thandi", "Paris"]:  
    print("Hi", f, "Please, come to my party on Saturday")
```

We have also seen iteration paired with the update idea to form the accumulator pattern. For example, to compute the sum of the first  $n$  integers, we could create a for loop using the `range` to produce the numbers 1 through  $n$ . Using the accumulator pattern, we can start with a running total variable initialized to 0 and on each iteration, add the current value of the loop variable. A function to compute this sum is shown below.

```
def sumTo(aBound):
    theSum = 0
    for aNumber in range(1, aBound + 1):
        theSum = theSum + aNumber

    return theSum

print(sumTo(4))

print(sumTo(1000))
```

To review, the variable `theSum` is called the accumulator. It is initialized to zero before we start the loop. The loop variable, `aNumber` will take on the values produced by the `range(1, aBound + 1)` function call. Note that this produces all the integers from 1 up to the value of `aBound`. If we had not added 1 to `aBound`, the range would have stopped one value short since `range` does not include the upper bound.

The assignment statement, `theSum = theSum + aNumber`, updates `theSum` each time through the loop. This accumulates the running total. Finally, we return the value of the accumulator.

### Check Your Understanding

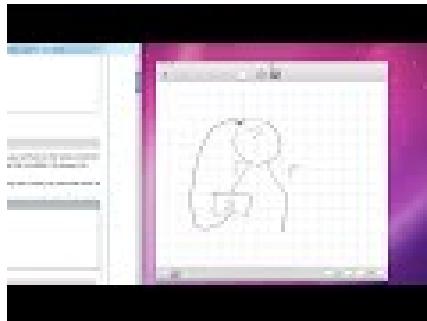
**Checkpoint 8.2.1** The following code contains an nested loop. How many times will the phrase “We made it here!” be printed on the console?

```
def printnums(x,y):
    for h in range(y):
        print("We\u2022made\u2022it\u2022here!")
        for i in range(x):
            print("We\u2022made\u2022it\u2022here!")

printnums(5, 3)
```

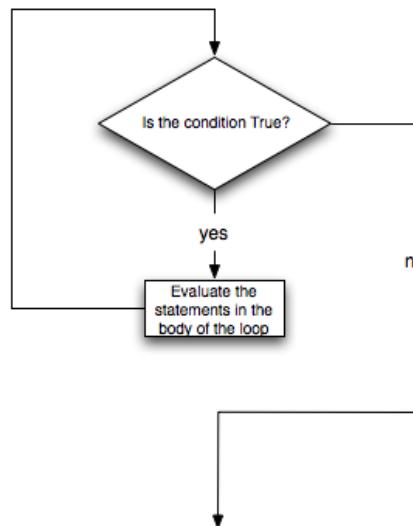
- A. 5
- B. 8
- C. 15
- D. 18
- E. 20

## 8.3 The `while` Statement



There is another Python statement that can also be used to build an iteration. It is called the `while` statement. The `while` statement provides a much more general mechanism for iterating. Similar to the `if` statement, it uses a boolean expression to control the flow of execution. The body of `while` will be repeated as long as the controlling boolean expression evaluates to `True`.

The following figure shows the flow of control.



We can use the `while` loop to create any type of iteration we wish, including anything that we have previously done with a `for` loop. For example, the program in the previous section could be rewritten using `while`. Instead of relying on the `range` function to produce the numbers for our summation, we will need to produce them ourselves. To do this, we will create a variable called `aNumber` and initialize it to 1, the first number in the summation. Every iteration will add `aNumber` to the running total until all the values have been used. In order to control the iteration, we must create a boolean expression that evaluates to `True` as long as we want to keep adding values to our running total. In this case, as long as `aNumber` is less than or equal to the bound, we should keep going.

Here is a new version of the summation program that uses a `while` statement.

```

def sumTo(aBound):
    """Return the sum of 1+2+3...+n"""

    theSum = 0
    aNumber = 1
    while aNumber <= aBound:
        theSum = theSum + aNumber
        aNumber = aNumber + 1
    return theSum

print(sumTo(4))
print(sumTo(1000))

```

You can almost read the `while` statement as if it were in natural language. It means, while `aNumber` is less than or equal to `aBound`, continue executing the body of the loop. Within the body, each time, update `theSum` using the accumulator pattern and increment `aNumber`. After the body of the loop, we go back up to the condition of the `while` and reevaluate it. When `aNumber` becomes greater than `aBound`, the condition fails and flow of control continues to the `return` statement.

The same program in codelens will allow you to observe the flow of execution.

```

def sumTo(aBound):
    """Return the sum of 1+2+3...n"""

    theSum = 0
    aNumber = 1
    while aNumber <= aBound:
        theSum = theSum + aNumber
        aNumber = aNumber + 1
    return theSum

print(sumTo(4))

```

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

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

The body consists of all of the statements below the header with the same indentation.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is `False` the first time through the loop, the statements inside the loop are never executed.

**Warning 8.3.1** Though Python's `while` is very close to the English "while", there is an important difference: In English "while X, do Y", we usually assume that immediately after X becomes false, we stop with Y. In Python there is *not* an immediate stop: After the initial test, any following tests come only after the execution of the *whole* body, even if the condition becomes false in the middle of the loop body.

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. This is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions written on the back of the shampoo bottle (lather, rinse, repeat) create an infinite loop.

In the case shown above, we can prove that the loop terminates because we know that the value of `aBound` is finite, and we can see that the value of `aNumber` increments each time through the loop, so eventually it will have to exceed `aBound`. In other cases, it is not so easy to tell.

**Note 8.3.2** Introduction of the `while` statement causes us to think about the types of iteration we have seen. The `for` statement will always iterate through a sequence of values like the list of names for the party or the list of numbers created by `range`. Since we know that it will iterate once for each value in the collection, it is often said that a `for` loop creates a **definite iteration** because we definitely know how many times we are going to iterate. On the other hand, the `while` statement is dependent on a condition that needs to evaluate to `False` in order for the loop to terminate. Since we do not necessarily know when this will happen, it creates what we call **indefinite iteration**. Indefinite iteration simply means that we don't know how many times we will repeat but eventually the condition controlling the iteration will fail and the iteration will stop. (Unless we have an infinite loop which is of course a problem.)

What you will notice here is that the `while` loop is more work for you — the programmer — than the equivalent `for` loop. When using a `while` loop you have to control the loop variable yourself. You give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates.

So why have two kinds of loop if `for` looks easier? The next section, [Section 8.4](#), shows an indefinite iteration where we need the extra power that we get from the `while` loop.

**Note 8.3.3** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

### Check your understanding

**Checkpoint 8.3.4** True or False: You can rewrite any for-loop as a while-loop.

- A. True
- B. False

**Checkpoint 8.3.5** The following code contains an infinite loop. Which is the best explanation for why the loop does not terminate?

```
n = 10
answer = 1
while n > 0:
    answer = answer + n
    n = n + 1
print(answer)
```

- A. n starts at 10 and is incremented by 1 each time through the loop, so it will always be positive
- B. answer starts at 1 and is incremented by n each time, so it will always be positive
- C. You cannot compare n to 0 in while loop. You must compare it to another variable.
- D. In the while loop body, we must set n to False, and this code does not do that.

**Checkpoint 8.3.6** What is printed by this code?

```
n = 1
x = 2
while n < 5:
    n = n + 1
    x = x + 1
    n = n + 2
    x = x + n
print(n, x)
```

- A. 4 7
- B. 5 7
- C. 7 15

## 8.4 Randomly Walking Turtles

Suppose we want to entertain ourselves by watching a turtle wander around randomly inside the screen. When we run the program we want the turtle and program to behave in the following way:

- 1 The turtle begins in the center of the screen.
- 2 Flip a coin. If it's heads then turn to the left 90 degrees. If it's tails then turn to the right 90 degrees.
- 3 Take 50 steps forward.
- 4 If the turtle has moved outside the screen then stop, otherwise go back to step 2 and repeat.

Notice that we cannot predict how many times the turtle will need to flip the coin before it wanders out of the screen, so we can't use a for loop in this case. In fact, although very unlikely, this program might never end, that is why we call this indefinite iteration.

So based on the problem description above, we can outline a program as follows:

```
create a window and a turtle

while the turtle is still in the window:
    generate a random number between 0 and 1
    if the number == 0 (heads):
        turn left
    else:
        turn right
    move the turtle forward 50
```

Now, probably the only thing that seems a bit confusing to you is the part about whether or not the turtle is still in the screen. But this is the nice thing about programming, we can delay the tough stuff and get *something* in our program working right away. The way we are going to do this is to delegate the work of deciding whether the turtle is still in the screen or not to a boolean function. Let's call this boolean function `isInScreen`. We can write a very simple version of this boolean function by having it always return `True`, or by having it decide randomly, the point is to have it do something simple so that we can focus on the parts we already know how to do well and get them working. Since having it always return `True` would not be a good idea we will write our version to decide randomly. Let's say that there is a 90% chance the turtle is still in the window and 10% that the turtle has escaped.

```
import random
import turtle

def isInScreen(w, t):
    if random.random() > 0.1:
        return True
    else:
        return False

t = turtle.Turtle()
wn = turtle.Screen()
```

```

t.shape('turtle')
while isInScreen(wn, t):
    coin = random.randrange(0, 2)
    if coin == 0:                      # heads
        t.left(90)
    else:                             # tails
        t.right(90)

    t.forward(50)

wn.exitonclick()

```

Now we have a working program that draws a random walk of our turtle that has a 90% chance of staying on the screen. We are in a good position, because a large part of our program is working and we can focus on the next bit of work – deciding whether the turtle is inside the screen boundaries or not.

We can find out the width and the height of the screen using the `window_width` and `window_height` methods of the screen object. However, remember that the turtle starts at position 0,0 in the middle of the screen. So we never want the turtle to go farther right than `width/2` or farther left than `negative width/2`. We never want the turtle to go further up than `height/2` or further down than `negative height/2`. Once we know what the boundaries are we can use some conditionals to check the turtle position against the boundaries and return `False` if the turtle is outside or `True` if the turtle is inside.

Once we have computed our boundaries we can get the current position of the turtle and then use conditionals to decide. Here is one implementation:

```

def isInScreen(wn, t):
    leftBound = -(wn.window_width() / 2)
    rightBound = wn.window_width() / 2
    topBound = wn.window_height() / 2
    bottomBound = -(wn.window_height() / 2)

    turtleX = t.xcor()
    turtleY = t.ycor()

    stillIn = True
    if turtleX > rightBound or turtleX < leftBound:
        stillIn = False
    if turtleY > topBound or turtleY < bottomBound:
        stillIn = False

    return stillIn

```

There are lots of ways that the conditional could be written. In this case we have given `stillIn` the default value of `True` and use two `if` statements to possibly set the value to `False`. You could rewrite this to use nested conditionals or `elif` statements and set `stillIn` to `True` in an `else` clause.

Here is the full version of our random walk program.

```

import random
import turtle

def isInScreen(w, t):
    leftBound = - w.window_width() / 2
    rightBound = w.window_width() / 2
    topBound = w.window_height() / 2
    bottomBound = -w.window_height() / 2

```

```
    turtleX = t.xcor()
    turtleY = t.ycor()

    stillIn = True
    if turtleX > rightBound or turtleX < leftBound:
        stillIn = False
    if turtleY > topBound or turtleY < bottomBound:
        stillIn = False

    return stillIn

t = turtle.Turtle()
wn = turtle.Screen()

t.shape('turtle')
while isInScreen(wn,t):
    coin = random.randrange(0, 2)
    if coin == 0:
        t.left(90)
    else:
        t.right(90)

    t.forward(50)

wn.exitonclick()
```

We could have written this program without using a boolean function. You might want to try to rewrite it using a complex condition on the while statement. However, using a boolean function makes the program much more readable and easier to understand. It also gives us another tool to use if this was a larger program and we needed to have a check for whether the turtle was still in the screen in another part of the program. Another advantage is that if you ever need to write a similar program, you can reuse this function with confidence the next time you need it. Breaking up this program into a couple of parts is another example of functional decomposition.

#### Check your understanding

**Checkpoint 8.4.1** Which type of loop can be used to perform the following iteration: You choose a positive integer at random and then print the numbers from 1 up to and including the selected integer.

- A. a for-loop or a while-loop
- B. only a for-loop
- C. only a while-loop

**Checkpoint 8.4.2** In the random walk program in this section, what does the `isInScreen` function do?

- A. Returns True if the turtle is still on the screen and False if the turtle is no longer on the screen.
- B. Uses a while loop to move the turtle randomly until it goes off the screen.
- C. Turns the turtle right or left at random and moves the turtle forward 50.
- D. Calculates and returns the position of the turtle in the window.

## 8.5 The $3n + 1$ Sequence

As another example of indefinite iteration, let's look at a sequence that has fascinated mathematicians for many years. The rule for creating the sequence is to start from some positive integer, call it  $n$ , and to generate the next term of the sequence from  $n$ , either by halving  $n$ , whenever  $n$  is even, or else by multiplying it by three and adding 1 when it is odd. The sequence terminates when  $n$  reaches 1.

This Python function captures that algorithm. Try running this program several times supplying different values for  $n$ .

```
def seq3np1(n):
    """Prints the 3n+1 sequence from n, terminating when it reaches 1."""
    while n != 1:
        print(n)
        if n % 2 == 0:          # n is even
            n = n // 2
        else:                   # n is odd
            n = n * 3 + 1
        print(n)                # the last print is 1

seq3np1(3)
```

The condition for this loop is  $n \neq 1$ . The loop will continue running until  $n == 1$  (which will make the condition false).

Each time through the loop, the program prints the value of  $n$  and then checks whether it is even or odd using the remainder operator. If it is even, the value of  $n$  is divided by 2 using integer division. If it is odd, the value is replaced by  $n * 3 + 1$ . Try some other examples.

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.

You might like to have some fun and see if you can find a small starting number that needs more than a hundred steps before it terminates.

### Note 8.5.1 Lab.

- Experimenting with the  $3n+1$  Sequence [Section 21.9](#) In this guided lab exercise we will try to learn more about this sequence.

Particular values aside, the interesting question is whether we can prove that this sequence terminates for *all* positive values of  $n$ . So far, no one has been able to prove it *or* disprove it!

Think carefully about what would be needed for a proof or disproof of the hypothesis “*All positive integers will eventually converge to 1*”. With fast computers we have been able to test every integer up to very large values, and so far, they all eventually end up at 1. But this doesn’t mean that there might not be some as-yet untested number which does not reduce to 1.

You’ll notice that if you don’t stop when you reach one, the sequence gets into its own loop: 1, 4, 2, 1, 4, 2, 1, 4, and so on. One possibility is that there might be other cycles that we just haven’t found.

**Note 8.5.2 Choosing between `for` and `while`.** Use a `for` loop if you know the maximum number of times that you’ll need to execute the body. For example, if you’re traversing a list of elements, or can formulate a suitable call

to `range`, then choose the `for` loop.

So any problem like “iterate this weather model run for 1000 cycles”, or “search this list of words”, “check all integers up to 10000 to see which are prime” suggest that a `for` loop is best.

By contrast, if you are required to repeat some computation until some condition is met, as we did in this  $3n + 1$  problem, you’ll need a `while` loop.

As we noted before, the first case is called **definite iteration** — we have some definite bounds for what is needed. The latter case is called **indefinite iteration** — we are not sure how many iterations we’ll need — we cannot even establish an upper bound!

### Check your understanding

**Checkpoint 8.5.3** Consider the code that prints the  $3n+1$  sequence in Active-Code box 6. Will the while loop in this code always terminate for any positive integer value of  $n$ ?

- A. Yes.
- B. No.
- C. No one knows.

## 8.6 Newton’s Method

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  $n$ . If you start with almost any approximation, you can compute a better approximation with the following formula:

```
|   better = 1/2 * (approx + n/approx)
```

Execute this algorithm a few times using your calculator. Can you see why each iteration brings your estimate a little closer? One of the amazing properties of this particular algorithm is how quickly it converges to an accurate answer.

The following implementation of Newton’s method requires two parameters. The first is the value whose square root will be approximated. The second is the number of times to iterate the calculation yielding a better result.

```
def newtonSqrt(n, howmany):
    approx = 0.5 * n
    for i in range(howmany):
        betterapprox = 0.5 * (approx + n/approx)
        approx = betterapprox
    return betterapprox

print(newtonSqrt(100, 10))
print(newtonSqrt(4, 10))
print(newtonSqrt(1, 10))
```

**Note 8.6.1 Modify the program ....** All three of the calls to `newtonSqrt` in the previous example produce the correct square root for the first parameter. However, were 10 iterations required to get the correct answer? Experiment with different values for the number of repetitions (the 10 on lines 8, 9, and 10). For each of these calls, find the **smallest** value for the number of repetitions that will produce the **correct** result.

Repeating more than the required number of times is a waste of computing resources. So definite iteration is not a good solution to this problem.

In general, Newton's algorithm will eventually reach a point where the new approximation is no better than the previous. At that point, we could simply stop. In other words, by repeatedly applying this formula until the better approximation gets close enough to the previous one, we can write a function for computing the square root that uses the number of iterations necessary and no more.

This implementation, shown in codelens, uses a `while` condition to execute until the approximation is no longer changing. Each time through the loop we compute a “better” approximation using the formula described earlier. As long as the “better” is different, we try again. Step through the program and watch the approximations get closer and closer.

```
def newtonSqrt(n):
    approx = 0.5 * n
    better = 0.5 * (approx + n/approx)
    while better != approx:
        approx = better
        better = 0.5 * (approx + n/approx)
    return approx

print(newtonSqrt(10))
```

**Note 8.6.2** The `while` statement shown above uses comparison of two floating point numbers in the condition. Since floating point numbers are themselves approximation of real numbers in mathematics, it is often better to compare for a result that is within some small threshold of the value you are looking for.

## 8.7 The Accumulator Pattern Revisited

Newton's method to calculate square roots is an example of an algorithm that repeats as long as it can improve the result. It's just a variation of our accumulator pattern. Many algorithms work this way and so require the use of indefinite iteration.

Here is another accumulator pattern program. It adds up the reciprocals of powers of two.

$$\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \cdots + \frac{1}{2^n}$$

You may have studied this sequence in a math class and learned that the sum approaches but never reaches 2.0. That is true in theory. However, when we implement this summation in a program, we see something different.

```
def sumTo():
    """Return the sum of reciprocals of powers of 2"""

    theSum = 0
    aNumber = 0
    while theSum < 2.0:
        theSum = theSum + 1/2**aNumber
        aNumber = aNumber + 1
```

```

    return theSum

print(sumTo())

```

**Note 8.7.1 Modify the program ....** If the sum never reaches 2.0, the loop would never terminate. But the loop does stop! How many repetitions did it make before it stopped?

On line 9 (not indented), print the value of `aNumber` and you will see.

But **why** did it reach 2.0? Are those math teachers wrong?

## 8.8 Other uses of `while`

### 8.8.1 Sentinel Values

Indefinite loops are much more common in the real world than definite loops.

- If you are selling tickets to an event, you don't know in advance how many tickets you will sell. You keep selling tickets as long as people come to the door and there's room in the hall.
- When the baggage crew unloads a plane, they don't know in advance how many suitcases there are. They just keep unloading while there are bags left in the cargo hold. (Why *your* suitcase is always the last one is an entirely different problem.)
- When you go through the checkout line at the grocery, the clerks don't know in advance how many items there are. They just keep ringing up items as long as there are more on the conveyor belt.

Let's implement the last of these in Python, by asking the user for prices and keeping a running total and count of items. When the last item is entered, the program gives the grand total, number of items, and average price. We'll need these variables:

- `total` - this will start at zero
- `count` - the number of items, which also starts at zero
- `moreItems` - a boolean that tells us whether more items are waiting; this starts as `True`

The pseudocode (code written half in English, half in Python) for the body of the loop looks something like this:

```

while moreItems
    ask for price
    add price to total
    add one to count

```

This pseudocode has no option to set `moreItems` to `False`, so it would run forever. In a grocery store, there's a little plastic bar that you put after your last item to separate your groceries from those of the person behind you; that's how the clerk knows you have no more items. We don't have a "little plastic bar" data type in Python, so we'll do the next best thing: we will use a `price` of zero to mean "this is my last item." In this program, zero is a **sentinel value**, a value used to signal the end of the loop. Here's the code:

```

def checkout():
    total = 0
    count = 0
    moreItems = True
    while moreItems:
        price = float(input('Enter price of item (0 when done):'))
        if price != 0:
            count = count + 1
            total = total + price
            print('Subtotal:', total)
        else:
            moreItems = False
    average = total / count
    print('Total items:', count)
    print('Total $', total)
    print('Average price per item: $', average)

checkout()

```

There are still a few problems with this program.

- If you enter a negative number, it will be added to the total and count. Modify the code so that negative numbers give an error message instead (but don't end the loop). Hint: `elif` is your friend.
- If you enter zero the first time you are asked for a price, the loop will end, and the program will try to divide by zero. Use an `if/else` statement outside the loop to avoid the division by zero and tell the user that you can't compute an average without data.
- This program doesn't display the amounts to two decimal places. In the next chapter you will see the [Section 9.5](#) that will do the trick.

### Check your understanding

**Checkpoint 8.8.1** True or False: A while loop will continue to iterate forever unless it meets a condition to stop.

- True
- False

### 8.8.2 Validating Input

You can also use a `while` loop when you want to **validate** input; when you want to make sure the user has entered valid input for a prompt. Let's say you want a function that asks a yes-or-no question. In this case, you want to make sure that the person using your program enters either a Y for yes or N for no (in either upper or lower case). Here is a program that uses a `while` loop to keep asking until it receives a valid answer. As a preview of coming attractions, it uses the `upper()` method which is described in [Section 9.5](#) to convert a string to upper case. When you run the following code, try typing something other than Y or N to see how the code reacts:

```

def get_yes_or_no(message):
    valid_input = False
    answer = input(message)
    while not valid_input:
        answer = answer.upper() # convert to upper case

```

```

        if answer == 'Y' or answer == 'N':
            valid_input = True
        else:
            answer = input('Please enter Y for yes or N for
                           no.\n' + message)
    return answer

response = get_yes_or_no('Do you like lima beans? Y es or
                           N o: ')
if response == 'Y':
    print('Great! They are very healthy.')
else:
    print('Too bad. If cooked right, they are quite tasty.')

```

## 8.9 Algorithms Revisited

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, understanding that hard problems can be solved by step-by-step algorithmic processes is one of the major simplifying breakthroughs that has had enormous benefits. So while the execution of the algorithm may be boring and may require no intelligence, algorithmic or computational thinking is having a vast impact. It is the process of designing algorithms that 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 a step-by-step mechanical algorithm.

**Note 8.9.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 8.10 Simple Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, “*This is great! We can use the computers to generate the tables, so there will be no errors.*” That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium processor chip used to perform floating-point division.

Although a power of 2 table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
print("\n", '\t', "2**n")      #table column headings
print("---", '\t', "-----")

for x in range(13):           # generate values for columns
    print(x, '\t', 2 ** x)
```

The string '\t' represents a **tab character**. The backslash character in '\t' indicates the beginning of an **escape sequence**. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence \n represents a **newline**.

An escape sequence can appear anywhere in a string. In this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?

As characters and strings are displayed on the screen, an invisible marker called the **cursor** keeps track of where the next character will go. After a `print` function is executed, the cursor normally goes to the beginning of the next line.

The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program. Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

### Check your understanding

**Checkpoint 8.10.1** What is the difference between a tab ('\t') and a sequence of spaces?

- A. A tab will line up items in a second column, regardless of how many characters were in the first column, while spaces will not.
- B. There is no difference
- C. A tab is wider than a sequence of spaces
- D. You must use tabs for creating tables. You cannot use spaces.

## 8.11 2-Dimensional Iteration: Image Processing

### 8.11.1 Introduction

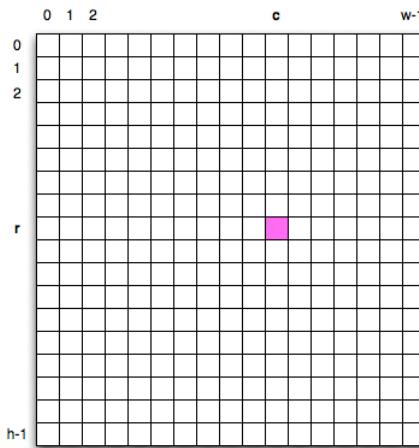
Two dimensional tables have both rows and columns. You have probably seen many tables like this if you have used a spreadsheet program. Another object

that is organized in rows and columns is a digital image. In this section we will explore how iteration allows us to manipulate these images.

A **digital image** is a finite collection of small, discrete picture elements called **pixels**. These pixels are organized in a two-dimensional grid. Each pixel represents the smallest amount of picture information that is available. Sometimes these pixels appear as small “dots”.

Each image (grid of pixels) has its own width and its own height. The width is the number of columns and the height is the number of rows. We can name the pixels in the grid by using the column number and row number. However, it is very important to remember that computer scientists like to start counting with 0! This means that if there are 20 rows, they will be named 0,1,2, and so on through 19. This will be very useful later when we iterate using range.

In the figure below, the pixel of interest is found at column **c** and row **r**.



### 8.11.2 The RGB Color Model

Each pixel of the image will represent a single color. The specific color depends on a formula that mixes various amounts of three basic colors: red, green, and blue. This technique for creating color is known as the **RGB Color Model**. The amount of each color, sometimes called the **intensity** of the color, allows us to have very fine control over the resulting color.

The minimum intensity value for a basic color is 0. For example if the red intensity is 0, then there is no red in the pixel. The maximum intensity is 255. This means that there are actually 256 different amounts of intensity for each basic color. Since there are three basic colors, that means that you can create 256<sup>3</sup> distinct colors using the RGB Color Model.

Here are the red, green and blue intensities for some common colors. Note that “Black” is represented by a pixel having no basic color. On the other hand, “White” has maximum values for all three basic color components.

**Table 8.11.1**

Color	Red	Green	Blue
Red	255	0	0
Green	0	255	0
Blue	0	0	255
White	255	255	255
Black	0	0	0
Yellow	255	255	0
Magenta	255	0	255

In order to manipulate an image, we need to be able to access individual pixels. This capability is provided by a module called `image`, provided in ActiveCode. See [Section 8.12](#) for ways to deal with images in standard Python. The `image` module defines two classes: `Image` and `Pixel`.

Each `Pixel` object has three attributes: the red intensity, the green intensity, and the blue intensity. A pixel provides three methods that allow us to ask for the intensity values. They are called `getRed`, `getGreen`, and `getBlue`. In addition, we can ask a pixel to change an intensity value using its `setRed`, `setGreen`, and `setBlue` methods.

**Table 8.11.2**

Method Name	Example	Explanation
<code>Pixel(r,g,b)</code>	<code>Pixel(20,100,50)</code>	Create a new pixel with 20 red, 100 green, and 50 blue.
<code>getRed()</code>	<code>r = p.getRed()</code>	Return the red component intensity.
<code>getGreen()</code>	<code>r = p.getGreen()</code>	Return the green component intensity.
<code>getBlue()</code>	<code>r = p.getBlue()</code>	Return the blue component intensity.
<code>setRed()</code>	<code>p.setRed(100)</code>	Set the red component intensity to 100.
<code>setGreen()</code>	<code>p.setGreen(45)</code>	Set the green component intensity to 45.
<code>setBlue()</code>	<code>p.setBlue(156)</code>	Set the blue component intensity to 156.

In the example below, we first create a pixel with 45 units of red, 76 units of green, and 200 units of blue. We then print the current amount of red, change the amount of red, and finally, set the amount of blue to be the same as the current amount of green.

```
import image

p = image.Pixel(45, 76, 200)
print(p.getRed())
p.setRed(66)
print(p.getRed())
p.setBlue(p.getGreen())
print(p.getGreen(), p.getBlue())
```

### Check your understanding

**Checkpoint 8.11.3** If you have a pixel whose RGB value is (50, 0, 0), what color will this pixel appear to be?

- A. Dark red
- B. Light red
- C. Dark green
- D. Light green

### 8.11.3 Image Objects

To access the pixels in a real image, we need to first create an `Image` object. Image objects can be created in two ways. First, an `Image` object can be made from the files that store digital images. The `Image` object has an attribute corresponding to the width, the height, and the collection of pixels in the image.

It is also possible to create an `Image` object that is “empty”. An `EmptyImage` has a width and a height. However, the pixel collection consists of only “White” pixels.

We can ask an `image` object to return its size using the `getWidth` and `getHeight` methods. We can also get a pixel from a particular location in the image using `getPixel` and change the pixel at a particular location using `setPixel`.

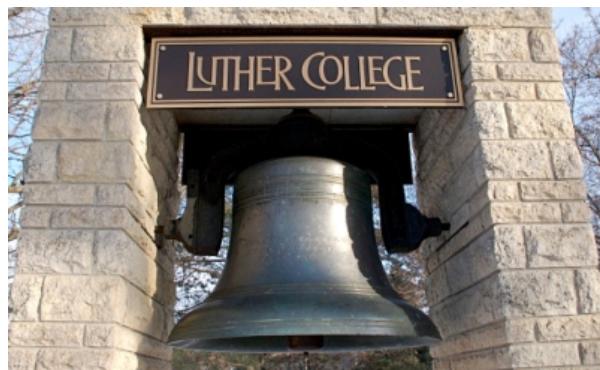
The `Image` class is shown below. Note that the first two entries show how to create image objects. The parameters are different depending on whether you are using an image file or creating an empty image.

**Table 8.11.4**

Method Name	Example	Explanation
<code>Image(filename)</code>	<code>img = image.Image("cy.png")</code>	Create an <code>Image</code> object from the file <code>cy.png</code>
<code>EmptyImage()</code>	<code>img = image.EmptyImage(100,200)</code>	Create an <code>Image</code> object that has all “White” pixels
<code>getWidth()</code>	<code>w = img.getWidth()</code>	Return the width of the image in pixels.
<code>getHeight()</code>	<code>h = img.getHeight()</code>	Return the height of the image in pixels.
<code>getPixel(col,row)</code>	<code>p = img.getPixel(35,86)</code>	Return the pixel at column 35, row 86.
<code>setPixel(col,row,p)</code>	<code>img.setPixel(100,50,mp)</code>	Set the pixel at column 100, row 50 to be <code>mp</code>

Consider the image shown below. Assume that the image is stored in a file called “`luther.jpg`”. Line 2 opens the file and uses the contents to create an `image` object that is referred to by `img`. Once we have an `image` object, we can use the methods described above to access information about the image or to get a specific pixel and check on its basic color intensities.

Data: `luther.jpg`



```

import image
img = image.Image("luther.jpg")

print(img.getWidth())
print(img.getHeight())

p = img.getPixel(45, 55)
print(p.getRed(), p.getGreen(), p.getBlue())

```

When you run the program you can see that the image has a width of 400 pixels and a height of 244 pixels. Also, the pixel at column 45, row 55, has RGB values of 165, 161, and 158. Try a few other pixel locations by changing the `getPixel` arguments and rerunning the program.

#### Check your understanding

**Checkpoint 8.11.5** Using the previous ActiveCode example, select the answer that is closest to the RGB values of the pixel at row 100, column 30? The values may be off by one or two due to differences in browsers.

- A. 149 132 122
- B. 183 179 170
- C. 165 161 158
- D. 201 104 115

#### 8.11.4 Image Processing and Nested Iteration

**Image processing** refers to the ability to manipulate the individual pixels in a digital image. In order to process all of the pixels, we need to be able to systematically visit all of the rows and columns in the image. The best way to do this is to use **nested iteration**.

Nested iteration simply means that we will place one iteration construct inside of another. We will call these two iterations the **outer iteration** and the **inner iteration**. To see how this works, consider the iteration below.

```
| for i in range(5):  
|     print(i)
```

We have seen this enough times to know that the value of `i` will be 0, then 1, then 2, and so on up to 4. The `print` will be performed once for each pass. However, the body of the loop can contain any statements including another iteration (another `for` statement). For example,

```
| for i in range(5):  
|     for j in range(3):  
|         print(i, j)
```

The `for i` iteration is the outer iteration and the `for j` iteration is the inner iteration. Each pass through the outer iteration will result in the complete processing of the inner iteration from beginning to end. This means that the output from this nested iteration will show that for each value of `i`, all values of `j` will occur.

Here is the same example in activecode. Try it. Note that the value of `i` stays the same while the value of `j` changes. The inner iteration, in effect, is moving faster than the outer iteration.

```
| for i in range(5):  
|     for j in range(3):  
|         print(i, j)
```

Another way to see this in more detail is to examine the behavior with codelens. Step through the iterations to see the flow of control as it occurs with the nested iteration. Again, for every value of `i`, all of the values of `j` will occur. You can see that the inner iteration completes before going on to the next pass of the outer iteration.

```

for i in range(5):
    for j in range(3):
        print(i, j)

```

Our goal with image processing is to visit each pixel. We will use an iteration to process each row. Within that iteration, we will use a nested iteration to process each column. The result is a nested iteration, similar to the one seen above, where the outer `for` loop processes the rows, from 0 up to but not including the height of the image. The inner `for` loop will process each column of a row, again from 0 up to but not including the width of the image.

The resulting code will look like the following. We are now free to do anything we wish to each pixel in the image.

```

for row in range(img.getHeight()):
    for col in range(img.getWidth()):
        # do something with the pixel at position (col, row)

```

One of the easiest image processing algorithms will create what is known as a **negative** image. A negative image simply means that each pixel will be the opposite of what it was originally. But what does opposite mean?

In the RGB color model, we can consider the opposite of the red component as the difference between the original red and 255. For example, if the original red component was 50, then the opposite, or negative red value would be 255-50 or 205. In other words, pixels with a lot of red will have negatives with little red and pixels with little red will have negatives with a lot. We do the same for the blue and green as well.

The program below implements this algorithm using the previous image (`luther.jpg`). Run it to see the resulting negative image. Note that there is a lot of processing taking place and this may take a few seconds to complete. In addition, here are two other images that you can use (`cy.png` and `goldygopher.png`). <h4 style="text-align: center;">cy.png</h4> <h4 style="text-align: center;">goldygopher.png</h4>Change the name of the file in the `image.Image()` call to see how these images look as negatives. Also, note that there is an `exitonclick` method call at the very end which will close the window when you click on it. This will allow you to “clear the screen” before drawing the next negative.

```

import image

img = image.Image("luther.jpg")
win = image.ImageWin(img.getWidth(), img.getHeight())
img.draw(win)
img.setDelay(1,15)    # setDelay(0) turns off animation

for row in range(img.getHeight()):
    for col in range(img.getWidth()):
        p = img.getPixel(col, row)

        newred = 255 - p.getRed()
        newgreen = 255 - p.getGreen()
        newblue = 255 - p.getBlue()

        newpixel = image.Pixel(newred, newgreen, newblue)

        img.setPixel(col, row, newpixel)

```

```

|     img.draw(win)
|     win.exitonclick()

```

Let's take a closer look at the code. After importing the image module, we create an image object called `img` that represents a typical digital photo. We will update each pixel in this image from top to bottom, left to right, which you should be able to observe. You can change the values in `setDelay` to make the program progress faster or slower.

Lines 8 and 9 create the nested iteration that we discussed earlier. This allows us to process each pixel in the image. Line 10 gets an individual pixel.

Lines 12-14 create the negative intensity values by extracting the original intensity from the pixel and subtracting it from 255. Once we have the `newred`, `newgreen`, and `newblue` values, we can create a new pixel (Line 15).

Finally, we need to replace the old pixel with the new pixel in our image. It is important to put the new pixel into the same location as the original pixel that it came from in the digital photo.

Try to change the program above so that the outer loop iterates over the columns and the inner loop iterates over the rows. We still create a negative image, but you can see that the pixels update in a very different order.

**Note 8.11.6 Other pixel manipulation.** There are a number of different image processing algorithms that follow the same pattern as shown above. Namely, take the original pixel, extract the red, green, and blue intensities, and then create a new pixel from them. The new pixel is inserted into an empty image at the same location as the original.

For example, you can create a **gray scale** pixel by averaging the red, green and blue intensities and then using that value for all intensities.

From the gray scale you can create **black white** by setting a threshold and selecting to either insert a white pixel or a black pixel into the empty image.

You can also do some complex arithmetic and create interesting effects, such as **Sepia Tone**<sup>1</sup>

You have just passed a very important point in your study of Python programming. Even though there is much more that we will do, you have learned all of the basic building blocks that are necessary to solve many interesting problems. From an algorithm point of view, you can now implement selection and iteration. You can also solve problems by breaking them down into smaller parts, writing functions for those parts, and then calling the functions to complete the implementation. What remains is to focus on ways that we can better represent our problems in terms of the data that we manipulate. We will now turn our attention to studying the main data collections provided by Python.

### Check your understanding

**Checkpoint 8.11.7** What will the following nested for-loop print? (Note, if you are having trouble with this question, review CodeLens 3).

```

|     for i in range(3):
|         for j in range(2):
|             print(i, j)

```

a.

```

0 0
0 1
1 0
1 1

```

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Sepia\\_tone#Sepia\\_toning](http://en.wikipedia.org/wiki/Sepia_tone#Sepia_toning)

```
2 0  
2 1
```

b.

```
0 0  
1 0  
2 0  
0 1  
1 1  
2 1
```

c.

```
0 0  
0 1  
0 2  
1 0  
1 1  
1 2
```

d.

```
0 1  
0 1  
0 1
```

- A. Output a
- B. Output b
- C. Output c
- D. Output d

**Checkpoint 8.11.8** What would the image produced from ActiveCode box 16 look like if you replaced the lines:

```
newred = 255 - p.getRed()  
newgreen = 255 - p.getGreen()  
newblue = 255 - p.getBlue()
```

with the lines:

```
newred = p.getRed()  
newgreen = 0  
newblue = 0
```

- A. It would look like a red-washed version of the bell image
- B. It would be a solid red rectangle the same size as the original image
- C. It would look the same as the original image
- D. It would look the same as the negative image in the example code

## 8.12 Image Processing on Your Own

If you want to try some image processing on your own, outside of the textbook you can do so using the `cImage` module. The easiest way to get this is to run the command `pip install cImage` from the command line. You can also download `cImage.py` from [The github page<sup>1</sup>](#). If you get the file from github, put `cImage.py` in the same folder as your program you can then do the following to be fully compatible with the code in this book.

```
import cImage as image
img = image.Image("myfile.gif")
```

**Note 8.12.1 Note.** One important caveat about using `cImage.py` is that it will only work with GIF files unless you also install the Python Image Library. Don't worry if you `pip install cImage` it will automatically take care of this for you. Otherwise, the easiest version to install is called `Pillow`. If you have the `pip` command installed on your computer this is really easy to install, with `pip install pillow` otherwise you will need to follow the instructions on the [Python Package Index<sup>2</sup>](#) page. With `Pillow` installed you will be able to use almost any kind of image that you download.

**Note 8.12.2** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 8.13 Glossary

### Glossary

**algorithm.** A step-by-step process for solving a category of problems.

**body.** The indented statements after a heading ending in a colon, for instance after a `for`-loop heading.

**counter.** A variable used to count something, usually initialized to zero and incremented in the body of a loop.

**cursor.** An invisible marker that keeps track of where the next character will be printed.

**definite iteration.** A loop where we have an upper bound on the number of times the body will be executed. Definite iteration is usually best coded as a `for` loop.

**escape sequence.** An escape character, `\`, followed by one or more printable characters used to designate a nonprintable character.

**generalize.** To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

**infinite loop.** A loop in which the terminating condition is never satisfied.

**indefinite iteration.** A loop where we just need to keep going until some condition is met. A `while` statement is used for this case.

**iteration.** Repeated execution of a set of programming statements.

**loop.** A statement or group of statements that execute repeatedly until a terminating condition is satisfied.

---

<sup>1</sup><https://github.com/bnmnntp/cImage>

<sup>2</sup><https://pypi.python.org/pypi/Pillow/>

- loop variable.** A variable used as part of the terminating condition of a loop.
- nested loop.** A loop inside the body of another loop.
- newline.** A special character that causes the cursor to move to the beginning of the next line.
- reassignment.** Making more than one assignment to the same variable during the execution of a program.
- tab.** A special character that causes the cursor to move to the next tab stop on the current line.

## 8.14 Exercises

1. Add a print statement to Newton's `sqrt` function that prints out better each time it is calculated. Call your modified function with 25 as an argument and record the results.

```
| def newtonSqrt(n):
```

2. Write a function `print_triangular_numbers(n)` that prints out the first  $n$  triangular numbers. A call to `print_triangular_numbers(5)` would produce the following output:

```
1      1
2      3
3      6
4     10
5     15
```

(*hint: use a web search to find out what a triangular number is.*)

```
Write a function ``print_triangular_numbers(n)`` that
    prints out the first
n triangular numbers. A call to
    ``print_triangular_numbers(5)`` would
produce the following output::
```

```
1      1
2      3
3      6
4     10
5     15
```

```
(*hint: use a web search to find out what a triangular
    number is.*)
~~~~
def print_triangular_numbers(n):
    # your code here
```

3. Write a function, `is_prime`, that takes a single integer argument and returns `True` when the argument is a *prime number* and `False` otherwise.

```

def is_prime(n):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(is_prime(2), True, "Tested on 2, which is a prime number.")
        self.assertEqual(is_prime(4187), False, "Tested on 4187, which is not a prime number. It is divisible by 53 and 79.")
        self.assertEqual(is_prime(22), False, "Tested on 22, which is not a prime number. It is divisible by 2 and 11.")
        self.assertEqual(is_prime(4813), True, "Tested on 4813, which is a prime number.")

myTests().main()

```

4. Modify the walking turtle program so that rather than a 90 degree left or right turn the angle of the turn is determined randomly at each step.
5. Modify the turtle walk program so that you have two turtles each with a random starting location. Keep the turtles moving until one of them leaves the screen.
6. Modify the previous turtle walk program so that the turtle turns around when it hits the wall or when one turtle collides with another turtle (when the positions of the two turtles are closer than some small number).
7. Write a function to remove all the red from an image. <h4 style="text-align: left;">For this and the following exercises, use the luther.jpg photo.</h4>
8. Write a function to convert the image to grayscale.
9. Write a function to convert an image to black and white.
10. Sepia Tone images are those brownish colored images that may remind you of times past. The formula for creating a sepia tone is as follows:

```

newR = (R * 0.393 + G * 0.769 + B * 0.189)
newG = (R * 0.349 + G * 0.686 + B * 0.168)
newB = (R * 0.272 + G * 0.534 + B * 0.131)

```

Write a function to convert an image to sepia tone. *Hint:* Remember that rgb values must be integers between 0 and 255.

11. Write a function to uniformly enlarge an image by a factor of 2 (double the size).
12. After you have scaled an image too much it looks blocky. One way of reducing the blockiness of the image is to replace each pixel with the average values of the pixels around it. This has the effect of smoothing out the changes in color. Write a function that takes an image as a parameter and smooths the image. Your function should return a new image that is the same as the old but smoothed.
13. Write a general pixel mapper function that will take an image and a pixel mapping function as parameters. The pixel mapping function should

perform a manipulation on a single pixel and return a new pixel.

14. When you scan in images using a scanner they may have lots of noise due to dust particles on the image itself or the scanner itself, or the images may even be damaged. One way of eliminating this noise is to replace each pixel by the median value of the pixels surrounding it.
15. Research the Sobel edge detection algorithm and implement it.

# Chapter 9

## Strings

### 9.1 Strings Revisited

Throughout the first chapters of this book we have used strings to represent words or phrases that we wanted to print out. Our definition was simple: a string is simply some characters inside quotes. In this chapter we explore strings in much more detail.

### 9.2 A Collection Data Type

So far we have seen built-in types like: `int`, `float`, `bool`, `str` and we've seen lists. `int`, `float`, and `bool` are considered to be simple or primitive data types because their values are not composed of any smaller parts. They cannot be broken down. On the other hand, strings and lists are different from the others because they are made up of smaller pieces. In the case of strings, they are made up of smaller strings each containing one **character**.

Types that are comprised of smaller pieces are called **collection data types**. Depending on what we are doing, we may want to treat a collection data type as a single entity (the whole), or we may want to access its parts. This ambiguity is useful.

Strings can be defined as sequential collections of characters. This means that the individual characters that make up the string are assumed to be in a particular order from left to right.

A string that contains no characters, often referred to as the **empty string**, is still considered to be a string. It is simply a sequence of zero characters and is represented by `"` or `""` (two single or two double quotes with nothing in between).

### 9.3 Operations on Strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that `message` has type `string`):

```
message - 1
"Hello" / 123
message * "Hello"
"15" + 2
```

Interestingly, the `+` operator does work with strings, but for strings, the `+` operator represents **concatenation**, not addition. Concatenation means joining the two operands by linking them end-to-end. For example:

```
fruit = "banana"
bakedGood = "nutbread"
print(fruit + bakedGood)
```

The output of this program is `banana nut bread`. The space before the word `nut` is part of the string and is necessary to produce the space between the concatenated strings. Take out the space and run it again.

The `*` operator also works on strings. It performs repetition. For example, `'Fun'*3` is `'FunFunFun'`. One of the operands has to be a string and the other has to be an integer.

```
print("Go" * 6)

name = "Packers"
print(name * 3)

print(name + "Go" * 3)

print((name + "Go") * 3)
```

This interpretation of `+` and `*` makes sense by analogy with addition and multiplication. Just as  $4 \times 3$  is equivalent to  $4+4+4$ , we expect `"Go" * 3` to be the same as `"Go" + "Go" + "Go"`, and it is. Note also in the last example that the order of operations for `*` and `+` is the same as it was for arithmetic. The repetition is done before the concatenation. If you want to cause the concatenation to be done first, you will need to use parenthesis.

#### Check your understanding

**Checkpoint 9.3.1** What is printed by the following statements?

```
s = "python"
t = "rocks"
print(s + t)
```

- A. python rocks
- B. python
- C. pythonrocks
- D. Error, you cannot add two strings together.

**Checkpoint 9.3.2** What is printed by the following statements?

```
s = "python"
excl = "!"
print(s+excl*3)
```

- A. python!!!
- B. python!python!python!
- C. pythonpythonpython!
- D. Error, you cannot perform concatenation and repetition at the same time.

## 9.4 Index Operator: Working with the Characters of a String

The **indexing operator** (Python uses square brackets to enclose the index) selects a single character from a string. The characters are accessed by their position or index value. For example, in the string shown below, the 14 characters are indexed left to right from position 0 to position 13.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
L	u	t	h	e	r		C	o	l	l	e	g	e
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

It is also the case that the positions are named from right to left using negative numbers where -1 is the rightmost index and so on. Note that the character at index 6 (or -8) is the blank character.

```

school = "LutherCollege"
m = school[2]
print(m)

lastchar = school[-1]
print(lastchar)

```

The expression `school[2]` selects the character at index 2 from `school`, and creates a new string containing just this one character. The variable `m` refers to the result.

Remember that computer scientists often start counting from zero. The letter at index zero of "Luther College" is L. So at position [2] we have the letter t.

If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets. Give it a try.

The expression in brackets is called an **index**. An index specifies a member of an ordered collection. In this case the collection of characters in the string. The index *indicates* which character you want. It can be any integer expression so long as it evaluates to a valid index value.

Note that indexing returns a *string* — Python has no special type for a single character. It is just a string of length 1.

### Check your understanding

**Checkpoint 9.4.1** What is printed by the following statements?

```

s = "python(rocks"
print(s[3])

```

- A. t
- B. h
- C. c
- D. Error, you cannot use the [ ] operator with a string.

**Checkpoint 9.4.2** What is printed by the following statements?

```

s = "python(rocks"
print(s[2] + s[-5])

```

- A. tr
- B. ps

- C. nn
- D. Error, you cannot use the [ ] operator with the + operator.

## 9.5 String Methods

### 9.5.1 Introduction

We previously saw that each turtle instance has its own attributes and a number of methods that can be applied to the instance. For example, we wrote `tess.right(90)` when we wanted the turtle object `tess` to perform the `right` method to turn to the right 90 degrees. The “dot notation” is the way we connect the name of an object to the name of a method it can perform.

Strings are also objects. Each string instance has its own attributes and methods. The most important attribute of the string is the collection of characters. There are a wide variety of methods. Try the following program.

```
ss = "Hello ,\uWorld"
print(ss.upper())

tt = ss.lower()
print(tt)
```

In this example, `upper` is a method that can be invoked on any string object to create a new string in which all the characters are in uppercase. `lower` works in a similar fashion changing all characters in the string to lowercase. (The original string `ss` remains unchanged. A new string `tt` is created.)

In addition to `upper` and `lower`, the following table provides a summary of some other useful string methods. There are a few activecode examples that follow so that you can try them out.

**Table 9.5.1**

Method	Parameters	Description
upper	none	Returns a string in all uppercase
lower	none	Returns a string in all lowercase
capitalize	none	Returns a string with first character capitalized, the rest lower
strip	none	Returns a string with the leading and trailing whitespace removed
lstrip	none	Returns a string with the leading whitespace removed
rstrip	none	Returns a string with the trailing whitespace removed
count	item	Returns the number of occurrences of item
replace	old, new	Replaces all occurrences of old substring with new
center	width	Returns a string centered in a field of width spaces
ljust	width	Returns a string left justified in a field of width spaces
rjust	width	Returns a string right justified in a field of width spaces
find	item	Returns the leftmost index where the substring item is found, or -1 if not found
rfind	item	Returns the rightmost index where the substring item is found, or -1 if not found
index	item	Like find except causes a runtime error if item is not found
rindex	item	Like rfind except causes a runtime error if item is not found
format	substitutions	Involved! See <a href="#">Subsection 9.5.2</a> , below

You should experiment with these methods so that you understand what they do. Note once again that the methods that return strings do not change the original. You can also consult the [Python documentation for strings](#)<sup>1</sup>.

---

<sup>1</sup><https://docs.python.org/3/library/stdtypes.html#string-methods>

```

ss = "Hello, World"
els = ss.count("l")
print(els)

print("***" + ss.strip() + "***")
print("***" + ss.lstrip() + "***")
print("***" + ss.rstrip() + "***")

news = ss.replace("o", "***")
print(news)

food = "banana bread"
print(food.capitalize())

print("*" + food.center(25) + "*")
print("*" + food.ljust(25) + "*")      # stars added to show
                                         bounds
print("*" + food.rjust(25) + "*")

print(food.find("e"))
print(food.find("na"))
print(food.find("b"))

print(food.rfind("e"))
print(food.rfind("na"))
print(food.rfind("b"))

print(food.index("e"))

```

### Check your understanding

**Checkpoint 9.5.2** What is printed by the following statements?

```

s = "python rocks"
print(s.count("o") + s.count("p"))

```

- A. 0
- B. 2
- C. 3

**Checkpoint 9.5.3** What is printed by the following statements?

```

s = "python rocks"
print(s[1] * s.index("n"))

```

- A. yyyyy
- B. 55555
- C. n
- D. Error, you cannot combine all those things together.

## 9.5.2 String Format Method

In grade school quizzes a common convention is to use fill-in-the blanks. For instance,

Hello \_\_\_\_\_!

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. *We use this as an analogy:* Python has a similar construction, better called fill-in-the-braces. The string method `format`, makes substitutions into places in a string enclosed in braces. Run this code:

```
person = input('Your name: ')
greeting = 'Hello {}!'.format(person)
print(greeting)
```

There are several new ideas here!

The string for the `format` method has a special form, with braces embedded. Such a string is called a *format string*. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the `format` method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier `greeting`, and then the string is printed.

The identifier `greeting` was introduced to break the operations into a clearer sequence of steps. However, since the value of `greeting` is only referenced once, it can be eliminated with the more concise version:

```
person = input('Enter your name: ')
print('Hello {}!'.format(person))
```

There can be multiple substitutions, with data of any type. Next we use floats. Try original price \$2.50 with a 7% discount:

```
origPrice = float(input('Enter the original price: '))
discount = float(input('Enter discount percentage: '))
newPrice = (1 - discount/100)*origPrice
calculation = '${} discounted by {}% is ${}.'.format(origPrice, discount, newPrice)
print(calculation)
```

The parameters are inserted into the braces in order.

If you used the data suggested, this result is not satisfying. Prices should appear with exactly two places beyond the decimal point, but that is not the default way to display floats.

Format strings can give further information inside the braces showing how to specially format data. In particular floats can be shown with a specific number of decimal places. For two decimal places, put `:.2f` inside the braces for the monetary values:

```
origPrice = float(input('Enter the original price: '))
discount = float(input('Enter discount percentage: '))
newPrice = (1 - discount/100)*origPrice
calculation = '${:.2f} discounted by {}% is ${:.2f}.'.format(origPrice, discount, newPrice)
print(calculation)
```

The 2 in the format modifier can be replaced by another integer to round to that specified number of digits.

This kind of format string depends directly on the order of the parameters to the `format` method. There are other approaches that we will skip here, explicitly numbering substitutions and taking substitutions from a dictionary.

A technical point: Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final *formatted* string. The rule is to double the braces: `{ { and } }`. For

example mathematical set notation uses braces. The initial and final doubled braces in the format string below generate literal braces in the formatted string:

```
a = 5
b = 9
setStr = 'The set is {{ {},{} }}.'.format(a, b)
print(setStr)
```

Unfortunately, at the time of this writing, the ActiveCode format implementation has a bug, printing doubled braces, but standard Python prints `{5, 9}`.

You can have multiple placeholders indexing the same argument, or perhaps even have extra arguments that are not referenced at all:

```
letter = """
Dear {0}{2}.
{0}, I have an interesting money-making proposition for
you!
If you deposit $10 million into my bank account, I can
double your money...
"""

print(letter.format("Paris", "Whitney", "Hilton"))
print(letter.format("Bill", "Henry", "Gates"))
```

**Checkpoint 9.5.4** What is printed by the following statements?

```
x = 2
y = 6
print('sum of {} and {} is {}; product: {}'.format( x, y,
    x+y, x*y))
```

- A. Nothing - it causes an error
- B. sum of {} and {} is {}; product: {}. 2 6 8 12
- C. sum of 2 and 6 is 8; product: 12.
- D. sum of {2} and {6} is {8}; product: {12}.

**Checkpoint 9.5.5** What is printed by the following statements?

```
v = 2.34567
print('{:.1f} {:.2f} {:.7f}'.format(v, v, v))
```

- A. 2.34567 2.34567 2.34567
- B. 2.3 2.34 2.34567
- C. 2.3 2.35 2.3456700

## 9.6 Length

The `len` function, when applied to a string, returns the number of characters in a string.

```
fruit = "Banana"
print(len(fruit))
```

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

```

fruit = "Banana"
sz = len(fruit)
last = fruit[sz]      # ERROR!
print(last)

```

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no letter at index position 6 in "Banana". Since we started counting at zero, the six indexes are numbered 0 to 5. To get the last character, we have to subtract 1 from the length. Give it a try in the example above.

```

fruit = "Banana"
sz = len(fruit)
lastch = fruit[sz-1]
print(lastch)

```

Alternatively in Python, we 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. Try it! Most other languages do *not* allow the negative indices, but they are a handy feature of Python!

#### Check your understanding

**Checkpoint 9.6.1** What is printed by the following statements?

```

s = "python(rocks"
print(len(s))

```

A. 11

B. 12

**Checkpoint 9.6.2** What is printed by the following statements?

```

s = "python(rocks"
print(s[len(s)-5])

```

A. o

B. r

C. s

D. Error, `len(s)` is 12 and there is no index 12.

**Checkpoint 9.6.3** What is printed by the following statements?

```

s = "python(rocks"
print(s[-3])

```

A. c

B. k

C. s

D. Error, negative indices are illegal.

## 9.7 The Slice Operator

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

```

    singers = "Peter , Paul , and Mary"
    print(singers[0:5])
    print(singers[7:11])
    print(singers[17:21])

```

The slice operator [n:m] returns the part of the string from the n'th character to the m'th character, including the first but excluding the last. In other words, start with the character at index n and go up to but do not include the character at index m. This behavior may seem counter-intuitive but if you recall the range function, it did not include its end point either.

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.

There is no Index Out Of Range exception for a slice. A slice is forgiving and shifts any offending index to something legal.

```

fruit = "banana"
print(fruit[:3])
print(fruit[3:])
print(fruit[3:-10])
print(fruit[3:99])

```

What do you think fruit[:] means?

**Check your understanding**

**Checkpoint 9.7.1** What is printed by the following statements?

```

s = "python(rocks"
print(s[3:8])

```

- A. python
- B. rocks
- C. hon r
- D. Error, you cannot have two numbers inside the [ ].

**Checkpoint 9.7.2** What is printed by the following statements?

```

s = "python(rocks"
print(s[7:11] * 3)

```

- A. rockrockrock
- B. rock rock rock
- C. rocksrocksrocks
- D. Error, you cannot use repetition with slicing.

**Note 9.7.3** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 9.8 String Comparison

The comparison operators also work on strings. To see if two strings are equal you simply write a boolean expression using the equality operator.

```

word = "banana"
if word == "banana":
    print("Yes , we have bananas !")

```

```

    else:
        print("Yes , we have NO bananas!")

```

Other comparison operations are useful for putting words in **lexicographical order**<sup>1</sup>. This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters.

```

word = "zebra"

if word < "banana":
    print("Your word , " + word + ", comes before banana .")
elif word > "banana":
    print("Your word , " + word + ", comes after banana .")
else:
    print("Yes , we have no bananas!")

```

It is probably clear to you that the word `apple` would be less than (come before) the word `banana`. After all, `a` is before `b` in the alphabet. But what if we consider the words `apple` and `Apple`? Are they the same?

```

print("apple" < "banana")

print("apple" == "Apple")
print("apple" < "Apple")

```

It turns out, as you recall from our discussion of variable names, that uppercase and lowercase letters are considered to be different from one another. The way the computer knows they are different is that each character is assigned a unique integer value. “`A`” is 65, “`B`” is 66, and “`5`” is 53. The way you can find out the so-called **ordinal value** for a given character is to use a character function called `ord`.

```

print(ord("A"))
print(ord("B"))
print(ord("5"))

print(ord("a"))
print("apple" > "Apple")

```

When you compare characters or strings to one another, Python converts the characters into their equivalent ordinal values and compares the integers from left to right. As you can see from the example above, “`a`” is greater than “`A`” so “`apple`” is greater than “`Apple`”.

Humans commonly ignore capitalization when comparing two words. However, computers do not. A common way to address this issue is to convert strings to a standard format, such as all lowercase, before performing the comparison.

There is also a similar function called `chr` that converts integers into their character equivalent.

```

print(chr(65))
print(chr(66))

print(chr(49))
print(chr(53))

print("The character for 32 is", chr(32), "!!!")
print(ord(" "))

```

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Lexicographic\\_order](http://en.wikipedia.org/wiki/Lexicographic_order)

One thing to note in the last two examples is the fact that the space character has an ordinal value (32). Even though you don't see it, it is an actual character. We sometimes call it a *nonprinting* character.

#### Check your understanding

**Checkpoint 9.8.1** Evaluate the following comparison:

```
| "Dog" < "Doghouse"
```

A. True

B. False

**Checkpoint 9.8.2** Evaluate the following comparison:

```
| "dog" < "Dog"
```

A. True

B. False

C. They are the same word

**Checkpoint 9.8.3** Evaluate the following comparison:

```
| "dog" < "Doghouse"
```

A. True

B. False

## 9.9 Strings are Immutable

One final thing that makes strings different from some other Python collection types is that you are not allowed to modify the individual characters in the collection. 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, in the following code, we would like to change the first letter of `greeting`.

```
greeting = "Hello, world!"
greeting[0] = 'J'           # ERROR!
print(greeting)
```

Instead of producing the output `Jello, world!`, this code produces the runtime error `TypeError: 'str' object does not support item assignment`.

Strings are **immutable**, which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original.

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print(newGreeting)
print(greeting)           # same as it was
```

The solution here is to concatenate a new first letter onto a slice of `greeting`. This operation has no effect on the original string.

#### Check your understanding

**Checkpoint 9.9.1** What is printed by the following statements:

```
s = "Ball"
s[0] = "C"
print(s)
```

- A. Ball
- B. Call
- C. Error

## 9.10 Traversal and the `for` Loop: By Item

A lot of computations involve processing a collection one item at a time. For strings this means that we would like to process one character at a time. Often we 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**.

We have previously seen that the `for` statement can iterate over the items of a sequence (a list of names in the case below).

```
for fname in ["Joe", "Amy", "Brad", "Angelina", "Zuki",
    "Thandi", "Paris"]:
    invitation = "Hi " + fname + ". Please come to my "
    party_on_Saturday!"
    print(invitation)
```

Recall that the loop variable takes on each value in the sequence of names. The body is performed once for each name. The same was true for the sequence of integers created by the `range` function.

```
for avalue in range(10):
    print(avalue)
```

Since a string is simply a sequence of characters, the `for` loop iterates over each character automatically.

```
for achar in "GoSpotGo":
    print(achar)
```

The loop variable `achar` is automatically reassigned each character in the string “Go Spot Go”. We will refer to this type of sequence iteration as **iteration by item**. Note that it is only possible to process the characters one at a time from left to right.

### Check your understanding

**Checkpoint 9.10.1** How many times is the word HELLO printed by the following statements?

```
s = "python(rocks"
for ch in s:
    print("HELLO")
```

- A. 10
- B. 11
- C. 12
- D. Error, the `for` statement needs to use the `range` function.

**Checkpoint 9.10.2** How many times is the word HELLO printed by the following statements?

```
s = "python(rocks"
for ch in s[3:8]:
    print("HELLO")
```

- A. 4

- B. 5
- C. 6
- D. Error, the for statement cannot use slice.

## 9.11 Traversal and the for Loop: By Index

It is also possible to use the `range` function to systematically generate the indices of the characters. The `for` loop can then be used to iterate over these positions. These positions can be used together with the indexing operator to access the individual characters in the string.

Consider the following codelens example.

```
fruit = "apple"
for idx in range(5):
    currentChar = fruit[idx]
    print(currentChar)
```

The index positions in “apple” are 0,1,2,3 and 4. This is exactly the same sequence of integers returned by `range(5)`. The first time through the for loop, `idx` will be 0 and the “a” will be printed. Then, `idx` will be reassigned to 1 and “p” will be displayed. This will repeat for all the range values up to but not including 5. Since “e” has index 4, this will be exactly right to show all of the characters.

In order to make the iteration more general, we can use the `len` function to provide the bound for `range`. This is a very common pattern for traversing any sequence by position. Make sure you understand why the `range` function behaves correctly when using `len` of the string as its parameter value.

```
fruit = "apple"
for idx in range(len(fruit)):
    print(fruit[idx])
```

You may also note that iteration by position allows the programmer to control the direction of the traversal by changing the sequence of index values. Recall that we can create ranges that count down as well as up so the following code will print the characters from right to left.

```
fruit = "apple"
for idx in range(len(fruit)-1, -1, -1):
    print(fruit[idx])
```

Trace the values of `idx` and satisfy yourself that they are correct. In particular, note the start and end of the range.

### Check your understanding

**Checkpoint 9.11.1** How many times is the letter o printed by the following statements?

```
s = "python\u rocks"
for idx in range(len(s)):
    if idx % 2 == 0:
        print(s[idx])
```

- A. 0
- B. 1
- C. 2

- D. Error, the for statement cannot have an if inside.

## 9.12 Traversal and the `while` Loop

The `while` loop can also control the generation of the index values. Remember that the programmer is responsible for setting up the initial condition, making sure that the condition is correct, and making sure that something changes inside the body to guarantee that the condition will eventually fail.

```
fruit = "apple"

position = 0
while position < len(fruit):
    print(fruit[position])
    position = position + 1
```

The loop condition is `position < len(fruit)`, so when `position` 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.

Here is the same example in codelens so that you can trace the values of the variables.

```
fruit = "apple"

position = 0
while position < len(fruit):
    print(fruit[position])
    position = position + 1
```

### Check your understanding

**Checkpoint 9.12.1** How many times is the letter o printed by the following statements?

```
s = "python(rocks"
idx = 1
while idx < len(s):
    print(s[idx])
    idx = idx + 2
```

- A. 0
- B. 1
- C. 2

**Note 9.12.2** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 9.13 The `in` and `not in` operators

The `in` operator tests if one string is a substring of another:

```
print('p' in 'apple')
print('i' in 'apple')
print('ap' in 'apple')
print('pa' in 'apple')
```

Note that a string is a substring of itself, and the empty string is a substring of any other string. (Also note that computer scientists like to think about these edge cases quite carefully!)

```

| print('a' in 'a')
| print('apple' in 'apple')
| print('' in 'a')
| print('' in 'apple')
```

The `not in` operator returns the logical opposite result of `in`.

```
| print('x' not in 'apple')
```

## 9.14 The Accumulator Pattern with Strings

Combining the `in` operator with string concatenation using `+` and the accumulator pattern, we can write a function that removes all the vowels from a string. The idea is to start with a string and iterate over each character, checking to see if the character is a vowel. As we process the characters, we will build up a new string consisting of only the nonvowel characters. To do this, we use the accumulator pattern.

Remember that the accumulator pattern allows us to keep a “running total”. With strings, we are not accumulating a numeric total. Instead we are accumulating characters onto a string.

```

def removeVowels(s):
    vowels = "aeiouAEIOU"
    sWithoutVowels = ""
    for eachChar in s:
        if eachChar not in vowels:
            sWithoutVowels = sWithoutVowels + eachChar
    return sWithoutVowels

print(removeVowels("compsci"))
print(removeVowels("aAbEefIijOopUus"))
```

Line 5 uses the `not in` operator to check whether the current character is not in the string `vowels`. The alternative to using this operator would be to write a very large `if` statement that checks each of the individual vowel characters. Note we would need to use logical `and` to be sure that the character is not any of the vowels.

```

if eachChar != 'a' and eachChar != 'e' and eachChar !=
    'i' and
    eachChar != 'o' and eachChar != 'u' and eachChar !=
    'A' and
    eachChar != 'E' and eachChar != 'I' and eachChar !=
    'O' and
    eachChar != 'U':
    sWithoutVowels = sWithoutVowels + eachChar
```

Look carefully at line 6 in the above program (`sWithoutVowels = sWithoutVowels + eachChar`). We will do this for every character that is not a vowel. This should look very familiar. As we were describing earlier, it is an example of the accumulator pattern, this time using a string to “accumulate” the final result. In words it says that the new value of `sWithoutVowels` will be the old value of `sWithoutVowels` concatenated with the value of `eachChar`. We are building the result string character by character.

Take a close look also at the initialization of `sWithoutVowels`. We start with an empty string and then begin adding new characters to the end.

Step through the function using codelens to see the accumulator variable grow.

```
def removeVowels(s):
    vowels = "aeiouAEIOU"
    sWithoutVowels = ""
    for eachChar in s:
        if eachChar not in vowels:
            sWithoutVowels = sWithoutVowels + eachChar
    return sWithoutVowels

print(removeVowels("compsci"))
```

### Check your understanding

**Checkpoint 9.14.1** What is printed by the following statements:

```
s = "ball"
r = ""
for item in s:
    r = item.upper() + r
print(r)
```

- A. Ball
- B. BALL
- C. LLAB

**Note 9.14.2** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 9.15 Turtles and Strings and L-Systems

This section describes a much more interested example of string iteration and the accumulator pattern. Even though it seems like we are doing something that is much more complex, the basic processing is the same as was shown in the previous sections.

In 1968 Aristid Lindenmayer, a biologist, invented a formal system that provides a mathematical description of plant growth known as an **L-system**. L-systems were designed to model the growth of biological systems. You can think of L-systems as containing the instructions for how a single cell can grow into a complex organism. L-systems can be used to specify the **rules** for all kinds of interesting patterns. In our case, we are going to use them to specify the rules for drawing pictures.

The rules of an L-system are really a set of instructions for transforming one string into a new string. After a number of these string transformations are complete, the string contains a set of instructions. Our plan is to let these instructions direct a turtle as it draws a picture.

To begin, we will look at an example set of rules:

**Table 9.15.1**

A	Axiom
A -> B	Rule 1 Change A to B
B -> AB	Rule 2 Change B to AB

Each rule set contains an axiom which represents the starting point in the transformations that will follow. The rules are of the form:

**left hand side -> right hand side**

where the left hand side is a single symbol and the right hand side is a sequence of symbols. You can think of both sides as being simple strings. The way the rules are used is to replace occurrences of the left hand side with the corresponding right hand side.

Now let's look at these simple rules in action, starting with the string A:

```
A
B      Apply Rule 1 (A is replaced by B)
AB     Apply Rule 2 (B is replaced by AB)
BAB    Apply Rule 1 to A then Rule 2 to B
ABBAB  Apply Rule 2 to B, Rule 1 to A, and Rule 2 to B
```

Notice that each line represents a new transformation for entire string. Each character that matches a left-hand side of a rule in the original has been replaced by the corresponding right-hand side of that same rule. After doing the replacement for each character in the original, we have one transformation.

So how would we encode these rules in a Python program? There are a couple of very important things to note here:

- 1 Rules are very much like if statements.
- 2 We are going to start with a string and iterate over each of its characters.
- 3 As we apply the rules to one string we leave that string alone and create a brand new string using the accumulator pattern. When we are all done with the original we replace it with the new string.

Let's look at a simple Python program that implements the example set of rules described above.

```
def applyRules(lhch):
    rhstr = ""
    if lhch == 'A':
        rhstr = 'B'    # Rule 1
    elif lhch == 'B':
        rhstr = 'AB'   # Rule 2
    else:
        rhstr = lhch   # no rules apply so keep the
                       # character

    return rhstr


def processString(oldStr):
    newstr = ""
    for ch in oldStr:
        newstr = newstr + applyRules(ch)

    return newstr


def createLSystem(numIters, axiom):
    startString = axiom
    endString = ""
    for i in range(numIters):
```

```

        endString = processString(startString)
        startString = endString

    return endString

print(createLSystem(4, "A"))

```

Try running the example above with different values for the `numIters` parameter. You should see that for values 1, 2, 3, and 4, the strings generated follow the example above exactly.

One of the nice things about the program above is that if you want to implement a different set of rules, you don't need to re-write the entire program. All you need to do is re-write the `applyRules` function.

Suppose you had the following rules:

**Table 9.15.2**

A	Axiom
A -> BAB	Rule 1 Change A to BAB

What kind of a string would these rules create? Modify the program above to implement the rule.

Now let's look at a real L-system that implements a famous drawing. This L-system has just two rules:

**Table 9.15.3**

F	Axiom
F -> F-F++F-F	Rule 1

This L-system uses symbols that will have special meaning when we use them later with the turtle to draw a picture.

**Table 9.15.4**

F	Go forward by some number of units
B	Go backward by some number of units
-	Turn left by some degrees
+	Turn right by some degrees

Here is the `applyRules` function for this L-system.

```

def applyRules(ch):
    newstr = ""
    if ch == 'F':
        newstr = 'F-F++F-F' # Rule 1
    else:
        newstr = ch # no rules apply so keep the character

    return newstr

```

Pretty simple so far. As you can imagine this string will get pretty long with a few applications of the rules. You might try to expand the string a couple of times on your own just to see.

The last step is to take the final string and turn it into a picture. Let's assume that we are always going to go forward or backward by 5 units. In addition we will also assume that when the turtle turns left or right we'll turn by 60 degrees. Now look at the string `F-F++F-F`. You might try to use the explanation above to show the resulting picture that this simple string

represents. At this point its not a very exciting drawing, but once we expand it a few times it will get a lot more interesting.

To create a Python function to draw a string we will write a function called `drawLsystem`. The function will take four parameters:

- A turtle to do the drawing
- An expanded string that contains the results of expanding the rules above.
- An angle to turn
- A distance to move forward or backward

```
def drawLsystem(aTurtle, instructions, angle, distance):
    for cmd in instructions:
        if cmd == 'F':
            aTurtle.forward(distance)
        elif cmd == 'B':
            aTurtle.backward(distance)
        elif cmd == '+':
            aTurtle.right(angle)
        elif cmd == '-':
            aTurtle.left(angle)
```

Here is the complete program in activecode. The `main` function first creates the L-system string and then it creates a turtle and passes it and the string to the drawing function.

```
import turtle

def createSystem(numIters, axiom):
    startString = axiom
    endString = ""
    for i in range(numIters):
        endString = processString(startString)
        startString = endString

    return endString

def processString(oldStr):
    newstr = ""
    for ch in oldStr:
        newstr = newstr + applyRules(ch)

    return newstr

def applyRules(ch):
    newstr = ""
    if ch == 'F':
        newstr = 'F-F++F-F' # Rule 1
    else:
        newstr = ch # no rules apply so keep the character

    return newstr

def drawLsystem(aTurtle, instructions, angle, distance):
    for cmd in instructions:
        if cmd == 'F':
            aTurtle.forward(distance)
```

```

        elif cmd == 'B':
            aTurtle.backward(distance)
        elif cmd == '+':
            aTurtle.right(angle)
        elif cmd == '-':
            aTurtle.left(angle)

def main():
    inst = createLSystem(4, "F")      # create the string
    print(inst)
    t = turtle.Turtle()              # create the turtle
    wn = turtle.Screen()

    t.up()
    t.back(200)
    t.down()
    t.speed(9)
    drawLsystem(t, inst, 60, 5)     # draw the picture
                                    # angle 60, segment
                                    # length 5
    wn.exitonclick()

main()

```

Feel free to try some different angles and segment lengths to see how the drawing changes.

## 9.16 Looping and Counting

We will finish this chapter with a few more examples that show variations on the theme of iteration through the characters of a string. We will implement a few of the methods that we described earlier to show how they can be done.

The following program counts the number of times a particular letter, `aChar`, appears in a string. It is another example of the accumulator pattern that we have seen in previous chapters.

```

def count(text, aChar):
    lettercount = 0
    for c in text:
        if c == aChar:
            lettercount = lettercount + 1
    return lettercount

print(count("banana", "a"))

```

The function `count` takes a string as its parameter. The `for` statement iterates through each character in the string and checks to see if the character is equal to the value of `aChar`. If so, the counting variable, `lettercount`, is incremented by one. When all characters have been processed, the `lettercount` is returned.

## 9.17 A `find` function

Here is an implementation for a restricted `find` method, where the target is a single character.

```

def find(astring, achar):
    """
    Find and return the index of achar in astring.
    Return -1 if achar does not occur in astring.
    """
    ix = 0
    found = False
    while ix < len(astring) and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
    if found:
        return ix
    else:
        return -1

print(find("CompSci", "p"))
print(find("CompSci", "C"))
print(find("CompSci", "i"))
print(find("CompSci", "x"))

```

In a sense, `find` is the opposite of the indexing operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears for the first time. If the character is not found, the function returns `-1`.

The `while` loop in this example uses a slightly more complex condition than we have seen in previous programs. Here there are two parts to the condition. We want to keep going if there are more characters to look through and we want to keep going if we have not found what we are looking for. The variable `found` is a boolean variable that keeps track of whether we have found the character we are searching for. It is initialized to `False`. If we find the character, we reassign `found` to `True`.

The other part of the condition is the same as we used previously to traverse the characters of the string. Since we have now combined these two parts with a logical `and`, it is necessary for them both to be `True` to continue iterating. If one part fails, the condition fails and the iteration stops.

When the iteration stops, we must ask a question to find out the individual condition that caused the termination, and then return the proper value. This is a pattern for dealing with `while` loops with compound conditions.

**Note 9.17.1** This pattern of computation is sometimes called a eureka traversal because as soon as we find what we are looking for, we can cry Eureka! and stop looking. The way we stop looking is by setting `found` to `True` which causes the condition to fail.

## 9.18 Optional parameters

To find the locations of the second or third occurrence of a character in a string, we can modify the `find` function, adding a third parameter for the starting position in the search string:

```

def find2(astring, achar, start):
    """
    Find and return the index of achar in astring.
    Return -1 if achar does not occur in astring.
    """

```

```

    ix = start
    found = False
    while ix < len(astring) and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
    if found:
        return ix
    else:
        return -1

print(find2('banana', 'a', 2))

```

The call `find2('banana', 'a', 2)` now returns 3, the index of the first occurrence of 'a' in 'banana' after index 2. What does `find2('banana', 'n', 3)` return? If you said, 4, there is a good chance you understand how `find2` works. Try it.

Better still, we can combine `find` and `find2` using an **optional parameter**.

```

def find3(astring, achar, start=0):
    """
    Find and return the index of achar in astring.
    Return -1 if achar does not occur in astring.
    """
    ix = start
    found = False
    while ix < len(astring) and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1
    if found:
        return ix
    else:
        return -1

print(find3('banana', 'a', 2))

```

The call `find3('banana', 'a', 2)` to this version of `find` behaves just like `find2`, while in the call `find3('banana', 'a')`, `start` will be set to the **default value** of 0.

Adding another optional parameter to `find` makes it search from a starting position, up to but not including the end position.

```

def find4(astring, achar, start=0, end=None):
    """
    Find and return the index of achar in astring.
    Return -1 if achar does not occur in astring.
    """
    ix = start
    if end == None:
        end = len(astring)

    found = False
    while ix < end and not found:
        if astring[ix] == achar:
            found = True
        else:
            ix = ix + 1

```

```

    if found:
        return ix
    else:
        return -1

ss = "Python\ustrings\uhave\usome\uinteresting\umethods ."

print(find4(ss, 's'))
print(find4(ss, 's', 7))
print(find4(ss, 's', 8))
print(find4(ss, 's', 8, 13))
print(find4(ss, '.'))

```

The optional value for `end` is interesting. We give it a default value `None` if the caller does not supply any argument. In the body of the function we test what `end` is and if the caller did not supply any argument, we reassign `end` to be the length of the string. If the caller has supplied an argument for `end`, however, the caller's value will be used in the loop.

The semantics of `start` and `end` in this function are precisely the same as they are in the `range` function.

## 9.19 Character classification

It is often helpful to examine a character and test whether it is upper- or lower-case, or whether it is a character or a digit. The `string` module provides several constants that are useful for these purposes. One of these, `string.digits` is equivalent to “0123456789”. It can be used to check if a character is a digit using the `in` operator.

The string `string.ascii_lowercase` contains all of the ascii letters that the system considers to be lowercase. Similarly, `string.ascii_uppercase` contains all of the uppercase letters. `string.punctuation` comprises all the characters considered to be punctuation. Run the following:

```

import string
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
print(string.punctuation)

```

For more information consult the `string` module documentaiton (see [Global Module Index<sup>1</sup>](#)).

**Note 9.19.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 9.20 Summary

This chapter introduced a lot of new ideas. The following summary may prove helpful in remembering what you learned.

### Glossary

**indexing ([]).** Access a single character in a string using its position (starting from 0). Example: `'This'[2]` evaluates to `'i'`.

---

<sup>1</sup><http://docs.python.org/py3k/py-modindex.html>

**length function (`len`).** Returns the number of characters in a string. Example: `len('happy')` evaluates to 5.

**for loop traversal (for).** *Traversing* a string means accessing each character in the string, one at a time. For example, the following for loop:

```
for ix in 'Example':
    ...
```

executes the body of the loop 7 times with different values of `ix` each time.

**slicing (`[::]`).** A *slice* is a substring of a string. Example: 'bananas and cream'[3:6] evaluates to ana (so does 'bananas and cream'[1:4]).

**string comparison (>, <, >=, <=, ==, !=).** The six common comparision operators work with strings, evaluating according to [lexigraphical order](#)<sup>1</sup>. Examples: 'apple' < 'banana' evaluates to True. 'Zeta' < 'Appricot' evaluates to False. 'Zebra' <= 'aardvark' evaluates to True because all upper case letters precede lower case letters.

**in and not in operator (in, not in).** The `in` operator tests whether one string is contained inside another string. Examples: 'heck' in "I'll be checking for you." evaluates to True. 'cheese' in "I'll be checking for you." evaluates to False.

## 9.21 Glossary

### Glossary

**collection data type.** A data type in which the values are made up of components, or elements, that are themselves values.

**default value.** The value given to an optional parameter if no argument for it is provided in the function call.

**dot notation.** Use of the **dot operator**, `.`, to access functions inside a module, or to access methods and attributes of an object.

**immutable data type.** A data type whose values cannot be changed. Modifying functions create a totally new object that does not change the original one.

**index.** A variable or value used to select a member of an ordered collection, such as a character from a string, or an element from a list.

**optional parameter.** A parameter written in a function header with an assignment to a default value which it will receive if no corresponding argument is given for it in the function call.

**slice.** A part of a string (substring) specified by a range of indices. More generally, a subsequence of any sequence type in Python can be created using the slice operator (`sequence[start:stop]`).

**traverse.** To iterate through the elements of a collection, performing a similar operation on each.

**whitespace.** Any of the characters that move the cursor without printing visible characters. The constant `string.whitespace` contains all the whitespace characters.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Lexicographic\\_order](http://en.wikipedia.org/wiki/Lexicographic_order)

## 9.22 Exercises

1. What is the result of each of the following:
  - a 'Python'[1]
  - b "Strings are sequences of characters." [5]
  - c len("wonderful")
  - d 'Mystery'[:4]
  - e 'p' in 'Pineapple'
  - f 'apple' in 'Pineapple'
  - g 'pear' not in 'Pineapple'
  - h 'apple' > 'pineapple'
  - i 'pineapple' < 'Peach'
2. 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 tries to output these names in order.

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for p in prefixes:
    print(p + suffix)
```

Of course, that's not quite right because Ouack and Quack are misspelled. Can you fix it?

```

prefixes = "JKLMNOPQ"
suffix = "ack"

for p in prefixes:
    print(p + suffix)

# Fix the loop to get the correct output.

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):

        o = self.getOutput()
        code = self.getEditorText()
        self.assertIn("if", code, "Needs a conditional.")
        self.assertIn("for", code, "Needs a loop.")
        self.assertIn("Jack", o, "J+ack=Jack")
        self.assertIn("Kack", o, "K+ack=Kack")
        self.assertIn("Lack", o, "L+ack=Lack")
        self.assertIn("Mack", o, "M+ack=Mack")
        self.assertIn("Nack", o, "N+ack=Nack")
        self.assertIn("Ouack", o, "Don't forget the misspellings. Quack is required.")
        self.assertIn("Pack", o, "P+ack=Pack")
        self.assertIn("Quack", o, "Don't forget the misspellings. Quack is required.")
        self.assertNotIn("Oack", o, "Account for the misspellings. Qack should not be in output.")
        self.assertNotIn("Qack", o, "Account for the misspellings. Qack should not be in output.")

myTests().main()

```

3. Assign to a variable in your program a triple-quoted string that contains your favorite paragraph of text - perhaps a poem, a speech, instructions to bake a cake, some inspirational verses, etc.

Write a function that counts the number of alphabetic characters (a through z, or A through Z) in your text and then keeps track of (and returns) how many are the letter 'e'. Your function should print an analysis of the text like this:

Your text contains 243 alphabetic characters, of which 109 (44.8%) are 'e'.

```

def count(p):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):
    def testOne(self):
        string1 = "e"
        string2 = "eieio"
        string3 = "eeeeeeeeeee"
        string4 = "elephant"
        self.assertEqual(count(string1), 1, "Just one e")
        self.assertEqual(count(""), 0, "Empty string")
        self.assertEqual(count(string2), 2, "Two")
        self.assertEqual(count(string3), 12, "Twelve")
        self.assertNotEqual(count(string4), 3, "Has two Es")
myTests().main()

```

4. Print out a neatly formatted multiplication table, up to 12 x 12.
5. Write a function that will return the number of digits in an integer.

```

def numDigits(n):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(numDigits(2), 1, "Tested numDigits on input of 2")
        self.assertEqual(numDigits(55), 2, "Tested numDigits on input of 55")
        self.assertEqual(numDigits(1352), 4, "Tested numDigits on input of 1352")
        self.assertEqual(numDigits(444), 3, "Tested numDigits on input of 444")

myTests().main()

```

6. Write a function that reverses its string argument.

```

def reverse(astring):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(reverse("happy"), "ppahy", "Tested\u
            reverse\uon\uinput\uof\u'happy'")
        self.assertEqual(reverse("Python"), "nohtyP", "Tested\u
            reverse\uon\uinput\uof\u'Python'")
        self.assertEqual(reverse(""), "", "Tested\ureverse\u
            on\uinput\uof\u'\'")

myTests().main()

```

7. Write a function called `mirror` that takes a string as input and returns the input string followed by the reversed version of the input string.
8. Write a function that removes all occurrences of a given letter from a string.

```

def remove_letter(theLetter, theString):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(remove_letter("a", "apple"), "pple", "Tested\u
            remove_letter\uon\uinputs\uof\u'a'\uand\u'apple'")
        self.assertEqual(remove_letter("a", "banana"), "bnn", "Tested\u
            remove_letter\uon\uinputs\uof\u'a'\uand\u'banana'")
        self.assertEqual(remove_letter("z", "banana"), "banana", "Tested\u
            remove_letter\uon\uinputs\uof\u'z'\uand\u'banana'")

myTests().main()

```

9. Write a function that recognizes palindromes. (Hint: use your `reverse` function to make this easy!).

```
def is_palindrome(myStr):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(is_palindrome("abba"),True,"Tested\u
            is_palindrome\uon\uinput\uof\u'abba'")
        self.assertEqual(is_palindrome("abab"),False,"Tested\u
            is_palindrome\uon\uinput\uof\u'abab'")
        self.assertEqual(is_palindrome("straw\u
            warts"),True,"Tested\uis_palindrome\uon\uinput\u
            of\u'straw\uwarts !")
        self.assertEqual(is_palindrome("a"),True,"Tested\u
            is_palindrome\uon\uinput\uof\u'a !")
        self.assertEqual(is_palindrome(""),True,"Tested\u
            is_palindrome\uon\uinput\uof\u' !')

myTests().main()
```

10. Write a function that counts how many non-overlapping occurrences of a substring appear in a string.

```

def count(substr, theStr):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(count("is", "Mississippi"), 2, "Tested\u
            count\uon\uinputs\uof\u'is'\uand\u
            'Mississippi')")
        self.assertEqual(count("an", "banana"), 2, "Tested\u
            count\uon\uinputs\uof\u'an'\uand\u'banana')")
        self.assertEqual(count("ana", "banana"), 1, "Tested\u
            count\uon\uinputs\uof\u'ana'\uand\u'banana')")
        self.assertEqual(count("nana", "banana"), 1, "Tested\u
            count\uon\uinputs\uof\u'nana'\uand\u'banana')")
        self.assertEqual(count("nanan", "banana"), 0, "Tested\u
            count\uon\uinputs\uof\u'nanan'\uand\u'banana')")
        self.assertEqual(count("aaa", "aaaaaa"), 2, "Tested\u
            count\uon\uinput\uof\u'aaa'\uand\u'aaaaaa')")

myTests().main()

```

11. Write a function that removes the first occurrence of a string from another string.

```

def remove(substr, theStr):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(remove("an", "banana"), "bana", "Tested\u
            remove\uon\uinputs\uof\u'an'\uand\u'banana')")
        self.assertEqual(remove("cyc", "bicycle"), "bile", "Tested\u
            remove\uon\uinputs\uof\u'cyc'\uand\u
            'bicycle')")
        self.assertEqual(remove("iss", "Mississippi"), "Mississippi", "Tested\u
            remove\uon\uinputs\uof\u'iss'\uand\u
            'Mississippi')")
        self.assertEqual(remove("egg", "bicycle"), "bicycle", "Tested\u
            remove\uon\uinputs\uof\u'egg'\uand\u
            'bicycle')")

myTests().main()

```

12. Write a function that removes all occurrences of a string from another string.

```

def remove_all(substr, theStr):
    # your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(remove_all("an", "banana"), "ba", "Tested\u
            remove_all\uon\uinputs\uof\u'an'\uand\u'banana'")\u
        self.assertEqual(remove_all("cyc", "bicycle"), "bile", "Tested\u
            remove_all\uon\uinputs\uof\u'cyc'\uand\u'bicycle'")\u
        self.assertEqual(remove_all("iss", "Mississippi"), "Mippi", "Tested\u
            remove_all\uon\uinputs\uof\u'iss'\uand\u
            'Mississippi'")\u
        self.assertEqual(remove_all("eggs", "bicycle"), "bicycle", "Tested\u
            remove_all\uon\uinputs\uof\u'eggs'\uand\u'bicycle'")\u

myTests().main()

```

13. Here is another interesting L-System called a Hilbert curve. Use 90 degrees:

```

L
L -> +RF-LFL-FR+
R -> -LF+RFR+FL-

```

14. Here is a dragon curve. Use 90 degrees.:

```

FX
X -> X+YF+
Y -> -FX-Y

```

15. Here is something called an arrowhead curve. Use 60 degrees.:

```

YF
X -> YF+XF+Y
Y -> XF-YF-X

```

16. Try the Peano-Gosper curve. Use 60 degrees.:

```

FX
X -> X+YF++YF-FX--FXFX-YF+
Y -> -FX+YFYF++YF+FX--FX-Y

```

- 17.

The Sierpinski Triangle. Use 60 degrees.:

```

FXF--FF--FF
F -> FF
X -> --FXF++FXF++FXF--

```

18. Write a function that implements a substitution cipher. In a substitution cipher one letter is substituted for another to garble the message. For

example A -> Q, B -> T, C -> G etc. your function should take two parameters, the message you want to encrypt, and a string that represents the mapping of the 26 letters in the alphabet. Your function should return a string that is the encrypted version of the message.

19. Write a function that decrypts the message from the previous exercise. It should also take two parameters. The encrypted message, and the mixed up alphabet. The function should return a string that is the same as the original unencrypted message.
20. Write a function called `remove_dups` that takes a string and creates a new string by only adding those characters that are not already present. In other words, there will never be a duplicate letter added to the new string.

```
def remove_dups(astring):
    # your code here

print(remove_dups("mississippi"))    #should print misp

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(remove_dups("pooh"), "poh", "Tested\u
            remove_dups\uon\ustring\u'pooh\u')
        self.assertEqual(remove_dups("mississippi"), "misp", "Tested\u
            remove_dups\uon\ustring\u'mississippi\u')
        self.assertEqual(remove_dups("potato"), "pota", "Tested\u
            remove_dups\uon\ustring\u'potato\u')
        self.assertEqual(remove_dups("bookkeeper"), "bokepr", "Tested\u
            remove_dups\uon\ustring\u'bookkeeper\u')
        self.assertEqual(remove_dups("oo"), "o", "Tested\u
            remove_dups\uon\ustring\u'oo\u')

myTests().main()
```

21. Write a function called `rot13` that uses the Caesar cipher to encrypt a message. The Caesar cipher works like a substitution cipher but each character is replaced by the character 13 characters to 'its right' in the alphabet. So for example the letter a becomes the letter n. If a letter is past the middle of the alphabet then the counting wraps around to the letter a again, so n becomes a, o becomes b and so on. *Hint:* Whenever you talk about things wrapping around its a good idea to think of modulo arithmetic.

```
def rot13(mess):
    # Your code here

print(rot13('abcde'))
print(rot13('nopqr'))
print(rot13(rot13('Since\urot13\uis\uasymmetric\uyou\ushould\u
    see\uthis\umessage')))
```

22. Modify this code so it prints each subtotal, the total cost, and average price to exactly two decimal places.

```
def checkout():
    total = 0
    count = 0
    moreItems = True
    while moreItems:
        price = float(input('Enter price of item (0 when done): '))
        if price != 0:
            count = count + 1
            total = total + price
            print('Subtotal:', total)
        else:
            moreItems = False
    average = total / count
    print('Total items:', count)
    print('Total $', total)
    print('Average price per item: $', average)

checkout()
```

# Chapter 10

## Lists

### 10.1 Lists

A **list** is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its **elements**. Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can have any type and for any one list, the items can be of different types.

### 10.2 List Values

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

```
| [10, 20, 30, 40]
| ["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. As we said above, 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.

```
| ["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be **nested** and the inner list is often called a  **sublist**. Finally, there is a special list that contains no elements. It is called the empty list and is denoted [ ].

As you would expect, we can also assign list values to variables and pass lists as parameters to functions.

```
vocabulary = ["iteration", "selection", "control"]
numbers = [17, 123]
empty = []
mixedlist = ["hello", 2.0, 5*2, [10, 20]]

print(numbers)
print(mixedlist)
newlist = [ numbers, vocabulary ]
print(newlist)
```

#### Check your understanding

**Checkpoint 10.2.1** A list can contain only integer items.

- A. False

- B. True

### 10.3 List Length

As with strings, the function `len` returns the length of a list (the number of items in the list). However, since lists can have items which are themselves lists, it is important to note that `len` only returns the top-most length. In other words, sublists are considered to be a single item when counting the length of the list.

```
alist = ["hello", 2.0, 5, [10, 20]]
print(len(alist))
print(len(['spam!', 1, ['Brie', 'Roquefort', 'Pol_le_Veq'],
           [1, 2, 3]]))
```

#### Check your understanding

**Checkpoint 10.3.1** What is printed by the following statements?

```
alist = [3, 67, "cat", 3.14, False]
print(len(alist))
```

- A. 4

- B. 5

**Checkpoint 10.3.2** What is printed by the following statements?

```
alist = [3, 67, "cat", [56, 57, "dog"], [], 3.14, False]
print(len(alist))
```

- A. 7

- B. 8

### 10.4 Accessing Elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string. We use the index operator (`[]` – not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at 0. Any integer expression can be used as an index and as with strings, negative index values will locate items from the right instead of from the left.

```
numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[2])
print(numbers[9 - 8])
print(numbers[-2])
print(numbers[len(numbers) - 1])
```

#### Check your understanding

**Checkpoint 10.4.1** What is printed by the following statements?

```
alist = [3, 67, "cat", [56, 57, "dog"], [], 3.14, False]
print(alist[5])
```

- A. []

- B. 3.14

- C. False

**Checkpoint 10.4.2** What is printed by the following statements?

```
alist = [3, 67, "cat", [56, 57, "dog"], [], 3.14, False]
print(alist[2].upper())
```

- A. Error, you cannot use the upper method on a list.
- B. 2
- C. CAT

**Checkpoint 10.4.3** What is printed by the following statements?

```
alist = [3, 67, "cat", [56, 57, "dog"], [], 3.14, False]
print(alist[2][0])
```

- A. 56
- B. c
- C. cat
- D. Error, you cannot have two index values unless you are using slicing.

## 10.5 List Membership

`in` and `not in` are boolean operators that test membership in a sequence. We used them previously with strings and they also work here.

```
fruit = ["apple", "orange", "banana", "cherry"]

print("apple" in fruit)
print("pear" in fruit)
```

**Check your understanding**

**Checkpoint 10.5.1** What is printed by the following statements?

```
alist = [3, 67, "cat", [56, 57, "dog"], [], 3.14, False]
print(3.14 in alist)
```

- A. True
- B. False

**Checkpoint 10.5.2** What is printed by the following statements?

```
alist = [3, 67, "cat", [56, 57, "dog"], [], 3.14, False]
print(57 in alist)
```

- A. True
- B. False

## 10.6 Concatenation and Repetition

Again, as with strings, the `+` operator concatenates lists. Similarly, the `*` operator repeats the items in a list a given number of times.

```
fruit = ["apple", "orange", "banana", "cherry"]
print([1, 2] + [3, 4])
print(fruit + [6, 7, 8, 9])
```

```

|     print([0] * 4)
|     print([1, 2, ["hello", "goodbye"]] * 2)

```

It is important to see that these operators create new lists from the elements of the operand lists. If you concatenate a list with 2 items and a list with 4 items, you will get a new list with 6 items (not a list with two sublists). Similarly, repetition of a list of 2 items 4 times will give a list with 8 items.

One way for us to make this more clear is to run a part of this example in codelens. As you step through the code, you will see the variables being created and the lists that they refer to. Pay particular attention to the fact that when `newlist` is created by the statement `newlist = fruit + numlist`, it refers to a completely new list formed by making copies of the items from `fruit` and `numlist`. You can see this very clearly in the codelens object diagram. The objects are different.

```

fruit = ["apple", "orange", "banana", "cherry"]
numlist = [6, 7]

newlist = fruit + numlist

zeros = [0] * 4

```

In Python, every object has a unique identification tag. Likewise, there is a built-in function that can be called on any object to return its unique id. The function is appropriately called `id` and takes a single parameter, the object that you are interested in knowing about. You can see in the example below that a real id is usually a very large integer value (corresponding to an address in memory).

```

>>> alist = [4, 5, 6]
>>> id(alist)
4300840544
>>>

```

### Check your understanding

**Checkpoint 10.6.1** What is printed by the following statements?

```

alist = [1, 3, 5]
blist = [2, 4, 6]
print(alist + blist)

```

- A. 6
- B. [1, 2, 3, 4, 5, 6]
- C. [1, 3, 5, 2, 4, 6]
- D. [3, 7, 11]

**Checkpoint 10.6.2** What is printed by the following statements?

```

alist = [1, 3, 5]
print(alist * 3)

```

- A. 9
- B. [1, 1, 1, 3, 3, 3, 5, 5, 5]
- C. [1, 3, 5, 1, 3, 5, 1, 3, 5]
- D. [3, 9, 15]

## 10.7 List Slices

The slice operation we saw with strings also work on lists. Remember that the first index is the starting point for the slice and the second number is one index past the end of the slice (up to but not including that element). Recall also that if you omit the first index (before the colon), the slice starts at the beginning of the sequence. If you omit the second index, the slice goes to the end of the sequence.

```
a_list = ['a', 'b', 'c', 'd', 'e', 'f']
print(a_list[1:3])
print(a_list[:4])
print(a_list[3:])
print(a_list[:])
```

### Check your understanding

**Checkpoint 10.7.1** What is printed by the following statements?

```
alist = [3, 67, "cat", [56, 57, "dog"], [], 3.14, False]
print(alist[4:])
```

- A. [[], 3.14, False]
- B. [[], 3.14]
- C. [ [56, 57, "dog"], [], 3.14, False]

## 10.8 Lists are Mutable

Unlike strings, lists are **mutable**. This means we can change an item in a list by accessing it directly as part of the assignment statement. Using the indexing operator (square brackets) on the left side of an assignment, we can update one of the list items.

```
fruit = ["banana", "apple", "cherry"]
print(fruit)

fruit[0] = "pear"
fruit[-1] = "orange"
print(fruit)
```

An assignment to an element of a list is called **item assignment**. Item assignment does not work for strings. Recall that strings are immutable.

Here is the same example in codelens so that you can step through the statements and see the changes to the list elements.

```
fruit = ["banana", "apple", "cherry"]

fruit[0] = "pear"
fruit[-1] = "orange"
```

By combining assignment with the slice operator we can update several elements at once.

```
alist = ['a', 'b', 'c', 'd', 'e', 'f']
alist[1:3] = ['x', 'y']
print(alist)
```

We can also remove elements from a list by assigning the empty list to them.

```

alist = ['a', 'b', 'c', 'd', 'e', 'f']
alist[1:3] = []
print(alist)

```

We can even insert elements into a list by squeezing them into an empty slice at the desired location.

```

alist = ['a', 'd', 'f']
alist[1:1] = ['b', 'c']
print(alist)
alist[4:4] = ['e']
print(alist)

```

### Check your understanding

**Checkpoint 10.8.1** What is printed by the following statements?

```

alist = [4, 2, 8, 6, 5]
alist[2] = True
print(alist)

```

- A. [4, 2, True, 8, 6, 5]
- B. [4, 2, True, 6, 5]
- C. Error, it is illegal to assign

## 10.9 List Deletion

Using slices to delete list elements can be awkward and therefore error-prone. Python provides an alternative that is more readable. The `del` statement removes an element from a list by using its position.

```

a = ['one', 'two', 'three']
del a[1]
print(a)

alist = ['a', 'b', 'c', 'd', 'e', 'f']
del alist[1:5]
print(alist)

```

As you might expect, `del` handles negative indices and causes a runtime error if the index is out of range. In addition, you can use a slice as an index for `del`. As usual, slices select all the elements up to, but not including, the second index, but do not cause runtime errors if the index limits go too far.

**Note 10.9.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 10.10 Objects and References

If we execute these assignment statements,

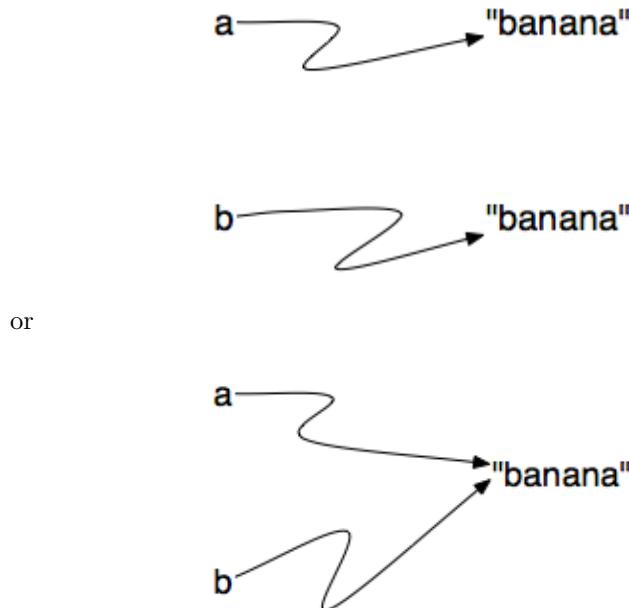
```

a = "banana"
b = "banana"

```

we know that `a` and `b` will refer to a string with the letters "banana". But we don't know yet whether they point to the *same* string.

There are two possible ways the Python interpreter could arrange its internal states:



In one case, `a` and `b` refer to two different string objects that have the same value. In the second case, they refer to the same object. Remember that an object is something a variable can refer to.

We already know that objects can be identified using their unique identifier. We can also test whether two names refer to the same object using the `is` operator. The `is` operator will return true if the two references are to the same object. In other words, the references are the same. Try our example from above.

```

a = "banana"
b = "banana"

print(a is b)

```

The answer is `True`. This tells us that both `a` and `b` refer to the same object, and that it is the second of the two reference diagrams that describes the relationship. Since strings are *immutable*, Python can optimize resources by making two names that refer to the same string literal value refer to the same object.

This is not the case with lists. Consider the following example. Here, `a` and `b` refer to two different lists, each of which happens to have the same element values.

```

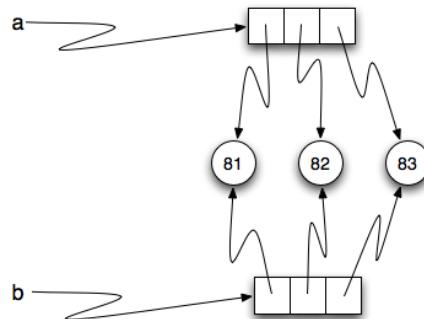
a = [81, 82, 83]
b = [81, 82, 83]

print(a is b)

print(a == b)

```

The reference diagram for this example looks like this:



`a` and `b` have the same value but do not refer to the same object.

There is one other important thing to notice about this reference diagram. The variable `a` is a reference to a **collection of references**. Those references actually refer to the integer values in the list. In other words, a list is a collection of references to objects. Interestingly, even though `a` and `b` are two different lists (two different collections of references), the integer object 81 is shared by both. Like strings, integers are also immutable so Python optimizes and lets everyone share the same object for some commonly used small integers.

Here is the example in codelens. Pay particular attention to the id values.

```

a = [81, 82, 83]
b = [81, 82, 83]

print(a is b)
print(a == b)

```

## 10.11 Aliasing

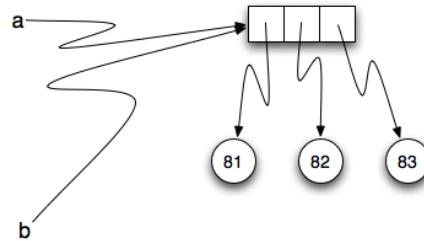
Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```

a = [81, 82, 83]
b = a
print(a is b)

```

In this case, the reference diagram looks like this:



Because the same list has two different names, `a` and `b`, we say that it is **aliased**. Changes made with one alias affect the other. In the codelens example below, you can see that `a` and `b` refer to the same list after executing the assignment statement `b = a`.

```

a = [81, 82, 83]
b = [81, 82, 83]

print(a == b)

```

```

print(a is b)

b = a
print(a == b)
print(a is b)

b[0] = 5
print(a)

```

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects, there's no problem. That's why Python is free to alias strings and integers when it sees an opportunity to economize.

#### Check your understanding

**Checkpoint 10.11.1** What is printed by the following statements?

```

alist = [4, 2, 8, 6, 5]
blist = alist
blist[3] = 999
print(alist)

```

- A. [4, 2, 8, 6, 5]
- B. [4, 2, 8, 999, 5]

**Checkpoint 10.11.2**

Consider the following lists:

```
| list1=[1,100,1000] list2=[1,100,1000] list3=list1
```

Which statements will output **True**? (Select **all** that apply).

- A. print(list1 == list2)
- B. print(list1 is list2)
- C. print(list1 is list3)
- D. print(list2 is not list3)
- E. print(list2 != list3)

## 10.12 Cloning Lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word *copy*.

The easiest way to clone a list is to use the slice operator.

Taking any slice of **a** creates a new list. In this case the slice happens to consist of the whole list.

```

a = [81, 82, 83]

b = a[:]      # make a clone using slice
print(a == b)
print(a is b)

b[0] = 5

```

```

print(a)
print(b)

```

Now we are free to make changes to `b` without worrying about `a`. Again, we can clearly see in codelens that `a` and `b` are entirely different list objects.

## 10.13 Repetition and References

We have already seen the repetition operator working on strings as well as lists. For example,

```

origlist = [45, 76, 34, 55]
print(origlist * 3)

```

With a list, the repetition operator creates copies of the references. Although this may seem simple enough, when we allow a list to refer to another list, a subtle problem can arise.

Consider the following extension on the previous example.

```

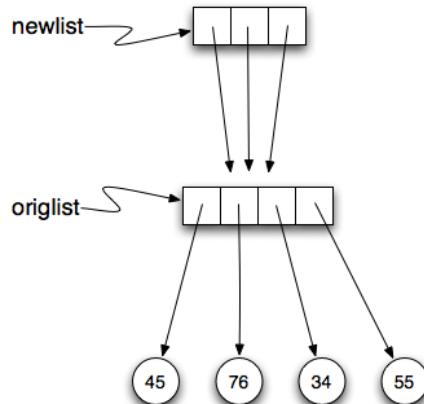
origlist = [45, 76, 34, 55]
print(origlist * 3)

newlist = [origlist] * 3

print(newlist)

```

`newlist` is a list of three references to `origlist` that were created by the repetition operator. The reference diagram is shown below.



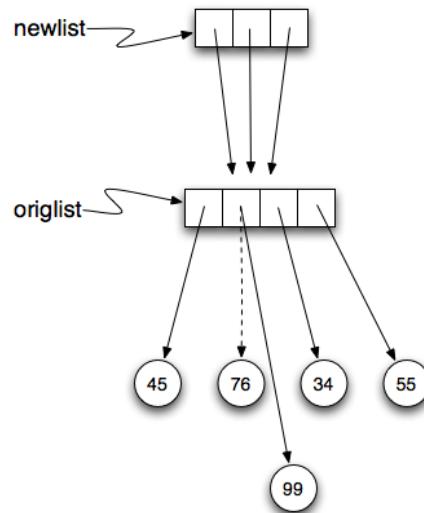
Now, what happens if we modify a value in `origlist`.

```

origlist = [45, 76, 34, 55]
newlist = [origlist] * 3
print(newlist)
origlist[1] = 99
print(newlist)

```

`newlist` shows the change in three places. This can easily be seen by noting that in the reference diagram, there is only one `origlist`, so any changes to it appear in all three references from `newlist`.



Here is the same example in codeLens. Step through the code paying particular attention to the result of executing the assignment statement `origlist[1] = 99`.

```

origlist = [45, 76, 34, 55]
newlist = [origlist] * 3
print(newlist)
origlist[1] = 99
print(newlist)

```

#### Check your understanding

**Checkpoint 10.13.1** What is printed by the following statements?

```

alist = [4, 2, 8, 6, 5]
blist = alist * 2
blist[3] = 999
print(alist)

```

A. [4, 2, 8, 999, 5, 4, 2, 8, 6, 5]

B. [4, 2, 8, 999, 5]

C. [4, 2, 8, 6, 5]

**Checkpoint 10.13.2** What is printed by the following statements?

```

alist = [4, 2, 8, 6, 5]
blist = [alist] * 2
alist[3] = 999
print(blist)

```

A. [4, 2, 8, 999, 5, 4, 2, 8, 999, 5]

B. [[4, 2, 8, 999, 5], [4, 2, 8, 999, 5]]

C. [4, 2, 8, 6, 5]

D. [[4, 2, 8, 999, 5], [4, 2, 8, 6, 5]]

## 10.14 List Methods

The dot operator can also be used to access built-in methods of list objects. `append` is a list method which adds the argument passed to it to the end of the list. Continuing with this example, we show several other list methods. Many of them are easy to understand.

```

mylist = []
mylist.append(5)
mylist.append(27)
mylist.append(3)
mylist.append(12)
print(mylist)

mylist.insert(1, 12)           # Insert 12 at pos 1, shift
                             other items up
print(mylist)
print(mylist.count(12))       # How many times is 12 in
                             mylist?

print(mylist.index(3))         # Find index of first 3 in
                             mylist
print(mylist.count(5))

mylist.reverse()
print(mylist)

mylist.sort()
print(mylist)

mylist.remove(5)              # Removes the second 12 in the
                             list
print(mylist)

lastitem = mylist.pop()        # Removes and returns the last
                             item of the list
print(lastitem)
print(mylist)

```

There are two ways to use the `pop` method. The first, with no parameter, will remove and return the last item of the list. If you provide a parameter for the position, `pop` will remove and return the item at that position. Either way the list is changed.

The following table provides a summary of the list methods shown above. The column labeled result gives an explanation as to what the return value is as it relates to the new value of the list. The word **mutator** means that the list is changed by the method but nothing is returned (actually `None` is returned). A **hybrid** method is one that not only changes the list but also returns a value as its result. Finally, if the result is simply a return, then the list is unchanged by the method.

Be sure to experiment with these methods to gain a better understanding of what they do.

**Table 10.14.1**

Method	Parameters	Result	Description
append	item	mutator	Adds a new item to the end of a list
insert	position, item	mutator	Inserts a new item at the position given
pop	none	hybrid	Removes and returns the last item
pop	position	hybrid	Removes and returns the item at position
sort	none	mutator	Modifies a list to be sorted
reverse	none	mutator	Modifies a list to be in reverse order
index	item	return idx	Returns the position of first occurrence of item
count	item	return ct	Returns the number of occurrences of item
remove	item	mutator	Removes the first occurrence of item

Details for these and others can be found in the [Python Documentation](#)<sup>1</sup>.

It is important to remember that methods like `append`, `sort`, and `reverse` all return `None`. This means that re-assigning `mylist` to the result of sorting `mylist` will result in losing the entire list. Calls like these will likely never appear as part of an assignment statement (see line 8 below).

```
mylist = []
mylist.append(5)
mylist.append(27)
mylist.append(3)
mylist.append(12)
print(mylist)

mylist = mylist.sort()    #probably an error
print(mylist)
```

### Check your understanding

**Checkpoint 10.14.2** What is printed by the following statements?

```
alist = [4, 2, 8, 6, 5]
alist.append(True)
alist.append(False)
print(alist)
```

- A. [4, 2, 8, 6, 5, False, True]
- B. [4, 2, 8, 6, 5, True, False]
- C. [True, False, 4, 2, 8, 6, 5]

**Checkpoint 10.14.3** What is printed by the following statements?

```
alist = [4, 2, 8, 6, 5]
alist.insert(2, True)
alist.insert(0, False)
print(alist)
```

- A. [False, 4, 2, True, 8, 6, 5]
- B. [4, False, True, 2, 8, 6, 5]
- C. [False, 2, True, 6, 5]

**Checkpoint 10.14.4** What is printed by the following statements?

---

<sup>1</sup><http://docs.python.org/py3k/library/stdtypes.html#sequence-types-str-bytes-bytearray-list-tuple-range>

```

alist = [4, 2, 8, 6, 5]
temp = alist.pop(2)
temp = alist.pop()
print(alist)

```

- A. [4, 8, 6]
- B. [2, 6, 5]
- C. [4, 2, 6]

**Checkpoint 10.14.5** What is printed by the following statements?

```

alist = [4, 2, 8, 6, 5]
alist = alist.pop(0)
print(alist)

```

- A. [2, 8, 6, 5]
- B. [4, 2, 8, 6, 5]
- C. 4
- D. None

**Note 10.14.6** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 10.15 The Return of L-Systems

Let's return to the L-systems we introduced in the previous chapter and introduce a very interesting new feature that requires the use of lists.

Suppose we have the following grammar:

```

X
X --> F[-X]+X
F --> FF

```

This L-system looks very similar to the old L-system except that we've added one change. We've added the characters '[' and ']'. The meaning of these characters adds a very interesting new dimension to our L-Systems. The '[' character indicates that we want to save the state of our turtle, namely its position and its heading so that we can come back to this position later. The ']' tells the turtle to warp to the most recently saved position. The way that we will accomplish this is to use lists. We can save the heading and position of the turtle as a list of 3 elements. [heading x y] The first index position in the list holds the heading, the second index position in the list holds the x coordinate, and the third index position holds the y coordinate.

Now, if we create an empty list and every time we see a '[' we append the list that contains [heading, x, y] we create a history of saved places the turtle has been where the most recently saved location will always be at the end of the list. When we find a ']' in the string we use the pop function to remove the the most recently appended information.

Let's modify our `drawLsystem` function to begin to implement this new behavior.

```

import turtle

def drawLsystem(aTurtle, instructions, angle, distance):
    savedInfoList = []

```

```

for cmd in instructions:
    if cmd == 'F':
        aTurtle.forward(distance)
    elif cmd == 'B':
        aTurtle.backward(distance)
    elif cmd == '+':
        aTurtle.right(angle)
    elif cmd == '-':
        aTurtle.left(angle)
    elif cmd == '[':
        savedInfoList.append([aTurtle.heading(),
                             aTurtle.xcor(), aTurtle.ycor()])
        print(savedInfoList)
    elif cmd == ']':
        newInfo = savedInfoList.pop()
        print(newInfo)
        print(savedInfoList)

t = turtle.Turtle()
inst = "FF[-F[-X]+X]+F[-X]+X"
drawLsystem(t, inst, 60, 20)

```

When we run this example we can see that the picture is not very interesting, but notice what gets printed out, and how the saved information about the turtle gets added and removed from the end of the list. In the next example we'll make use of the information from the list to save and restore the turtle's position and heading when needed. We'll use a longer example here so you get an idea of what the kind of drawing the L-System can really make.

```

import turtle

def drawLsystem(aTurtle, instructions, angle, distance):
    savedInfoList = []
    for cmd in instructions:
        if cmd == 'F':
            aTurtle.forward(distance)
        elif cmd == 'B':
            aTurtle.backward(distance)
        elif cmd == '+':
            aTurtle.right(angle)
        elif cmd == '-':
            aTurtle.left(angle)
        elif cmd == '[':
            savedInfoList.append([aTurtle.heading(),
                                  aTurtle.xcor(), aTurtle.ycor()])
            print(savedInfoList)
        elif cmd == ']':
            newInfo = savedInfoList.pop()
            aTurtle.setheading(newInfo[0])
            aTurtle.setposition(newInfo[1], newInfo[2])

    t = turtle.Turtle()
    inst =
        "FFFFFFF(((((((((F))))))))[-F(((((((((F))))))))[-F(((((((((F))))))))[-F(((F))))]]"
    t.setposition(0, -200)
    t.left(90)
    drawLsystem(t, inst, 30, 2)

```

Rather than use the `inst` string supplied here, use the code from the string chapter, and write your own `applyRules` function to implement this L-system.

This example only uses 6 expansions. Try it out with a larger number of expansions. You may also want to try this example with different values for the angle and distance parameters.

## 10.16 Append versus Concatenate

The `append` method adds a new item to the end of a list. It is also possible to add a new item to the end of a list by using the concatenation operator. However, you need to be careful.

Consider the following example. The original list has 3 integers. We want to add the word “cat” to the end of the list.

```
origlist = [45, 32, 88]
origlist.append("cat")
```

Here we have used `append` which simply modifies the list. In order to use concatenation, we need to write an assignment statement that uses the accumulator pattern:

```
origlist = origlist + ["cat"]
```

Note that the word “cat” needs to be placed in a list since the concatenation operator needs two lists to do its work.

```
origlist = [45, 32, 88]
origlist = origlist + ["cat"]
```

It is also important to realize that with `append`, the original list is simply modified. On the other hand, with concatenation, an entirely new list is created. This can be seen in the following codelens example where `newlist` refers to a list which is a copy of the original list, `origlist`, with the new item “cat” added to the end. `origlist` still contains the three values it did before the concatenation. This is why the assignment operation is necessary as part of the accumulator pattern.

```
origlist = [45, 32, 88]
newlist = origlist + ["cat"]
```

### Check your understanding

**Checkpoint 10.16.1** What is printed by the following statements?

```
alist = [4, 2, 8, 6, 5]
alist = alist + 999
print(alist)
```

- A. [4, 2, 8, 6, 5, 999]
- B. Error, you cannot concatenate a list with an integer.

## 10.17 Lists and `for` loops

It is also possible to perform **list traversal** using iteration by item as well as iteration by index.

```
fruits = ["apple", "orange", "banana", "cherry"]
```

```

for afruit in fruits:      # by item
    print(afruit)

```

It almost reads like natural language: For (every) fruit in (the list of) fruits, print (the name of the) fruit.

We can also use the indices to access the items in an iterative fashion.

```

fruits = ["apple", "orange", "banana", "cherry"]

for position in range(len(fruits)):      # by index
    print(fruits[position])

```

In this example, each time through the loop, the variable `position` is used as an index into the list, printing the `position`-eth element. Note that we used `len` as the upper bound on the range so that we can iterate correctly no matter how many items are in the list.

Any sequence expression can be used in a `for` loop. For example, the `range` function returns a sequence of integers.

```

for number in range(20):
    if number % 3 == 0:
        print(number)

```

This example prints all the multiples of 3 between 0 and 19.

Since lists are mutable, it is often desirable to traverse a list, modifying each of its elements as you go. The following code squares all the numbers from 1 to 5 using iteration by position.

```

numbers = [1, 2, 3, 4, 5]
print(numbers)

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

print(numbers)

```

Take a moment to think about `range(len(numbers))` until you understand how it works. We are interested here in both the *value* and its *index* within the list, so that we can assign a new value to it.

### Check your understanding

**Checkpoint 10.17.1** What is printed by the following statements?

```

alist = [4, 2, 8, 6, 5]
blist = []
for item in alist:
    blist.append(item+5)
print(blist)

```

- A. [4, 2, 8, 6, 5]
- B. [4, 2, 8, 6, 5, 5]
- C. [9, 7, 13, 11, 10]
- D. Error, you cannot concatenate inside an append.

## 10.18 The Accumulator Pattern with Lists

### 10.18.1 Introduction

Remember the [Section 6.5](#)? Many algorithms involving lists make use of this pattern to process the items in a list and compute a result. In this section, we'll explore the use of the accumulator pattern with lists.

Let's take the problem of adding up all of the items in a list. The following program computes the sum of a list of numbers.

```
sum = 0
for num in [1, 3, 5, 7, 9]:
    sum = sum + num
print(sum)
```

The program begins by defining an accumulator variable, `sum`, and initializing it to 0 (line 1).

Next, the program iterates over the list (lines 2-3), and updates the sum on each iteration by adding an item from the list (line 3). When the loop is finished, `sum` has accumulated the sum of all of the items in the list.

Take a moment to step through this program using CodeLens to see how it works. It's important to grasp the basic techniques.

Sometimes when we're accumulating, we don't want to add to our accumulator every time we iterate. Consider, for example, the following program which counts the number of names with more than 3 letters.

```
long_names = 0
for name in ["Joe", "Sally", "Amy", "Brad"]:
    if len(name) > 3:
        long_names += 1
print(long_names)
```

Here, we **initialize** the accumulator variable to be zero on line 1.

We **iterate** through the sequence (line 2).

The **update** step happens in two parts. First, we check to see if the name is longer than 3 letters. If so, then we increment the accumulator variable `long_names` (on line 4) by adding one to it.

At the end, we have accumulated the total number of long names.

We can use conditionals to also count if particular items are in a string or list. The following code finds all occurrences of vowels in a string.

```
s = "what\u00e9if\u00e9we\u00e9went\u00e9to\u00e9the\u00e9zoo"
num_vowels = 0
for i in s:
    if i in ['a', 'e', 'i', 'o', 'u']:
        num_vowels += 1
print(num_vowels)
```

We can also use `==` to execute a similar operation. Here, we'll check to see if the character we are iterating over is an “o”. If it is an “o” then we will update our counter.

### 10.18.2 Accumulating the Max Value

We can also use the accumulation pattern with conditionals to find the maximum or minimum value. Instead of continuing to build up the accumulator value like we have when counting or finding a sum, we can reassign the accumulator variable to a different value.

The following example shows how we can get the maximum value from a list of integers.

```
nums = [9, 3, 8, 11, 5, 29, 2]
best_num = 0
for n in nums:
    if n > best_num:
        best_num = n
print(best_num)
```

Here, we initialize `best_num` to zero, assuming that there are no negative numbers in the list.

In the for loop, we check to see if the current value of `n` is greater than the current value of `best_num`. If it is, then we want to **update** `best_num` so that it now is assigned the higher number. Otherwise, we do nothing and continue the for loop.

You may notice that the current structure could be a problem. If the numbers were all negative what would happen to our code? What if we were looking for the smallest number but we initialized `best_num` with zero? To get around this issue, we can initialize the accumulator variable using one of the numbers in the list.

```
nums = [9, 3, 8, 11, 5, 29, 2]
best_num = nums[0]
for n in nums:
    if n > best_num:
        best_num = n
print(best_num)
```

The only thing we changed was the value of `best_num` on line 2 so that the value of `best_num` is the first element in `nums`, but the result is still the same!

### 10.18.3 Accumulating a String Result

The accumulator pattern can be used to convert a list of items to a string.

Consider the following program:

```
scores = [85, 95, 70]
result = ''
for score in scores:
    result = result + str(score) + ','
print("The scores are " + result)
```

Here, the accumulator variable is `result`. Each time through the loop, the program concatenates the current contents of `result` with the comma separator and a score from the list, and updates the `result` with the new value. Use CodeLens to step through this example to see it in action.

The output of the program has some undesirable formatting problems: there is a trailing comma instead of a period, and there are no spaces between the items. The next activity lets you work to correct those problems.

**Checkpoint 10.18.1** Let's work to improve the formatting of the sentence produced by the program above. Revise the following code so that it outputs the sentence:

The scores are 85, 95, and 70.

```

scores = [85, 95, 70]
result = ''
for score in scores:
    result = result + str(score) + ','
print("The scores are " + result)
=====
from unittest.gui import TestCaseGui
class myTests(TestCaseGui):
    def testOne(self):
        self.assertEqual(self.getOutput().rstrip('\n'),
                        'The scores are 85,95, and 70.', "Output should be 'The scores are 85,95, and 70.'")
myTests().main()

```

**Check your understanding****Checkpoint 10.18.2** What is printed by the following statements?

```

s = "We are learning!"
x = 0
for i in s:
    if i in ['a', 'b', 'c', 'd', 'e']:
        x += 1
print(x)

```

- A. 2
- B. 5
- C. 0
- D. There is an error in the code so it cannot run.

**Checkpoint 10.18.3** What is printed by the following statements?

```

list= [5, 2, 1, 4, 9, 10]
min_value = 0
for item in list:
    if item < min_value:
        min_value = item
print(min_value)

```

- A. 10
- B. 1
- C. 0
- D. There is an error in the code so it cannot run.

**Checkpoint 10.18.4 Challenge** For each word in `words`, add ‘d’ to the end of the word if the word ends in “e” to make it past tense. Otherwise, add ‘ed’ to make it past tense. Save these past tense words to a list called `past_tense`.

```

words = ["adopt", "bake", "beam", "confide", "grill",
         "plant", "time", "wave", "wish"]
=====
from unittest.gui import TestCaseGui
class myTests(TestCaseGui):
    def testNine(self):
        self.assertEqual(past_tense, ['adopted', 'baked',
                                     'beamed', 'confided', 'grilled', 'planted',
                                     'timed', 'waved', 'wished'], "Testing that the
                                     past_tense list is correct.")
        self.assertIn("else", self.getEditorText(),
                      "Testing output (Don't worry about actual and
                      expected values).")
        self.assertIn("for", self.getEditorText(),
                      "Testing output (Don't worry about actual and
                      expected values).")
myTests().main()

```

## 10.19 Using Lists as Parameters

Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**. Passing a list as an argument actually passes a reference to the list, not a copy of the list. Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing. For example, the function below takes a list as an argument and multiplies each element in the list by 2:

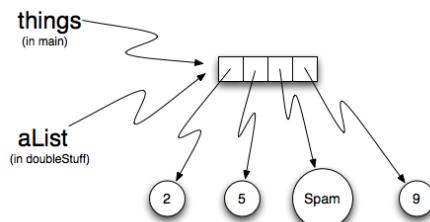
```

def doubleStuff(aList):
    """Overwrite each element in aList with double its
    value."""
    for position in range(len(aList)):
        aList[position] = 2 * aList[position]

things = [2, 5, 9]
print(things)
doubleStuff(things)
print(things)

```

The parameter `aList` and the variable `things` are aliases for the same object.



Since the list object is shared by two references, there is only one copy. If a function modifies the elements of a list parameter, the caller sees the change since the change is occurring to the original.

This can be easily seen in codelens. Note that after the call to `doubleStuff`, the formal parameter `aList` refers to the same object as the actual parameter `things`. There is only one copy of the list object itself.

```

def doubleStuff(aList):
    """Overwrite each element in aList with double its value."""
    for position in range(len(aList)):
        aList[position] = 2 * aList[position]

things = [2, 5, 9]

doubleStuff(things)

```

## 10.20 Pure Functions

A **pure function** does not produce side effects. It communicates with the calling program only through parameters (which it does not modify) and a return value. Here is the `doubleStuff` function from the previous section written as a pure function. To use the pure function version of `double_stuff` to modify `things`, you would assign the return value back to `things`.

```

def doubleStuff(a_list):
    """Return a new list in which contains doubles of the elements in a_list."""
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)
    return new_list

things = [2, 5, 9]
print(things)
things = doubleStuff(things)
print(things)

```

Once again, codelens helps us to see the actual references and objects as they are passed and returned.

```

def doubleStuff(a_list):
    """Return a new list in which contains doubles of the elements in a_list."""
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)
    return new_list

things = [2, 5, 9]
things = doubleStuff(things)

```

## 10.21 Which is Better?

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. Nevertheless, modifiers are convenient at times, and in some cases, functional programs are less efficient.

In general, we recommend that you write pure functions whenever it is reasonable to do so and resort to modifiers only if there is a compelling advantage. This approach might be called a *functional programming style*.

## 10.22 Functions that Produce Lists

The pure version of `doubleStuff` above made use of an important **pattern** for your toolbox. Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
initialize a result variable to be an empty list
loop
    create a new element
    append it to result
return the result
```

Let us show another use of this pattern. Assume you already have a function `is_prime(x)` that can test if `x` is prime. Now, write a function to return a list of all prime numbers less than `n`:

```
def primes_upto(n):
    """ Return a list of all prime numbers less than n. """
    result = []
    for i in range(2, n):
        if is_prime(i):
            result.append(i)
    return result
```

## 10.23 List Comprehensions

The previous example creates a list from a sequence of values based on some selection criteria. An easy way to do this type of processing in Python is to use a **list comprehension**. List comprehensions are concise ways to create lists. The general syntax is:

`[<expression> for <item> in <sequence> if <condition>]`

where the if clause is optional. For example,

```
mylist = [1, 2, 3, 4, 5]

yourlist = [item ** 2 for item in mylist]

print(yourlist)
```

The expression describes each element of the list that is being built. The `for` clause iterates through each item in a sequence. The items are filtered by the if clause if there is one. In the example above, the `for` statement lets `item` take on all the values in the list `mylist`. Each item is then squared before it is added to the list that is being built. The result is a list of squares of the values in `mylist`.

To write the `primes_upto` function we will use the `is_prime` function to filter the sequence of integers coming from the `range` function. In other words, for every integer from 2 up to but not including `n`, if the integer is prime, keep it in the list.

```
def primes_upto(n):
    """Return a list of all prime numbers less than n
    using a list comprehension."""

    result = [num for num in range(2, n) if is_prime(num)]
    return result
```

**Note 10.23.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

#### Check your understanding

**Checkpoint 10.23.2** What is printed by the following statements?

```
alist = [4, 2, 8, 6, 5]
blist = [num*2 for num in alist if num%2==1]
print(blist)
```

- A. [4,2,8,6,5]
- B. [8,4,16,12,10]
- C. 10
- D. [10].

## 10.24 Nested Lists

A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list. If we print(nested[3]), we get [10, 20]. To extract an element from the nested list, we can proceed in two steps. First, extract the nested list, then extract the item of interest. It is also possible to combine those steps using bracket operators that evaluate from left to right.

```
nested = ["hello", 2.0, 5, [10, 20]]
innerlist = nested[3]
print(innerlist)
item = innerlist[1]
print(item)

print(nested[3][1])
```

#### Check your understanding

**Checkpoint 10.24.1** What is printed by the following statements?

```
alist = [ [4, [True, False], 6, 8], [888, 999] ]
if alist[0][1][0]:
    print(alist[1][0])
else:
    print(alist[1][1])
```

- A. 6
- B. 8
- C. 888
- D. 999

## 10.25 Strings and Lists

Two of the most useful methods on strings involve lists of strings. The `split` method breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary.

```
song = "The\u00a9rain\u00a9in\u00a9Spain..."
wds = song.split()
```

```
|   print(wds)
```

An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries. The following example uses the string `ai` as the delimiter:

```
| song = "The\u00a9rain\u00a9in\u00a9Spain..."  
wds = song.split('ai')  
print(wds)
```

Notice that the delimiter doesn't appear in the result.

The inverse of the `split` method is `join`. You choose a desired **separator** string, (often called the *glue*) and join the list with the glue between each of the elements.

```
| wds = ["red", "blue", "green"]  
glue = ';'!  
s = glue.join(wds)  
print(s)  
print(wds)  
  
print("***".join(wds))  
print("") .join(wds))
```

The list that you glue together (`wds` in this example) is not modified. Also, you can use empty glue or multi-character strings as glue.

### Check your understanding

**Checkpoint 10.25.1** What is printed by the following statements?

```
| myname = "Edgar\u00a9Allan\u00a9Poe"  
namelist = myname.split()  
init = ""  
for fname in namelist:  
    init = init + fname[0]  
print(init)
```

- A. Poe
- B. EdgarAllanPoe
- C. EAP
- D. William Shakespeare

## 10.26 List - Type Conversion Function

Python has a built-in type conversion function called `list` that tries to turn whatever you give it into a list. For example, try the following:

```
| xs = list("Crunchy\u00a9Frog")  
print(xs)
```

The string “Crunchy Frog” is turned into a list by taking each character in the string and placing it in a list. In general, any sequence can be turned into a list using this function. The result will be a list containing the elements in the original sequence. It is not legal to use the `list` conversion function on any argument that is not a sequence.

It is also important to point out that the `list` conversion function will place each element of the original sequence in the new list. When working with strings, this is very different than the result of the `split` method. Whereas

`split` will break a string into a list of “words”, `list` will always break it into a list of characters.

## 10.27 Tuples and Mutability

So far you have seen two types of sequential collections: strings, which are made up of characters; and lists, which are made up of elements of any type. One of the differences we noted is that the elements of a list can be modified, but the characters in a string cannot. In other words, strings are **immutable** and lists are **mutable**.

A **tuple**, like a list, is a sequence of items of any type. Unlike lists, however, tuples are immutable. Syntactically, a tuple is a comma-separated sequence of values. Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009,
         "Actress", "Atlanta", "Georgia")
```

Tuples are useful for representing what other languages often call *records* — some related information that belongs together, like your student record. There is no description of what each of these *fields* means, but we can guess. A tuple lets us “chunk” together related information and use it as a single thing.

Tuples support the same sequence operations as strings and lists. For example, the index operator selects an element from a tuple. A tuple can be the sequence in a for-loop.

As with strings, if we try to use item assignment to modify one of the elements of the tuple, we get an error.

```
julia[0] = 'X'
TypeError: 'tuple' object does not support item assignment
```

Of course, even if we can’t modify the elements of a tuple, we can make a variable reference a new tuple holding different information. To construct the new tuple, it is convenient that we can slice parts of the old tuple and join up the bits to make the new tuple. So `julia` has a new recent film, and we might want to change her tuple. We can easily slice off the parts we want and concatenate them with the new tuple.

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009,
         "Actress", "Atlanta", "Georgia")
print(julia[2])
print(julia[2:6])

print(len(julia))

for field in julia:
    print(field)

julia = julia[:3] + ("Eat", "Pray", "Love", 2010) + julia[5:]
print(julia)
```

To create a tuple with a single element (but you’re probably not likely to do that too often), we have to include the final comma, because without the final comma, Python treats the (5) below as an integer in parentheses:

```
tup = (5,)
print(type(tup))
```

```
| x = (5)
| print(type(x))
```

## 10.28 Tuple Assignment

Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

```
| (name, surname, birth_year, movie, movie_year, profession,
|   birth_place) = julia
```

This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap `a` and `b`:

```
| temp = a
| a = b
| b = temp
```

Tuple assignment solves this problem neatly:

```
| (a, b) = (b, a)
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Naturally, the number of variables on the left and the number of values on the right have to be the same.

```
| >>> (a, b, c, d) = (1, 2, 3)
| ValueError: need more than 3 values to unpack
```

## 10.29 Tuples as Return Values

Functions can return tuples as return values. This is very useful — we often want to know some batsman's highest and lowest score, or we want to find the mean and the standard deviation, or we want to know the year, the month, and the day, or if we're doing some ecological modeling we may want to know the number of rabbits and the number of wolves on an island at a given time. In each case, a function (which can only return a single value), can create a single tuple holding multiple elements.

For example, we could write a function that returns both the area and the circumference of a circle of radius `r`.

```
| def circleInfo(r):
|   """Return (circumference, area) of a circle of radius r"""
|   c = 2 * 3.14159 * r
|   a = 3.14159 * r * r
|   return (c, a)
|
| print(circleInfo(10))
```

**Note 10.29.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 10.30 Glossary

### Glossary

**aliases.** Multiple variables that contain references to the same object.

**clone.** To create a new object that has the same value as an existing object. Copying a reference to an object creates an alias but doesn't clone the object.

**delimiter.** A character or string used to indicate where a string should be split.

**element.** One of the values in a list (or other sequence). The bracket operator selects elements of a list.

**index.** An integer variable or value that indicates an element of a list.

**list.** A collection of objects, where each object is identified by an index. Like other types `str`, `int`, `float`, etc. there is also a `list` type-converter function that tries to turn its argument into a list.

**list traversal.** The sequential accessing of each element in a list.

**modifier.** A function which changes its arguments inside the function body. Only mutable types can be changed by modifiers.

**mutable data type.** A data type in which the elements can be modified. All mutable types are compound types. Lists are mutable data types; strings are not.

**nested list.** A list that is an element of another list.

**object.** A thing to which a variable can refer.

**pattern.** A sequence of statements, or a style of coding something that has general applicability in a number of different situations. Part of becoming a mature Computer Scientist is to learn and establish the patterns and algorithms that form your toolkit. Patterns often correspond to your "mental chunking".

**pure function.** A function which has no side effects. Pure functions only make changes to the calling program through their return values.

**sequence.** Any of the data types that consist of an ordered collection of elements, with each element identified by an index.

**side effect.** A change in the state of a program made by calling a function that is not a result of reading the return value from the function. Side effects can only be produced by modifiers.

**tuple.** A sequential collection of items, similar to a list. Any python object can be an element of a tuple. However, unlike a list, tuples are immutable.

## 10.31 Exercises

1.

2. Create a list called `myList` with the following six items: 76, 92.3, "hello", True, 4, 76. Begin with the empty list shown below, and add 6 statements to add each item, one per item. The first three statements should use the `append` method to append the item to the list, and the last three statements should use concatenation.

```

myList = []

# Your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(myList,[76, 92.3, 'hello',
                               True, 4, 76],"myList\u00b7should\u00b7contain\u00b7the\u00b7
                               specified\u00b7items")
        self.assertIn(".append()", self.getEditorText(),
                      'append\u00b7method\u00b7must\u00b7be\u00b7used')

myTests().main()

```

3. Starting with the list of the previous exercise, write Python statements to do the following:

- a Append “apple” and 76 to the list.
- b Insert the value “cat” at position 3.
- c Insert the value 99 at the start of the list.
- d Find the index of “hello”.
- e Count the number of 76s in the list.
- f Remove the first occurrence of 76 from the list.
- g Remove True from the list using `pop` and `index`.

```

myList = [76, 92.3, 'hello', True, 4, 76]

# Your code here

```

4. Write a function called `average` that takes a list of numbers as a parameter and returns the average of the numbers.

```

def average(numlist):
    # Complete the function definition

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(int(average([1, 3, 5,
                                      7])),4,"average([1,\u00b73,\u00b75,\u00b77])\u00b7should\u00b7be\u00b74")
        self.assertEqual(int(average([9, 5,
                                      4])),6,"average([9,\u00b75,\u00b74])\u00b7should\u00b7be\u00b76")

myTests().main()

```

5. Write a Python function named `max` that takes a parameter containing a nonempty list of integers and returns the maximum value. (Note: there is a builtin function named `max` but pretend you cannot use it.)

```

def max(lst):
    # Complete the function

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(max([3, 31, 5, 7]), 31, "max([3, 31, 5, 7]) should be 31")
        self.assertEqual(max([3, 13, 51, 7]), 51, "max([3, 13, 51, 7]) should be 51")

myTests().main()

```

6. Write a function `sum_of_squares(xs)` that computes the sum of the squares of the numbers in the list `xs`. For example, `sum_of_squares([2, 3, 4])` should return  $4+9+16$  which is 29:

```

def sum_of_squares(xs):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(sum_of_squares([2,3,4]), 29, "Tested sum_of_squares on input [2,3,4]")
        self.assertEqual(sum_of_squares([0,1,-1]), 2, "Tested sum_of_squares on input [0,1,-1]")
        self.assertEqual(sum_of_squares([5,12,14]), 365, "Tested sum_of_squares on input [5,12,14]")

myTests().main()

```

7. Write a function to count how many odd numbers are in a list.

```

def countOdd(lst):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(countOdd([1,3,5,7,9]), 5, "Tested countOdd on input [1,3,5,7,9]")
        self.assertEqual(countOdd([1,2,3,4,5]), 3, "Tested countOdd on input [-1,-2,-3,-4,-5]")
        self.assertEqual(countOdd([2,4,6,8,10]), 0, "Tested countOdd on input [2,4,6,8,10]")
        self.assertEqual(countOdd([0,-1,12,-33]), 2, "Tested countOdd on input [0,-1,12,-33]")

myTests().main()

```

8. Sum up all the even numbers in a list.

```
def sumEven(lst):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(sumEven([1,3,5,7,9]),0,"Tested\u
            sumEven\uon\uinput\u[1,3,5,7,9]")
        self.assertEqual(sumEven([-1,-2,-3,-4,-5]),-6,"Tested\u
            sumEven\uon\uinput\u[-1,-2,-3,-4,-5]")
        self.assertEqual(sumEven([2,4,6,7,9]),12,"Tested\u
            sumEven\uon\uinput\u[2,4,6,7,9]")
        self.assertEqual(sumEven([0,1,12,33]),12,"Tested\u
            sumEven\uon\uinput\u[0,1,12,33]")

myTests().main()
```

9. Sum up all the negative numbers in a list.

```
def sumNegatives(lst):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(sumNegatives([-1,-2,-3,-4,-5]),-15,"Tested\u
            sumNegatives\uon\uinput\u[-1,-2,-3,-4,-5]")
        self.assertEqual(sumNegatives([1,-3,5,-7,9]),-10,"Tested\u
            sumNegatives\uon\uinput\u[1,-3,5,-7,9]")
        self.assertEqual(sumNegatives([-2,-4,6,-7,9]),-13,"Tested\u
            sumNegatives\uon\uinput\u[-2,-4,6,-7,9]")
        self.assertEqual(sumNegatives([0,1,2,3,4]),0,"Tested\u
            sumNegatives\uon\uinput\u[0,1,2,3,4]")

myTests().main()
```

10. Count how many words in a list have length 5.

```
def countWords(lst):
    # your code here
```

- 11.

12. Count how many words occur in a list up to and including the first occurrence of the word “sam”.

```
def count(lst):
    # your code here
```

13. Although Python provides us with many list methods, it is good practice and very instructive to think about how they are implemented. Implement a Python function that works like the following:

a count

b in

c reverse

d index

e insert

14. Write a function `replace(s, old, new)` that replaces all occurrences of `old` with `new` in a string `s`:

```
test(replace('Mississippi', 'i', 'I'), 'MIssIssIppI')

s = 'I love spom! Spom is my favorite food. Spom, spom, spom, yum!'
test(replace(s, 'om', 'am'),
     'I love spam! Spam is my favorite food. Spam, spam, spam, yum!')

test(replace(s, 'o', 'a'),
     'I lave spam! Spam is my favarite faad. Spam, spam, spam, yum!')
```

*Hint:* use the `split` and `join` methods.

```
def replace(s, old, new):
    # your code here

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(replace('Mississippi','i','I'),'MIssIssIppI',"Tested"
                         "replace\u00d7on\u00d7input\u00d7'Mississippi','i','I'")")
        self.assertEqual(replace('Bookkeeper','e','A'),'BookkAApAr',"Tested"
                         "failed\u00d7on\u00d7input\u00d7'Bookkeeper','e','A'")")
        self.assertEqual(replace('Deeded','e','q'),'Dqqdqd',"Tested"
                         "failed\u00d7on\u00d7input\u00d7'Deeded','e','q'")")

myTests().main()
```

15. Here are the rules for an L-system that creates something that resembles a common garden herb. Implement the following rules and try it. Use an angle of 25.7 degrees.

```
H
H --> HFX[+H][-H]
X --> X[-FFF][+FFF]FX
```

16. Here is another L-System. Use an Angle of 25.

```
F
F --> F[-F]F[+F]F
```

17. Create a list named `randlist` containing 100 random integers between 0 and 1000 (use iteration, append, and the random module).

```
=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(len(randlist),100,"randlist\u
            should\ucontain\u100\unumbers")

myTests().main()
```

# Chapter 11

## Files

### 11.1 Working with Data Files

So far, the data we have used in this book have all been either coded right into the program, or have been entered by the user. In real life data reside in files. For example the images we worked with in the image processing unit ultimately live in files on your hard drive. Web pages, and word processing documents, and music are other examples of data that live in files. In this short chapter we will introduce the Python concepts necessary to use data from files in our programs.

For our purposes, we will assume that our data files are text files—that is, files filled with characters. The Python programs that you write are stored as text files. We can create these files in any of a number of ways. For example, we could use a text editor to type in and save the data. We could also download the data from a website and then save it in a file. Regardless of how the file is created, Python will allow us to manipulate the contents.

In Python, we must **open** files before we can use them and **close** them when we are done with them. As you might expect, once a file is opened it becomes a Python object just like all other data. [Table 11.1.1](#) shows the functions and methods that can be used to open and close files.

**Table 11.1.1**

Method Name	Use	Explanation
open	open(filename,'r')	Open a file called filename and use it for reading. This will return a file variable.
open	open(filename,'w')	Open a file called filename and use it for writing. This will also return a file variable.
close	filevariable.close()	File use is complete.

### 11.2 Finding a File on your Disk

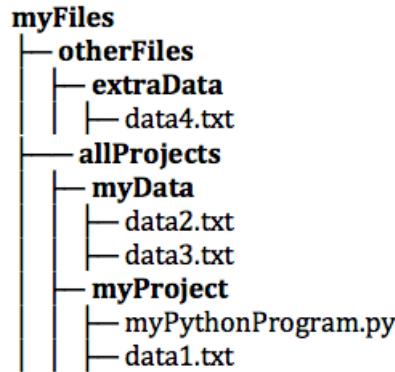
Opening a file requires that you, as a programmer, and Python agree about the location of the file on your disk. The way that files are located on disk is by their **path**. You can think of the filename as the short name for a file, and the path as the full name. For example on a Mac if you save the file `hello.txt` in your home directory the path to that file is `/Users/yourname/hello.txt`. On a Windows machine the path looks a bit different but the same principles are in use. For example on windows the path might be `C:\Users\yourname\My Documents\hello.txt`.

### Note 11.2.1 The History of Path Separators

Why is the path separator a / on Unix/Linux/MacOS systems and \ on Microsoft Windows systems? The concept of a hierarchy of folders was first developed in Unix. On a Unix command line a / is used to separate folder names in a file path and dashes are used to specify command line options, e.g., `path/to/file/myfile -long -reverse`. On a Windows system the / character is used for command line options, so the designers of Windows decided to use the \ for separating folder names in a file path, e.g., `path\to\file\myfile /long /reverse`. Using a \ to separate folder names in a path is problematic because the \ character is also used as an escape character for special characters, such as \n for a new line character. Bottom line, we will always use the / character to separate folder names in a path, and even on Windows system the file path will work just fine.

You can access files in sub-folders, also called directories, under your home directory by adding a slash and the name of the folder. For example, if you had a file called `hello.py` in a folder called `CS150` that is inside a folder called `PyCharmProjects` under your home directory, then the full name for the file `hello.py` is `/Users/yourname/PyCharmProjects/CS150/hello.py`. This is called an *absolute file path*. An *absolute file path* typically only works on a specific computer. Think about it for a second. What other computer in the world is going to have an *absolute file path* that starts with `/Users/yourname`?

If a file is not in the same folder as your python program, you need to tell the computer how to reach it. A *relative file path* starts from the folder that contains your python program and follows a computer's file hierarchy. A file hierarchy contains folders which contains files and other sub-folders. Specifying a sub-folder is easy – you simply specify the sub-folder's name. To specify a *parent* folder you use the special .. notation because every file and folder has one unique parent. You can use the .. notation multiple times in a file path to move multiple levels up a file hierarchy. Here is an example file hierarchy that contains multiple folders, files, and sub-folders. Folders in the diagram are displayed in **bold** type.



Using the example file hierarchy above, the program, `myPythonProgram.py` could access each of the data files using the following *relative file paths*:

- `data1.txt`
- `../myData/data2.txt`
- `../myData/data3.txt`
- `../../otherFiles/extraData/data4.txt`

Here's the important rule to remember: If your file and your Python program are in the same directory you can simply use the filename like this: `open('myfile.txt', 'r')`. If your file and your Python program are in different directories then you must refer to one or more directories, either in a *relative file path* to the file like this: `open('../myData/data3.txt', 'r')`, or in an *absolute file path* like `open('/users/bmiller/myFiles/allProjects/myData/data3.txt', 'r')`.

### 11.3 Reading a File

As an example, suppose we have a text file called `ccdata.txt` that contains the following data representing statistics about climate change. Although it would be possible to consider entering this data by hand each time it is used, you can imagine that it would be time-consuming and error-prone to do this. In addition, it is likely that there could be data from more sources and other years. The format of the data file is as follows:

```
Year, Global Average Temperature, Global Emission of CO2
```

To open this file, we would call the `open` function. The variable, `fileref`, now holds a reference to the file object returned by `open`. When we are finished with the file, we can close it by using the `close` method. After the file is closed any further attempts to use `fileref` will result in an error.

```
>>> fileref = open("ccdata.txt", "r")
>>>
>>> fileref.close()
>>>
```

### 11.4 Iterating over lines in a file

Recall the contents of the `ccdata.txt` file.

**Table 11.4.1**

Year	Global Average Temp. (Celcius)	Global Emissions CO2 (Giga-tons)
1990	55	10
1991	56	11
1992	57	12
1993	58	13
1994	59	14
1995	60	15
1996	61	16
1997	62	17
1998	63	18
1999	64	19
2000	65	20
2001	66	21
2002	67	22
2003	68	23
2004	69	24
2005	70	25
2006	71	26
2007	72	27
2008	73	28
2009	74	29
2010	75	30
2011	76	31
2012	77	32
2013	78	33
2014	79	34
2015	80	35
2016	81	36
2017	82	37
2018	83	38
2019	84	39
2020	85	40

We will now use this file as input in a program that will do some data processing. In the program, we will **read** each line of the file and print it with some additional text. Because text files are sequences of lines of text, we can use the `for` loop to iterate through each line of the file.

A **line** of a file is defined to be a sequence of characters up to and including a special character called the **newline** character. If you evaluate a string that contains a newline character you will see the character represented as `\n`. If you print a string that contains a newline you will not see the `\n`, you will just see its effects. When you are typing a Python program and you press the enter or return key on your keyboard, the editor inserts a newline character into your text at that point.

As the `for` loop iterates through each line of the file the loop variable will contain the current line of the file as a string of characters. The general pattern for processing each line of a text file is as follows:

```
for line in myFile:
    statement1
    statement2
    ...
    ...
```

To process all of our climate change data, we will use a *for* loop to iterate over the lines of the file. Using the `split` method, we can break each line into a list containing all the fields of interest about climate change. We can then take the values corresponding to year, global average temperature, and global emissions to construct a simple sentence.

```
ccfile = open("ccdata.txt", "r")

for aline in ccfile:
    values = aline.split()
    print('In', values[0], 'the average temp. was',
          values[1], '°C and CO2 emissions were', values[2],
          'gigatons.')

ccfile.close()
```

**Note 11.4.2** You can obtain a line from the keyboard with the `input` function, and you can process lines of a file. However “line” is used differently: With `input` Python reads through the newline you enter from the keyboard, but the newline ('\n') is *not* included in the line returned by `input`. It is dropped. When a line is taken from a file, the terminating newline *is* included as the last character (unless you are reading the final line of a file that happens to not have a newline at the end).

In the climate change example it is irrelevant whether the final line has a newline character at the end or not, since it would be stripped off by the `split` method call.

## 11.5 Alternative File Reading Methods

Again, recall the contents of the `ccdata.txt` file. In addition to the `for` loop, Python provides three methods to read data from the input file. The `readline` method reads one line from the file and returns it as a string. The string returned by `readline` will contain the newline character at the end. This method returns the empty string when it reaches the end of the file. The `readlines` method returns the contents of the entire file as a list of strings, where each item in the list represents one line of the file. It is also possible to read the entire file into a single string with `read`. [Table 11.5.1](#) summarizes these methods and the following session shows them in action.

Note that we need to reopen the file before each read so that we start from the beginning. Each file has a marker that denotes the current read position in the file. Any time one of the read methods is called the marker is moved to the character immediately following the last character returned. In the case of `readline` this moves the marker to the first character of the next line in the file. In the case of `read` or `readlines` the marker is moved to the end of the file.

```
>>> infile = open("ccdata.txt", "r")
>>> aline = infile.readline()
>>> aline
'1850\ -0.37\ 2.24E-7\n'
>>>
>>> infile = open("ccdata.txt", "r")
>>> linelist = infile.readlines()
>>> print(len(linelist))
18
```

```

>>> print(linelist[0:4])
['1850\~-0.37\2.24E-7\n',
 '1860\~-0.34\3.94E-7\n',
 '1870\~-0.28\6.6E-7\n',
 '1880\~-0.24\1.1\n']
>>>
>>> infile = open("ccdata.txt", "r")
>>> filestring = infile.read()
>>> print(len(filestring))
1282
>>> print(filestring[:256])
1850 -0.37 2.24E-7
1860 -0.34 3.94E-7
1870 -0.28 6.6E-7
1880 -0.24
>>>

```

**Table 11.5.1**

Method Name	Use	Explanation
write	filevar.write(astring)	Add astring to the end of the file. filevar must refer to a file
read(n)	filevar.read()	Reads and returns a string of n characters, or the entire file
readline(n)	filevar.readline()	Returns the next line of the file with all text up to and inclu
readlines(n)	filevar.readlines()	Returns a list of strings, each representing a single line of the

Now let's look at another method of reading our file using a `while` loop. This is important because many other programming languages do not support the `for` loop style for reading files but they do support the pattern we'll show you here.

```

infile = open("ccdata.txt", "r")
line = infile.readline()
while line:
    values = line.split()
    print('In', values[0], 'the\u00b7average\u00b7temp.\u00b7was',
          values[1], '\u00b0C\u00b7and\u00b7CO2\u00b7emmisions\u00b7were', values[2],
          'gigatons.')
    line = infile.readline()

infile.close()

```

There are several important things to notice in this code:

On line 2 we have the statement `line = infile.readline()`. We call this initial read the **priming read**. It is very important because the `while` condition needs to have a value for the `line` variable.

The `readline` method will return the empty string if there is no more data in the file. An empty string is an empty sequence of characters. When Python is looking for a Boolean condition, as in `while line:`, it treats an empty sequence type as `False`, and a non-empty sequence as `True`. Remember that a blank line in the file actually has a single character, the `\n` character (newline). So, the only way that a line of data from the file can be empty is if you are reading at the end of the file, and the `while` condition becomes `False`.

Finally, notice that the last line of the body of the `while` loop performs another `readline`. This statement will reassign the variable `line` to the next line of the file. It represents the change of state that is necessary for the iteration to function correctly. Without it, there would be an infinite loop processing the same line of data over and over.

## 11.6 Writing Text Files

One of the most commonly performed data processing tasks is to read data from a file, manipulate it in some way, and then write the resulting data out to a new data file to be used for other purposes later. To accomplish this, the `open` function discussed above can also be used to create a new file prepared for writing. Note in [Section 11.1](#) above that the only difference between opening a file for writing and opening a file for reading is the use of the '`w`' flag instead of the '`r`' flag as the second parameter. When we open a file for writing, a new, empty file with that name is created and made ready to accept our data. As before, the function returns a reference to the new file object.

[Section 11.5](#) above shows one additional file method that we have not used thus far. The `write` method allows us to add data to a text file. Recall that text files contain sequences of characters. We usually think of these character sequences as being the lines of the file where each line ends with the newline `\n` character. Be very careful to notice that the `write` method takes one parameter, a string. When invoked, the characters of the string will be added to the end of the file. This means that it is the programmers job to include the newline characters as part of the string if desired.

As an example, consider the `ccdata.txt` file once again. Assume that we have been asked to provide a file consisting of only the global emission and the year of this climate change. In addition, the year should come first followed by the global emmision, separated by space.

To construct this file, we will approach the problem using a similar algorithm as above. After opening the file, we will iterate through the lines, break each line into its parts, choose the parts that we need, and then output them. Eventually, the output will be written to a file.

The program below solves part of the problem. Notice that it reads the data and creates a string consisting of the year of the climate change followed by the global emission. In this example, we simply print the lines as they are created.

```
infile = open("ccdata.txt", "r")
aline = infile.readline()
print("Year\tEmission\n")
while aline:
    items = aline.split()
    dataline = items[0] + '\t' + items[2]
    print(dataline)
    aline = infile.readline()

infile.close()
```

When we run this program, we see the lines of output on the screen. Once we are satisfied that it is creating the appropriate output, the next step is to add the necessary pieces to produce an output file and write the data lines to it. To start, we need to open a new output file by adding another call to the `open` function, `outfile = open("emissiondata.txt",'w')`, using the '`w`' flag. We can choose any file name we like. If the file does not exist, it will be created. However, if the file does exist, it will be reinitialized as empty and you will lose any previous contents.

Once the file has been created, we just need to call the `write` method passing the string that we wish to add to the file. In this case, the string is already being printed so we will just change the `print` into a call to the `write` method. However, there is one additional part of the data line that we need to include. The newline character needs to be concatenated to the end of the

line. The entire line now becomes `outfile.write(dataline + '\n')`. We also need to close the file when we are done.

The complete program is shown below.

```
infile = open("ccdata.txt", "r")
outfile = open("emissiondata.txt", "w")

aline = infile.readline()
outfile.write("Year\tEmission\n")
while aline:
    items = aline.split()
    dataline = items[0] + '\t' + items[2]
    outfile.write(dataline + '\n')
    aline = infile.readline()

infile.close()
outfile.close()
```

## 11.7 With Statements

**Note 11.7.1** This section is a bit of an advanced topic and can be easily skipped. But with statements are becoming very common and it doesn't hurt to know about them in case you run into one in the wild.

Now that you have seen and practiced a bit with opening and closing files, there is another mechanism that Python provides for us that cleans up the often forgotten close. Forgetting to close a file does not necessarily cause a runtime error in the kinds of programs you typically write in an introductory CS course. However if you are writing a program that may run for days or weeks at a time that does a lot of file reading and writing you may run into trouble.

In version 2.5 Python introduced the concept of a context manager. The context manager automates the process of doing common operations at the start of some task, as well as automating certain operations at the end of some task. In the context of reading and writing a file, the normal operation is to open the file and assign it to a variable. At the end of working with a file the common operation is to make sure that file is closed.

The Python with statement makes using context managers easy. The general form of a with statement is:

```
with <create some object that understands context> as <some name>:
    do some stuff with the object
    ...
```

When the program exits the with block, the context manager handles the common stuff that normally happens. For example closing a file. A simple example will clear up all of this abstract discussion of contexts.

```
with open('mydata.txt') as md:
    print(md)
    for line in md:
        print(line)
print(md)
```

The first line of the with statement opens the file and assigns it to `md` then we can iterate over the file in any of the usual ways. and when we are done we simply stop indenting and let Python take care of closing the file and cleaning up.

## 11.8 Fetching Something From The Web

The Python libraries are pretty messy in places. But here is a very simple example that copies the contents at some web URL to a local file. We will need to get a few things right before this works:

- The resource we're trying to fetch must exist! Check this using a browser.
- We'll need permission to write to the destination filename, and the file will be created in the "current directory" - i.e. the same folder that the Python program is saved in.
- If we are behind a proxy server that requires authentication, (as some students are), this may require some more special handling to work around our proxy. Use a local resource for the purpose of this demonstration!

We will try to retrieve the content of the HTML of this page as in the following code.

```
import urllib.request

def retrieve_page(url):
    """_Retrieve the contents of a webpage.
    """
    my_socket = urllib.request.urlopen(url)
    dta = my_socket.read()
    return dta

the_text =
    retrieve_page("https://runestone.academy/runestone/books/published/thinkcspy/Files")
print(the_text)
```

## 11.9 Glossary

### Glossary

**open.** You must open a file before you can read its contents.

**close.** When you are done with a file, you should close it.

**read.** Will read the entire contents of a file as a string. This is often used in an assignment statement so that a variable can reference the contents of the file.

**readline.** Will read a single line from the file, up to and including the first instance of the newline character.

**readlines.** Will read the entire contents of a file into a list where each line of the file is a string and is an element in the list.

**write.** Will add characters to the end of a file that has been opened for writing.

**absolute file path.** The name of a file that includes a path to the file from the *root* directory of a file system. An *absolute file path* always starts with a /.

**relative file path.** The name of a file that includes a path to the file from the current working directory of a program. An *relative file path* never starts with a /.

## 11.10 Exercises

1. The following sample file called `studentdata.txt` contains one line for each student in an imaginary class. The student's name is the first thing on each line, followed by some exam scores. The number of scores might be different for each student. Using the text file `studentdata.txt` write a program that prints out the names of students that have more than six quiz scores.
2. Using the text file `studentdata.txt` (shown in exercise 1) write a program that calculates the average grade for each student, and print out the student's name along with their average grade.
3. Using the text file `studentdata.txt` (shown in exercise 1) write a program that calculates the minimum and maximum score for each student. Print out their name as well.
4. Here is a file called `labdata.txt` that contains some sample data from a lab experiment. Interpret the data file `labdata.txt` such that each line contains a an x,y coordinate pair. Write a function called `plotRegression` that reads the data from this file and uses a turtle to plot those points and a best fit line according to the following formulas:

$$y = \bar{y} + m(x - \bar{x})$$
$$m = \frac{\sum x_i y_i - n\bar{x}\bar{y}}{\sum x_i^2 - n\bar{x}^2}$$

where  $\bar{x}$  is the mean of the x-values,  $\bar{y}$  is the mean of the y-values and n is the number of points. If you are not familiar with the mathematical  $\sum$  it is the sum operation. For example  $\sum x_i$  means to add up all the x values.

Your program should analyze the points and correctly scale the window using `setworldcoordinates` so that that each point can be plotted. Then you should draw the best fit line, in a different color, through the points.

5. At the bottom of this page is a very long file called `mystery.txt`. The lines of this file contain either the word UP or DOWN or a pair of numbers. UP and DOWN are instructions for a turtle to lift up or put down its tail. The pairs of numbers are some x,y coordinates. Write a program that reads the file `mystery.txt` and uses the turtle to draw the picture described by the commands and the set of points.

Here is the `mystery.txt` file:

# Chapter 12

## Dictionaries

### 12.1 Dictionaries

All of the compound data types we have studied in detail so far — strings, lists, and tuples — are sequential collections. This means that the items in the collection are ordered from left to right and they use integers as indices to access the values they contain.

**Dictionaries** are a different kind of collection. They are Python’s built-in **mapping type**. A map is an unordered, associative collection. The association, or mapping, is from a **key**, which can be any immutable type, to a **value**, which can be any Python data object.

As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the keys are strings and the values will also be strings.

One way to create a dictionary is to start with the empty dictionary and add **key-value pairs**. The empty dictionary is denoted {}

```
eng2sp = {}
eng2sp['one'] = 'uno'
eng2sp['two'] = 'dos'
eng2sp['three'] = 'tres'
```

The first assignment creates an empty dictionary named eng2sp. The other assignments add new key-value pairs to the dictionary. The left hand side gives the dictionary and the key being associated. The right hand side gives the value being associated with that key. We can print the current value of the dictionary in the usual way. The key-value pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.

The order of the pairs may not be what you expected. Python uses complex algorithms, designed for very fast access, to determine where the key-value pairs are stored in a dictionary. For our purposes we can think of this ordering as unpredictable.

Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as the previous output.

```
eng2sp = {'three': 'tres', 'one': 'uno', 'two': 'dos'}
print(eng2sp)
```

It doesn’t matter what order we write the pairs. The values in a dictionary are accessed with keys, not with indices, so there is no need to care about ordering.

Here is how we use a key to look up the corresponding value.

```

eng2sp = {'three': 'tres', 'one': 'uno', 'two': 'dos'}

value = eng2sp['two']
print(value)

```

The key 'two' yields the value 'dos'.

**Note 12.1.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

#### Check your understanding

**Checkpoint 12.1.2** A dictionary is an unordered collection of key-value pairs.

A. False

B. True

**Checkpoint 12.1.3** What is printed by the following statements?

```

mydict = {"cat":12, "dog":6, "elephant":23}
print(mydict["dog"])

```

A. 12

B. 6

C. 23

D. Error, you cannot use the index operator with a dictionary.

## 12.2 Dictionary Operations

The `del` statement removes a key-value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock. If someone buys all of the pears, we can remove the entry from the dictionary.

```

inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
             'pears': 217}

del inventory['pears']

```

Dictionaries are also mutable. As we've seen before with lists, this means that the dictionary can be modified by referencing an association on the left hand side of the assignment statement. In the previous example, instead of deleting the entry for `pears`, we could have set the inventory to `0`.

```

inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
             'pears': 217}

inventory['pears'] = 0

```

Similarly, a new shipment of 200 bananas arriving could be handled like this.

```

inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
             'pears': 217}
inventory['bananas'] = inventory['bananas'] + 200

numItems = len(inventory)

```

Notice that there are now 512 bananas—the dictionary has been modified. Note also that the `len` function also works on dictionaries. It returns the number of key-value pairs:

### Check your understanding

**Checkpoint 12.2.1** What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23}
mydict["mouse"] = mydict["cat"] + mydict["dog"]
print(mydict["mouse"])
```

- A. 12
- B. 0
- C. 18
- D. Error, there is no entry with mouse as the key.

## 12.3 Dictionary Methods

Dictionaries have a number of useful built-in methods. The following table provides a summary and more details can be found in the [Python Documentation](#)<sup>1</sup>.

**Table 12.3.1**

Method	Parameters	Description
keys	none	Returns a view of the keys in the dictionary
values	none	Returns a view of the values in the dictionary
items	none	Returns a view of the key-value pairs in the dictionary
get	key	Returns the value associated with key; None otherwise
get	key,alt	Returns the value associated with key; alt otherwise

The `keys` method returns what Python 3 calls a `view` of its underlying keys. We can iterate over the view or turn the view into a list by using the `list` conversion function.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
             'pears': 217}

for akey in inventory.keys():      # the order in which we
    get the keys is not defined
    print("Got key", akey, "which maps to value",
          inventory[akey])

ks = list(inventory.keys())
print(ks)
```

It is so common to iterate over the keys in a dictionary that you can omit the `keys` method call in the `for` loop — iterating over a dictionary implicitly iterates over its keys.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
             'pears': 217}

for k in inventory:
    print("Got key", k)
```

---

<sup>1</sup><http://docs.python.org/py3k/library/stdtypes.html#mapping-types-dict>

As we saw earlier with strings and lists, dictionary methods use dot notation, which specifies the name of the method to the right of the dot and the name of the object on which to apply the method immediately to the left of the dot. The empty parentheses in the case of `keys` indicate that this method takes no parameters.

The `values` and `items` methods are similar to `keys`. They return view objects which can be turned into lists or iterated over directly. Note that the `items` are shown as tuples containing the key and the associated value.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
             'pears': 217}

print(list(inventory.values()))
print(list(inventory.items()))

for (k,v) in inventory.items():
    print("Got", k, "that maps to", v)

for k in inventory:
    print("Got", k, "that maps to", inventory[k])
```

Note that tuples are often useful for getting both the key and the value at the same time while you are looping. The two loops do the same thing.

The `in` and `not in` operators can test if a key is in the dictionary:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
             'pears': 217}
print('apples' in inventory)
print('cherries' in inventory)

if 'bananas' in inventory:
    print(inventory['bananas'])
else:
    print("We have no bananas")
```

This operator can be very useful since looking up a non-existent key in a dictionary causes a runtime error.

The `get` method allows us to access the value associated with a key, similar to the `[ ]` operator. The important difference is that `get` will not cause a runtime error if the key is not present. It will instead return `None`. There exists a variation of `get` that allows a second parameter that serves as an alternative return value in the case where the key is not present. This can be seen in the final example below. In this case, since “cherries” is not a key, return 0 (instead of `None`).

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
             'pears': 217}

print(inventory.get("apples"))
print(inventory.get("cherries"))

print(inventory.get("cherries", 0))
```

**Note 12.3.2** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

### Check your understanding

**Checkpoint 12.3.3** What is printed by the following statements?

```

mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
keylist = list(mydict.keys())
keylist.sort()
print(keylist[3])

```

- A. cat
- B. dog
- C. elephant
- D. bear

**Checkpoint 12.3.4** What is printed by the following statements?

```

mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
answer = mydict.get("cat") // mydict.get("dog")
print(answer)

```

- A. 2
- B. 0.5
- C. bear
- D. Error, divide is not a valid operation on dictionaries.

**Checkpoint 12.3.5** What is printed by the following statements?

```

mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
print("dog" in mydict)

```

- A. True
- B. False

**Checkpoint 12.3.6** What is printed by the following statements?

```

mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
print(23 in mydict)

```

- A. True
- B. False

**Checkpoint 12.3.7** What is printed by the following statements?

```

total = 0
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
for akey in mydict:
    if len(akey) > 3:
        total = total + mydict[akey]
print(total)

```

- A. 18
- B. 43
- C. 0
- D. 61

## 12.4 Aliasing and Copying

Because dictionaries are mutable, you need to be aware of aliasing (as we saw with lists). Whenever two variables refer to the same dictionary object, changes to one affect the other. For example, `opposites` is a dictionary that contains pairs of opposites.

```
opposites = {'up': 'down', 'right': 'wrong', 'true':
             'false'}
alias = opposites

print(alias is opposites)

alias['right'] = 'left'
print(opposites['right'])
```

As you can see from the `is` operator, `alias` and `opposites` refer to the same object.

If you want to modify a dictionary and keep a copy of the original, use the dictionary `copy` method. Since `acopy` is a copy of the dictionary, changes to it will not effect the original.

```
acopy = opposites.copy()
acopy['right'] = 'left'      # does not change opposites
```

### Check your understanding

**Checkpoint 12.4.1** What is printed by the following statements?

```
mydict = {"cat":12, "dog":6, "elephant":23, "bear":20}
yourdict = mydict
yourdict["elephant"] = 999
print(mydict["elephant"])
```

- A. 23
- B. None
- C. 999
- D. Error, there are two different keys named elephant.

## 12.5 Sparse Matrices

A matrix is a two dimensional collection, typically thought of as having rows and columns of data. One of the easiest ways to create a matrix is to use a list of lists. For example, consider the matrix shown below.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

We can represent this collection as five rows, each row having five columns. Using a list of lists representation, we will have a list of five items, each of which is a list of five items. The outer items represent the rows and the items in the nested lists represent the data in each column.

```
matrix = [[0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 3, 0]]
```

One thing that you might note about this example matrix is that there are many items that are zero. In fact, only three of the data values are nonzero. This type of matrix has a special name. It is called a [sparse matrix](#)<sup>1</sup>.

Since there is really no need to store all of the zeros, the list of lists representation is considered to be inefficient. An alternative representation is to use a dictionary. For the keys, we can use tuples that contain the row and column numbers. Here is the dictionary representation of the same matrix.

```
| matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

We only need three key-value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer.

To access an element of the matrix, we could use the [] operator:

```
matrix[(0, 3)]
```

Notice that the syntax for the dictionary representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers.

There is one problem. If we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key. The alternative version of the `get` method solves this problem. The first argument will be the key. The second argument is the value `get` should return if the key is not in the dictionary (which would be 0 since it is sparse).

```
matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
print(matrix.get((0,3)))

print(matrix.get((1, 3), 0))
```

#### Note 12.5.1 Lab.

- Counting Letters [Section 21.5](#) In this guided lab exercise we will work through a problem solving exercise that will use dictionaries to generalize the solution to counting the occurrences of all letters in a string.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Sparse\\_matrix](http://en.wikipedia.org/wiki/Sparse_matrix)

**Note 12.5.2 Lab.**

- Letter Count Histogram [Section 21.6](#) Combine the previous lab with the histogram example.

## 12.6 Glossary

### Glossary

**dictionary.** A collection of key-value pairs that maps from keys to values. The keys can be any immutable type, and the values can be any type.

**key.** A data item that is *mapped to* a value in a dictionary. Keys are used to look up values in a dictionary.

**key-value pair.** One of the pairs of items in a dictionary. Values are looked up in a dictionary by key.

**mapping type.** A mapping type is a data type comprised of a collection of keys and associated values. Python's only built-in mapping type is the dictionary. Dictionaries implement the [associative array](#)<sup>1</sup> abstract data type.

## 12.7 Exercises

1. Write a program that allows the user to enter a string. It then prints a table of the letters of the alphabet in alphabetical order which occur in the string together with the number of times each letter occurs. Case should be ignored. A sample run of the program might look this this:

```
Please enter a sentence: This is String with Upper and lower case Letters.
a 2
c 1
d 1
e 5
g 1
h 2
i 4
l 2
n 2
o 1
p 2
r 4
s 5
t 5
u 1
w 2
$
```

Give the Python interpreter's response to each of the following from a continuous interpreter session:

a	>>> d = {'apples': 15, 'bananas': 35, 'grapes': 12} >>> d['banana']
b	

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Associative\\_array](http://en.wikipedia.org/wiki/Associative_array)

```

    |     >>> d['oranges'] = 20
    |     >>> len(d)
c    |
    |     >>> 'grapes' in d
d    |
    |     >>> d['pears']
e    |
    |     >>> d.get('pears', 0)
f    |
    |     >>> fruits = d.keys()
    |     >>> fruits.sort()
    |     >>> print(fruits)
g    |
    |     >>> del d['apples']
    |     >>> 'apples' in d

```

Be sure you understand why you get each result. Then apply what you have learned to fill in the body of the function below, and add code for the tests indicated:

```

def add_fruit(inventory, fruit, quantity=0):
    pass

# make these tests work...
new_inventory = {}
add_fruit(new_inventory, 'strawberries', 10)
# test that 'strawberries' in new_inventory
# test that new_inventory['strawberries'] is 10
add_fruit(new_inventory, 'strawberries', 25)
# test that new_inventory['strawberries'] is now 35)

```

2. Write a program called `alice_words.py` that creates a text file named `alice_words.txt` containing an alphabetical listing of all the words, and the number of times each occurs, in the text version of Alice's Adventures in Wonderland. (You can obtain a free plain text version of the book, along with many others, from <http://www.gutenberg.org><sup>1</sup>.) The first 10 lines of your output file should look something like this

Table 12.7.1

Word	Count
a	631
a-piece	1
abide	1
able	1
about	94
above	3
absence	1
absurd	2

How many times does the word, `alice`, occur in the book? If you are writing this in the activecode window simply print out the results rather than write them to a file.

3. What is the longest word in Alice in Wonderland? How many characters does it have?

---

<sup>1</sup><http://www.gutenberg.org>

4. Here's a table of English to Pirate translations

**Table 12.7.2**

English	Pirate
sir	matey
hotel	fleabag inn
student	swabbie
boy	matey
madam	proud beauty
professor	foul blaggart
restaurant	galley
your	yer
excuse	arr
students	swabbies
are	be
lawyer	foul blaggart
the	th'
restroom	head
my	me
hello	avast
is	be
man	matey

Write a function named `translator` that takes a parameter containing a sentence in English (no punctuation and all words in lowercase) and returns that sentence translated to Pirate.

For example, the sentence “hello there students” should be translated to “avast there swabbies”.

```
def translator(english):

    pirate = {}
    pirate['sir'] = 'matey'
    pirate['hotel'] = 'fleabag\u00e9inn'
    pirate['student'] = 'swabbie'
    pirate['boy'] = 'matey'
    pirate['restaurant'] = 'galley'
    pirate['hello'] = 'avast'
    pirate['students'] = 'swabbies'

    # Complete the function

=====
from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(translator("hello\u00e9there\u00e9
            students"),'avast\u00e9there\u00e9
            swabbies','translator("hello\u00e9there\u00e9
            students")\u00d7yields\u00d7"avast\u00e9there\u00e9swabbies"')
        self.assertEqual(translator("the\u00e9boy\u00e9stayed\u00e9in\u00e9
            the\u00e9hotel"),'the\u00e9matey\u00e9stayed\u00e9in\u00e9the\u00e9
            fleabag\u00e9inn','translator("the\u00e9boy\u00e9stayed\u00e9in\u00e9
            the\u00e9hotel")\u00d7yields\u00d7"the\u00e9matey\u00e9stayed\u00e9in\u00e9the\u00e9
            fleabag\u00e9inn"')

myTests().main()
```

# Chapter 13

## Exceptions

### 13.1 What is an exception?

An *exception* is a signal that a condition has occurred that can't be easily handled using the normal flow-of-control of a Python program. *Exceptions* are often defined as being “errors” but this is not always the case. All errors in Python are dealt with using *exceptions*, but not all *exceptions* are errors.

### 13.2 Runetime Stack and `raise` command

There are cases where the sequential flow-of-control does not work well. An example will best explain this.

Let's suppose that a program contains complex logic that is appropriately subdivided into functions. The program is running and it currently is executing function D, which was called by function C, which was called by function B, which was called by function A, which was called from the main function. This is illustrated by the following simplistic code example:

```
def main():
    A()

def A():
    B()

def B():
    C()

def C():
    D()

def D()
    # processing
```

Function D determines that the current processing won't work for some reason and needs to send a message to the main function to try something different. However, all that function D can do using normal flow-of-control is to return a value to function C. So function D returns a special value to function C that means “try something else”. Function C has to recognize this value, quit its processing, and return the special value to function B. And so forth and so on. It would be very helpful if function D could communicate directly with the main function (or functions A and B) without sending a

special value through the intermediate calling functions. Well, that is exactly what an *exception* does. An *exception* is a message to any function currently on the executing program’s “run-time-stack”. (The “run-time-stack” is what keeps track of the active function calls while a program is executing.)

In Python, you create an *exception* message using the `raise` command. The simplest format for a `raise` command is the keyword `raise` followed by the name of an exception. For example:

```
| raise ExceptionName
```

So what happens to an *exception* message after it is created? The normal flow-of-control of a Python program is interrupted and Python starts looking for any code in its run-time-stack that is interested in dealing with the message. It always searches from its current location at the bottom of the run-time-stack, up the stack, in the order the functions were originally called. A `try: except:` block is used to say “hey, I can deal with that message.” The first `try: except:` block that Python finds on its search back up the run-time-stack will be executed. If there is no `try: except:` block found, the program “crashes” and prints its run-time-stack to the console.

Let’s take a look at several code examples to illustrate this process. If function D had a `try: except:` block around the code that `raised` a `MyException` message, then the flow-of-control would be passed to the local `except` block. That is, function D would handle it’s own issues.

```
def main():
    A()

def A():
    B()

def B():
    C()

def C():
    D()

def D():
    try:
        # processing code
        if something_special_happened:
            raise MyException
    except MyException:
        # execute if the MyException message happened
```

But perhaps function C is better able to handle the issue, so you could put the `try: except:` block in function C:

```
def main():
    A()

def A():
    B()

def B():
    C()

def C():
    try:
        D()
```

```

except MyException:
    # execute if the MyException message happened

def D():
    # processing code
    if something_special_happened:
        raise MyException

```

But perhaps the main function is better able to handle the issue, so you could put the `try: except:` block in the main function:

```

def main():
    try:
        A()
    except MyException:
        # execute if the MyException message happened

    def A():
        B()

    def B():
        C()

    def C():
        D()

    def D():
        # processing code
        if something_special_happened:
            raise MyException

```

### 13.3 Standard Exceptions

Most of the standard *exceptions* built into Python are listed below. They are organized into related groups based on the types of issues they deal with.

**Table 13.3.1**

Language Exceptions	Description
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ImportError	Raised when an import statement fails.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
TypeError	Raised when an operation or function is attempted that is invalid for the specified type.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
ValueError	Raised when the built-in function for a data type has the valid type of argument.
RuntimeError	Raised when a generated error does not fall into any category.
MemoryError	Raised when a operation runs out of memory.
RecursionError	Raised when the maximum recursion depth has been exceeded.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered, the program will terminate.

**Table 13.3.2**

<b>Math Exceptions</b>	<b>Description</b>
ArithmetError	Base class for all errors that occur for numeric calculation. You know a math error occurs.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.

**Table 13.3.3**

<b>I/O Exceptions</b>	<b>Description</b>
FileNotFoundException	Raised when a file or directory is requested but doesn't exist.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function.
PermissionError	Raised when trying to run an operation without the adequate access rights.
EOFError	Raised when there is no input from either the raw_input() or input() function and the user reaches the end of the file.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.

**Table 13.3.4**

<b>Other Exceptions</b>	<b>Description</b>
Exception	Base class for all exceptions. This catches most exception messages.
StopIteration	Raised when the next() method of an iterator does not point to any object.
AssertionError	Raised in case of failure of the Assert statement.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled, it exits the program.
OSErrror	Raises for operating system related errors.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
AttributeError	Raised in case of failure of an attribute reference or assignment.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is called.

All exceptions are objects. The classes that define the objects are organized in a hierarchy, which is shown below. This is important because the parent class of a set of related exceptions will catch all exception messages for itself and its child exceptions. For example, an `ArithmetError` exception will catch itself and all `FloatingPointError`, `OverflowError`, and `ZeroDivisionError` exceptions.

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmetError
        |    +-- FloatingPointError
        |    +-- OverflowError
        |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    +-- LookupError
        |    +-- IndexError
        |    +-- KeyError
    +-- MemoryError
    +-- NameError
        |    +-- UnboundLocalError

```

```

+-- OSError
|   +-- BlockingIOError
|   +-- ChildProcessError
|   +-- ConnectionError
|       |   +-- BrokenPipeError
|       |   +-- ConnectionAbortedError
|       |   +-- ConnectionRefusedError
|       |   +-- ConnectionResetError
|   +-- FileExistsError
|   +-- FileNotFoundError
|   +-- InterruptedError
|   +-- IsADirectoryError
|   +-- NotADirectoryError
|   +-- PermissionError
|   +-- ProcessLookupError
|   +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|   +-- NotImplementedError
|   +-- RecursionError
+-- SyntaxError
|   +-- IndentationError
|       +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning

```

## 13.4 Principles for using Exceptions

There are many bad examples of *exception* use on the Internet. The purpose of an *exception* is to modify the flow-of-control, not to catch simple errors. If your `try: except:` block is in the same function that `raises` the exception, you are probably mis-using exceptions. If a condition can be handled using the normal flow-of-control, don't use an exception! Example 1:

**Table 13.4.1**

**DON'T DO THIS:**

```

try:
    average = sum(a_list) / len(a_list)
except ZeroDivisionError:
    average = 0

```

Example 2:

When you can just as easily

```

if len(a_list) > 0:
    average = sum(a_list) / len(a_list)
else:
    average = 0

```

**Table 13.4.2****DON'T DO THIS:**

```
try:
    value = my_list[index]
except IndexError:
    value = -1
```

Example 3:

**Table 13.4.3****DON'T DO THIS:**

```
try:
    value = my_dictionary[key]
except KeyError:
    value = -1
```

If you call a function that potentially raises exceptions, and you can do something appropriate to deal with the exception, then surround the code that contains the function call with a `try: except:` block. Example: Suppose you have a function that reads a file to set the state of an application when it starts up. You should catch any errors related to reading the file and set the state of the application to default values if they can't be set from the file.

```
try:
    load_state('previous_state.txt')
except OSError:
    set_state_to_defaults()
```

If you call a function that potentially raises exceptions, and you can't do anything meaningful about the conditions that are raised, then don't catch the exception message(s).

## 13.5 Exceptions Syntax

### 13.5.1 Introduction

There are many variations on the code that catches exceptions. Here is a brief summary, but other code variations are possible.

### 13.5.2 Catch All Exceptions

Catch all exceptions, regardless of their type. This will prevent your program from crashing, but this type of exception handling is rarely useful because you can't do anything meaningful to recover from the abnormal condition.

```
try:
    # Your normal code goes here.
    # Your code should include function calls which might
        raise exceptions.
except:
    # If any exception was raised, then execute this code
        block.
```

### 13.5.3 Catch A Specific Exception

This is perhaps the most often used syntax. It catches one specific condition and tries to re-cover from the condition.

When you can just as easily

```
if 0 <= index < len(my_list):
    value = my_list[index]
else:
    value = -1
```

When you can just as easily

```
if key in my_dictionary:
    value = my_dictionary[key]
else:
    value = -1
```

```

try:
    # Your normal code goes here.
    # Your code should include function calls which might
        raise exceptions.
except ExceptionName:
    # If ExceptionName was raised, then execute this block.

```

### 13.5.4 Catch Multiple Specific Exceptions

```

try:
    # Your normal code goes here.
    # Your code should include function calls which might
        raise exceptions.
except Exception_one:
    # If Exception_one was raised, then execute this block.
except Exception_two:
    # If Exception_two was raised, then execute this block.
else:
    # If there was no exception then execute this block.

```

### 13.5.5 Clean-up After Exceptions

If you have code that you want to be executed even if exceptions occur, you can include a `finally` code block:

```

try:
    # Your normal code goes here.
    # Your code might include function calls which might
        raise exceptions.
    # If an exception is raised, some of these statements
        might not be executed.
finally:
    # This block of code will always execute, even if there
        are exceptions raised

```

### 13.5.6 An Example of File I/O

One place where you will always want to include exception handling is when you read or write to a file. Here is a typical example of file processing. Note that the outer `try: except:` block takes care of a missing file or the fact that the existing file can't be opened for writing. The inner `try: except:` block protects against output errors, such as trying to write to a device that is full. The `finally` code guarantees that the file is closed properly, even if there are errors during writing.

```

try:
    f = open("my_file.txt", "w")
    try:
        f.write("Writing some data to the file")
    finally:
        f.close()
except IOError:
    print "Error: my_file.txt does not exist or it can't be
        opened for output."

```

## 13.6 The `finally` clause of the `try` statement

A common programming pattern is to grab a resource of some kind — e.g. we create a window for turtles to draw on, or we dial up a connection to our internet service provider, or we may open a file for writing. Then we perform some computation which may raise an exception, or may work without any problems.

Whatever happens, we want to “clean up” the resources we grabbed — e.g. close the window, disconnect our dial-up connection, or close the file. The `finally` clause of the `try` statement is the way to do just this. Consider this (somewhat contrived) example:

```
import turtle
import time

def show_poly():
    try:
        win = turtle.Screen()      # Grab/create a resource, e.g.
                                    # a window
        tess = turtle.Turtle()

        # This dialog could be cancelled,
        # or the conversion to int might fail, or n might be
        # zero.
        n = int(input("How many sides do you want in your
                      polygon?"))
        angle = 360 / n
        for i in range(n):          # Draw the polygon
            tess.forward(35)
            tess.left(angle)
        time.sleep(3)                # Make program wait a few
                                    # seconds
    except Exception as e:
        print("insufficient number of sides")
        print(e)
    finally:
        win.bye()                  # Close the turtle's window

show_poly()
```

In lines 20–22, `show_poly` is called three times. Each one creates a new window for its turtle, and draws a polygon with the number of sides input by the user. But what if the user enters a string that cannot be converted to an `int`? What if they close the dialog? We’ll get an exception, *but even though we’ve had an exception, we still want to close the turtle’s window*. Lines 17–18 does this for us. Whether we complete the statements in the `try` clause successfully or not, the `finally` block will always be executed.

Notice that the exception is still unhandled — only an `except` clause can handle an exception, so our program will still crash. But at least its turtle window will be closed before it crashes!

## 13.7 Glossary

### Glossary

**exception.** An error that occurs at runtime.

**handle an exception.** To prevent an exception from terminating a program by wrapping the block of code in a `try / except` construct.

**raise.** To cause an exception by using the `raise` statement.

## 13.8 Exercises

Write a function named `readposint` that uses the `input` dialog to prompt the user for a positive integer and then checks the input to confirm that it meets the requirements. It should be able to handle inputs that cannot be converted to `int`, as well as negative `int`, and edge cases (e.g. when the user closes the dialog, or does not enter anything at all.)

# Chapter 14

# Web Applications

## 14.1 Web Applications

In this chapter, you will learn how to create web applications in Python. There are two kinds of web applications – “client side” web applications and “server side” web applications. You’ve probably used both kinds in your travels on the World-Wide Web.

- **Client-side** web applications are programs that are downloaded to a web browser and executed on the user’s local machine. Client-side applications are typically written in JavaScript and embedded in web pages.
- **Server-side** web applications are programs that run on web server computers, rather than on the user’s local machine. Typically, a server-side web application displays a form with text boxes and other data collection mechanisms. The user fills out the form, clicks a submit button, and the browser sends the form to the web application on the server, which processes the request, and responds with another web page. If you’ve ever shopped online, you’ve used a server-side web application.

In this chapter, you will learn to build server-side web applications, which I will refer to simply as “web applications.”

## 14.2 How the Web Works

Before learning how to write a web application, you need to understand a bit about how web browsers work, and how web applications interact with users.

A **web browser**, at its core, is a fairly simple application. Basically, web browsers

**Request files  
from web  
servers.**

1

The World-Wide Web is composed of thousands of web servers connected to the Internet. Each web server contains lots of different kinds of files that web browsers can request: HTML pages, image files, audio files, and other resources. When you click a link on a web page, the web browser sends a request to the web server, which transmits the requested file back to the browser.

**Process the downloaded files appropriately.**

2

Once the web browser has downloaded the requested file, it needs to do something with it. Web browsers know how to render an HTML document, show images, play audio files, and so on. If the web browser doesn't know what to do with a file, it usually prompts the user to save the file, so the user can do something with it.

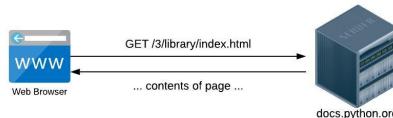
Let's take a specific example. Use your browser to access the following URL:

[`https://docs.python.org/3/library/index.html`](https://docs.python.org/3/library/index.html)<sup>1</sup>

**Note 14.2.1** A **URL** (“Uniform Resource Locator”) is the address of a resource on the Web. It has three sections: the **protocol** (ex. `https:`) the browser uses to request the resource, the **server** where the document is located (ex. `docs.python.org`), and the **path** to the requested resource on the server (ex. `/3/library/index.html`).

When you click on this link, here's what happens:

- 1 The browser opens a network connection to the web server named `docs.python.org`
- 2 The browser requests a file located on the server at `/3/library/index.html`
- 3 The web server transmits the HTML file back to the browser
- 4 The browser renders the HTML document



**Figure 14.2.2**

If you want to see the file transmitted by the web server to the browser, right-click in the browser window and choose **View Page Source** (your browser's option to view the source may be slightly different). The browser shows you the file it downloaded from the web server.

### 14.3 How Web Applications Work

Most of the time, when your browser requests a file from a web server, the server simply transmits the contents of the file back to the browser. But sometimes, the “file” your browser requests isn't really a file at all.

Try typing the following URL into your browser:

[`https://google.com/search?q=Microsoft`](https://google.com/search?q=Microsoft)<sup>1</sup>

You'll get back a page of search results about Microsoft from the Google search engine (at least, you will unless Google has changed how it performs searches since this chapter was written). How did this happen?

Well, your browser did what it always does when you type in a URL:

---

<sup>1</sup>[`https://docs.python.org/3/library/index.html`](https://docs.python.org/3/library/index.html)

<sup>1</sup>[`https://google.com/search?q=Microsoft`](https://google.com/search?q=Microsoft)

- 1 The browser opened a network connection to the web server named google.com
- 2 The browser requested the “file” named /search?q=Microsoft from the web server
- 3 What the web server did at this point is different than the example above. There’s no “file” named “/search?q=Microsoft” on the Google web server. Instead, the web server ran a web application to search through Google’s massive database of websites for pages that mention “Microsoft”. The web application dynamically generated an HTML document containing the search results, and the web server transmitted that document back to the browser.
- 4 The browser rendered the HTML document

As far as your browser is concerned, there is no difference between requesting a “static” HTML file from a web server, and requesting a dynamically generated HTML file. It’s up to the web server to examine the request submitted by the web browser to determine whether it should serve up a regular document, or run a web application to generate a response.

Anytime you’re browsing the web, and you notice that the URL of the page you’re viewing has a question mark (?), you can be fairly certain that the page was generated “on the fly” by a web application on a web server. By the way, the portion of the URL that comes after the ? is called the “query string,” and contains input for the web application. Try changing the **query string** by substituting “Firefox” for “Microsoft” to see what I mean.

In summary, a (server-side) web application is a program that is run by a web server to produce output in response to an incoming request from a web browser.

## 14.4 Web Applications and HTML Forms

Perhaps you’re thinking, “I don’t usually perform searches by typing in URL’s — I fill out a search form.” True — if web applications forced users to interact with them by entering query strings, the World-Wide Web would be a much less popular place.

Let’s explore the relationship between forms and query strings a bit. Bring up the Google home page (I’ll wait):

<https://google.com><sup>1</sup>

Now, type in your query. When I type in “Microsoft” and click Search, here is what I see:

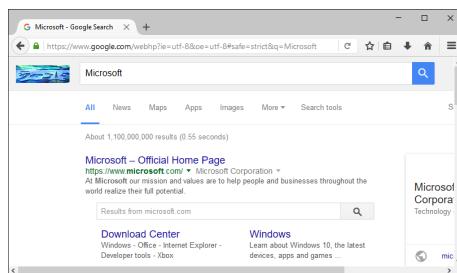


Figure 14.4.1

---

<sup>1</sup><https://google.com>

Now, take a good look at the URL in the title bar — notice the query string? It's a bit more complicated than the one I had you create by hand earlier. But you can probably pick out the “q=Microsoft” if you look closely. How did all of that get there? Well, when you clicked Search, the browser took the information you typed into the form, packaged it up into a query string, and transmitted it to the Google web server. You see, when you fill out a form on a web page and click Submit, the browser uses the form data to construct a URL, and then sends a normal request to the web server.

Even if you're a novice at writing **HTML pages**, it's not hard to learn to create HTML forms. Take a look at this simplified version of the Google home page:

```
<html>
<head>
    <title>Google</title>
</head>
<body>
    <div align="center">
        <br><br>
        <form action="https://google.com/search">
            Enter your search words: <input type="text"
                name="q"><br><br>
            <input type="submit" name="btnG" value="Google"
                Search">
        </form>
    </div>
</body>
</html>
```

Focus on the region of this example in between the `<form>` tags. Here's a quick overview of this part of the page:

- The form is the region of the page in between the `<form>` and `</form>` tags.
- The form can contain a mixture of text, regular HTML formatting tags, and form `<input>` tags
- Each `<input>` tag has a `type` and a `name` attribute. The `type` attribute specifies what kind of input area it is (“text” for a text box, “submit” for submit button, etc.). The `name` attribute specifies a name for the input area.
- When the user fills out the form and clicks the submit button, the browser constructs a URL by taking the form's `action` attribute (<https://google.com/search><sup>2</sup>), appending a ?, and constructing the query string using the names of the form input areas, together with the data entered by the user.

Try it out! Using Notepad, type in this example, and save it as `google-form.html`. Open it in your browser; you should see something like this:

---

<sup>2</sup><https://google.com/search>

**Figure 14.4.2**

Fill out the form, and, if Google still works as it did when this chapter was written, you should see search results appear in your browser.

For more information about creating HTML forms, you might take a look at the excellent [tutorial at w3schools.com](https://www.w3schools.com/html/html_forms.asp)<sup>3</sup>.

## 14.5 Writing Web Applications With Flask

In this section, you will learn how to write web applications using a Python framework called Flask.

Here is an example of a Flask web application:

```
from flask import Flask
from datetime import datetime

app = Flask(__name__)

@app.route('/')
def hello():
    return """<html><body>
Hello, world!
The time is: """ + str(datetime.now()) + """
</body></html>"""

if __name__ == "__main__":
    # Launch the Flask dev server
    app.run(host="localhost", debug=True)
```

The application begins by importing the Flask framework on line 1. Lines 6-11 define a function `hello()` that serves up a simple web page containing the date and time. The call to `app.run()` on Line 14 starts a small web server. The `run()` method does not return; it executes an infinite loop that waits for incoming requests from web browsers. When a web browser sends a request to the Flask web server, the server invokes the `hello()` function and returns the HTML code generated by the function to the web browser, which displays the result.

To see this in action, copy the code above into a text editor and save it as `flaskhello.py` (or whatever name you like). Then, download the Flask framework and install it on your computer. In many cases, you can accomplish this using the `pip` command included with your Python distribution:

```
pip install flask
```

Next, execute your `flaskhello.py` program from the command line:

```
python flaskhello.py
```

---

<sup>3</sup>[https://www.w3schools.com/html/html\\_forms.asp](https://www.w3schools.com/html/html_forms.asp)

**Note 14.5.1** If you are using a Mac or Linux computer, use the following command to install flask:

```
pip3 install flask
```

and execute your flaskhello.py program using the following command:

```
python3 flaskhello.py
```

When you launch the program, you should see a message similar to the following appear on the console:

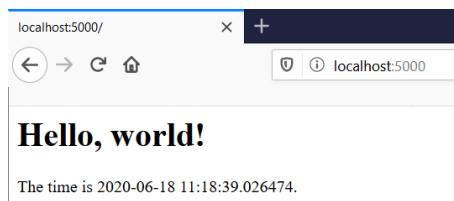
```
* Serving Flask app "sample" (lazy loading)
* Environment: production
WARNING: Do not use the development server in a production environment.
Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 244-727-575
* Running on http://localhost:5000/ (Press CTRL+C to quit)
```

**Note 14.5.2** If you get an error message of some sort, it is possible that your computer may be running a server application that is using the port number that Flask wants to use. See the next section, “More About Flask,” for a discussion of port numbers and how to address this issue.

Once the Flask server is running, use your browser to navigate to the following URL:

<http://localhost:5000/><sup>1</sup>

Your browser sends a request to the Flask server, and you should see a “Hello, world!” message appear:



**Figure 14.5.3**

To send the request again, press the Reload button in your browser. You should see the date and time change.

## 14.6 More About Flask

When you executed the flaskhello.py program in the previous section and used a web browser to access it with the url <http://localhost:5000/><sup>1</sup>, in addition to seeing a “Hello, world!” message in the browser, you should also have observed a log message like the following in the console:

```
127.0.0.1 - - [21/Apr/2016 08:02:28] "GET / HTTP/1.1" 200 -
```

Every time the Flask server receives a request from a browser, it writes a log message to the console. The message contains information such as the IP

---

<sup>1</sup><http://localhost:5000/>

<sup>1</sup><http://localhost:5000/>

**address** of the computer that sent the request (127.0.0.1 is a special address indicating the request came from the browser on the same computer that the Flask server is running on); the date and time of the request; the path of the incoming request (“/” in this case); and the status of the result (here, 200 indicates the request was successfully processed).

The Flask server continues running until you press Ctrl-C to stop it. At that point, if you try to send a request to the application from the browser, the browser will display an error message indicating that it cannot contact the server. Go ahead and try this, so you can recognize what the error message looks like in your particular browser.

Recall that every URL has at least three components: the **protocol**, **server**, and the **path**. In our case the URL <http://localhost:5000/><sup>2</sup> has the server name *localhost*, the path /, and an additional component: the port number, *5000*. Let’s discuss some details about each of these.

**Server name** When you use the name *localhost* in a URL, the browser attempts to connect to a server program running on your computer. This is the usual scenario when you are developing a web application: the browser and the server application are both running on the same computer. When you deploy the application to be hosted on an actual server, you will use the name of the server in the URL instead of the name *localhost*. If you want to experiment with deploying Flask applications to a public web server, check out [pythonanywhere.com](https://pythonanywhere.com)<sup>3</sup>, which (at the time of writing) provides free hosting for Flask web applications.

**Port number** Each server application running on a computer is assigned a distinct port number so that clients can connect to it. Port numbers range from 0 to 65,535. Web servers generally are assigned port number 80, and when the URL does not contain a port number, the web browser attempts to connect to a web server listening on port 80. But the default port number for Flask applications is 5000, so the URL must include that port number. You can specify a different port number for your Flask application in the line that launches the Flask server like this:

```
app.run(host="localhost", port=5001, debug=True)
```

Here, the Flask server binds to port 5001, and you would need to use that port number instead of 5000 in the URL in the browser.

---

<sup>2</sup><http://localhost:5000/>

<sup>3</sup><https://pythonanywhere.com>

**Path** When the Flask receives an incoming request, it examines the path and uses it to determine which function in your program should be executed to handle the request and generate a response. A Flask application can contain one or more of these request handler functions, which are decorated by a line immediately preceding the function that looks like this:

```
@app.route('/')
```

The path in the quotes is matched to the path of the incoming request from the browser. If the incoming path from the browser does not match any of the handler function paths defined by `@app.route()` decorators, an error occurs. For example, try entering the following URL into your browser when the `flaskhello.py` program in the last section is running:

`http://localhost:5000/blah`<sup>4</sup>

You will see an error message appear in the browser, and the log message that appears in the console will have the number 404 after the path, indicating that the path did not match, as shown below:

`127.0.0.1 - - [21/Apr/2016 08:02:51] "GET /blah HTTP/1.1" 404 -`

Here's another version of the `flaskhello.py` program that has two different pages. The first page displays a “Hello world” message and invites the user to click a link to view the time. When the user clicks the link, the time appears.

```
from flask import Flask
from datetime import datetime

app = Flask(__name__)

@app.route('/')
def hello():
    return HELLO_HTML

HELLO_HTML = """
<html><body>
<h1>Hello, world!</h1>
Click <a href="/time">here</a> for the time.
</body></html>
"""

@app.route('/time')
def time():
    return TIME_HTML.format(datetime.now())

TIME_HTML = """
<html><body>
The time is {0}.
</body></html>
"""
```

---

<sup>4</sup><http://localhost:5000/blah>

```

if __name__ == "__main__":
    # Launch the Flask dev server
    app.run(host="localhost", debug=True)

```

Here's how it works:

- 1 To begin, the user enters the URL <http://localhost:5000><sup>5</sup>, and the browser sends the request to the application. The Flask server matches that path “/” to the hello() function, invokes the function and returns the response to the browser.
- 2 The user clicks the link, which triggers the browser to send a request with the URL <http://localhost:5000/time><sup>6</sup> to the Flask server. The server matches the path “/time” to the time() function, invokes the function and returns a response containing the time to the browser.

Note that the user does not have to click the link in order to display the time. For example, the user could enter the URL <http://localhost:5000/time><sup>7</sup> directly into the browser to bypass the greeting page and get directly to the page showing the time.

The example above used the format() method to build an HTML string. For more information on format(), see [Section 9.5](#).

Also, notice how the example above defines separate HELLO\_HTML and TIME\_HTML variables to hold the HTML. This helps reduce cluttering the handler functions with HTML code, and separating the Python logic from the HTML also improves the overall readability and maintainability of the code.

## 14.7 Input For A Flask Web Application

In this section, we will design a web application that obtains input from the user. In the example in this section, the user must encode the input directly into the URL. In the next section, we'll provide a more user-friendly approach for obtaining input.

The URL used to interact with a web application can contain input data in addition to the path. This input data is typically encoded into the URL in the form of a query string. Here's an example of a URL containing a query string:

<http://www.bing.com/search?q=python+flask&go=Submit><sup>1</sup>

The query string is the portion that comes after the ? symbol:

q=python+flask&go=Submit

It contains a set of query variables and values, each query variable/value pair separated from the others by the & symbol. This example has a query variable named q whose value is *python+flask*, and a variable named go whose value is *Submit*.

Flask applications can access query variables using a dictionary named `request.args` (dictionaries are discussed in detail in [Section 12.1](#)). When a browser sends a request to a Flask application that contains a query string, the data in the query string is placed in the `request.args` dictionary, where it can be retrieved by the application. For example, in the Bing search URL

---

<sup>5</sup><http://localhost:5000>

<sup>6</sup><http://localhost:5000/time>

<sup>7</sup><http://localhost:5000/time>

<sup>1</sup><http://www.bing.com/search?q=python+flask&go=Submit>

above, if Bing were a Flask application, it could access the values in the query string like this:

```
q = request.args['q']
go = request.args['go']
```

This would retrieve the values ‘python flask’ and ‘Submit’ from the query string and store them, respectively, in `q` and `go`.

Here is an enhanced version of the original `flaskhello.py` program that gets the user’s name from the query string and uses it to greet the user:

```
from flask import Flask, request
from datetime import datetime

app = Flask(__name__)

@app.route('/')
def hello():
    name = request.args['name']
    return HELLO_HTML.format(
        name, str(datetime.now()))

HELLO_HTML = """
<html><body>
    Hello, {0}!
    The time is {1}.
</body></html>"""

if __name__ == "__main__":
    # Launch the Flask dev server
    app.run(host="localhost", debug=True)
```

To test this example, you would need to enter the following URL into the browser:

<http://localhost:5000/?name=Frank><sup>2</sup>

If the name parameter is omitted, the application will crash when it attempts to retrieve the query parameter from the dictionary, because indexing a dictionary with a key that is not present in the dictionary is illegal. To make the application more robust, we could change line 8 to check to see if the name parameter was submitted:

```
if 'name' in request.args:
    name = request.args['name']
else:
    name = 'World'
```

The test `'name' in request.args` is True if ‘name’ was present in the query parameters, and False if not.

A shorter way to handle a missing query parameter is to use the dictionary `get()` method, which allows us to supply a default value to use in case the user omits the query parameter. The if statement above could be rewritten with a single line of code:

```
name = request.args.get('name', 'World')
```

This line does the same check as the if statement. If ‘name’ is present in the query parameters, its value is stored in `name`. Otherwise, the value ‘World’ is stored in `name` if no name parameter was supplied.

---

<sup>2</sup><http://localhost:5000/?name=Frank>

## 14.8 Web Applications With a User Interface

This section builds on the material in the preceding sections to present a web application that prompts the user to provide input, performs some processing, and displays results.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def home():
    return HOME_HTML

HOME_HTML = """
<html><body>
<h2>Welcome to the Greeter</h2>
<form action="/greet">
    What's your name? <input type='text' name='username'><br>
    What's your favorite food? <input type='text' name='favfood'><br>
    <input type='submit' value='Continue'>
</form>
</body></html>"""

@app.route('/greet')
def greet():
    username = request.args.get('username', '')
    favfood = request.args.get('favfood', '')
    if username == '':
        username = 'World'
    if favfood == '':
        msg = 'You did not tell me your favorite food.'
    else:
        msg = 'I like ' + favfood + ', too.'

    return GREET_HTML.format(username, msg)

GREET_HTML = """
<html><body>
<h2>Hello, {0}!</h2>
{1}
</body></html>
"""

if __name__ == "__main__":
    # Launch the Flask dev server
    app.run(host="localhost", debug=True)
```

The program is organized as follows:

- Lines 6-8 define the `home()` function, which defines the starting page for the application. It displays a form that prompts for the user's name and favorite food.
- The form's `action` attribute on Line 13 specifies that the form submission will be directed to the path `/greet`. Processing for this path is defined by the `greet()` function on lines 20-31.

- Lines 22-29 extract the information submitted on the form and compute a response message.

## 14.9 Glossary

### Glossary

**Client-side web applications.** programs that are downloaded to a web browser and executed on the user's local machine. Client-side applications are typically written in JavaScript and embedded in web pages.

**Server-side web applications.** programs that run on web server computers, rather than on the user's local machine. Typically, a server-side web application displays a form with text boxes and other data collection mechanisms. The user fills out the form, clicks a submit button, and the browser sends the form to the web application on the server, which processes the request, and responds with another web page.

**web browser.** an application software for accessing the World Wide Web. When a user requests a web page from a particular website, the web browser retrieves the necessary content from a web server and then displays the page on the user's device.

**URL (Uniform Resource Locator).** the address of a web page.

**protocol.** a system of rules that allows two or more entities of a communications system to transmit information. The protocol defines the rules, syntax, semantics and synchronization of communication and possible error recovery methods.

**server.** a computer or computer program which manages access to a centralized resource or service in a network.

**query string.** the part of a URL that assigns values to specified parameters, such as a search query in Google

**HTML(Hypertext Markup Language) pages::** the standard markup language for documents designed to be displayed in a web browser. It tells the web browser that the user sees how it should look and what information should be on the page.

**IP address.** a special address indicating the request came from the browser on the same computer that the Flask server is running on.

# Chapter 15

# GUI and Event Driven Programming

## 15.1 Graphical User Interfaces

A **graphical user interface** (GUI) allows a user to interact with a computer program using a pointing device that manipulates small pictures on a computer screen. The small pictures are called **icons** or **widgets**. Various types of pointing devices can be used, such as a mouse, a stylus pen, or a human finger on a touch screen.

We refer to programs that use a **graphical user interface** as “GUI programs.” A GUI program is very different from a program that uses a **command line interface** which receives user input from typed characters on a keyboard. Typically programs that use a **command line interface** perform a series of tasks in a predetermined order and then terminate. However, a GUI program creates the **icons** and **widgets** that are displayed to a user and then it simply waits for the user to interact with them. The order that tasks are performed by the program is under the user’s control – not the program’s control! This means a GUI program must keep track of the “state” of its processing and respond correctly to user commands that are given in any order the user chooses. This style of programming is called “event driven programming.” In fact, by definition, all *GUI programs* are *event-driven programs*.

## 15.2 Tkinter Standard Dialog Boxes

### 15.2.1 Introduction

There are many common programming tasks that can be performed using pre-defined GUI dialog boxes. The following discussion describes these dialog boxes and provides some simple examples. You can refer to the Python documentation for additional optional parameters.

### 15.2.2 Messages

A **messagebox** can display information to a user. There are three variations on these dialog boxes based on the type of message you want to display. The functions’ first parameter gives a name for the dialog box which is displayed in the window’s header. The second parameter is the text of the message. The functions return a string which is typically ignored.

```

from tkinter import messagebox

messagebox.showinfo("Information", "Informativemessage")
messagebox.showerror("Error", "Errormessage")
messagebox.showwarning("Warning", "Warningmessage")

```

### 15.2.3 Yes/No Questions

The tkinter messagebox object also allows you to ask a user simple yes/no type questions and varies the button names based on the type of question. These functions are:

```

from tkinter import messagebox

answer = messagebox.askokcancel("Question", "Do you want to open this file?")
answer = messagebox.askretrycancel("Question", "Do you want to try that again?")
answer = messagebox.askyesno("Question", "Do you like Python?")
answer = messagebox.askyesnocancel("Question", "Continue playing?")

```

The return value is a Boolean, True or False, answer to the question. If “cancel” is an option and the user selects the “cancel” button, None is returned.

### 15.2.4 Single Value Data Entry

If you want to ask the user for a single data value, either a string, integer, or floating point value, you can use a simpledialog object. A user can enter the requested value and hit “OK”, which will return the entered value. If the user hits “Cancel,” then None is returned.

```

import tkinter as tk
from tkinter import simpledialog

application_window = tk.Tk()

answer = simpledialog.askstring("Input", "What is your first name?", parent=application_window)
if answer is not None:
    print("Your first name is", answer)
else:
    print("You don't have a first name?")

answer = simpledialog.askinteger("Input", "What is your age?", parent=application_window, minvalue=0, maxvalue=100)
if answer is not None:
    print("Your age is", answer)
else:
    print("You don't have an age?")

answer = simpledialog.askfloat("Input", "What is your salary?", parent=application_window,

```

```

        minvalue=0.0,
        maxvalue=100000.0)

if answer is not None:
    print("Your salary is", answer)
else:
    print("You don't have a salary?")

```

### 15.2.5 File Chooser

A common task is to select the names of folders and files on a storage device. This can be accomplished using a `filedialog` object. Note that these commands do not save or load a file. They simply allow a user to select a file. Once you have the file name, you can open, process, and close the file using appropriate Python code. These dialog boxes always return you a “fully qualified file name” that includes a full path to the file. Also note that if a user is allowed to select multiple files, the return value is a tuple that contains all of the selected files. If a user cancels the dialog box, the returned value is an empty string.

```

import tkinter as tk
from tkinter import filedialog
import os

application_window = tk.Tk()

# Build a list of tuples for each file type the file dialog
# should display
my_filetypes = [('all files', '.*'), ('text files', '.txt')]

# Ask the user to select a folder.
answer = filedialog.askdirectory(parent=application_window,
                                  initialdir=os.getcwd(),
                                  title="Please select a folder:")

# Ask the user to select a single file name.
answer =
    filedialog.askopenfilename(parent=application_window,
                               initialdir=os.getcwd(),
                               title="Please select a file:",
                               filetypes=my_filetypes)

# Ask the user to select a one or more file names.
answer =
    filedialog.askopenfilenames(parent=application_window,
                               initialdir=os.getcwd(),
                               title="Please select one or more files:",
                               filetypes=my_filetypes)

# Ask the user to select a single file name for saving.
answer =
    filedialog.asksaveasfilename(parent=application_window,
                                 initialdir=os.getcwd(),
                                 title="Please select a file name for"

```

```
    saving:",
    filetypes=my_filetypes)
```

### 15.2.6 Color Chooser

Tkinter includes a nice dialog box for choosing colors. You provide it with a parent window and an initial color, and it returns a color in two different specifications: 1) a RGB value as a tuple, such as (255, 0, 0) which represents red, and 2) a hexadecimal string used in web pages, such as "#FF0000" which also represents red. If the user cancels the operation, the return values are None and None.

```
from tkinter import colorchooser

rgb_color, web_color =
    colorchooser.askcolor(parent=application_window,
                          initialcolor=(255,
                                         0, 0))
```

## 15.3 GUI Widgets

As we discussed in the introduction, a GUI program allows a user to interact with a computer program using a pointing device that manipulates small pictures called **icons** or **widgets**. The first task of a GUI program is to create the widgets needed for a program's interface. Each widget is designed for specific purposes and your program will be more user friendly if you use each widget according to its intended purpose.

Widgets are basically images on a computer screen and they have a “look-and-feel” depending on the details of how the image is drawn. The “look-and-feel” of a widget is typically controlled by the operating system. For example, GUI programs on a Macintosh computer typically look different from programs on a Microsoft Windows computer. The `tkinter` module implements two versions of widgets: one is “generic,” which makes widgets look the same regardless of what computer your program is running on, and the other implements widgets that emulate a computer’s “look-and-feel”. How you import the `tkinter` module determines which widgets are defined. Using the import statements shown below, the standard convention uses the name `tk` to access the “generic” widgets and the name `ttk` to access the stylized, “look-and-feel” widgets. You always need to import the `tk` functionality because that allows you to create an application window. You can import the `ttk` functionality if you want “look-and-feel” widgets. You can inter-mix the `tk` and `ttk` widgets in an interface if you so choose.

```
# To use the "generic" widgets
import tkinter as tk

# To use the stylized, "look-and-feel" widgets
from tkinter import ttk
```

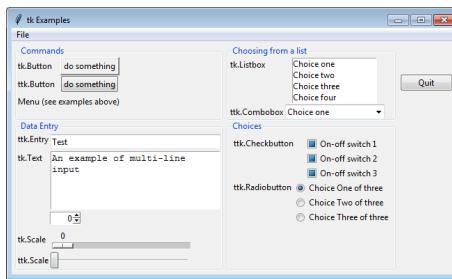
The following two charts list the standard, pre-defined widgets in the `tkinter` module.

The following widgets are used for user input. In some cases you have a choice between the `tk` and `ttk` versions. In other cases you must use the `tk` version because the equivalent `ttk` versions don't exist.

**Table 15.3.1**

Widget	Purpose
<code>tk.Button, ttk.Button</code>	Execute a specific task; a “do this now” command.
<code>tk.Menu</code>	Implements toplevel, pulldown, and popup menus.
<code>ttk.Menubutton</code>	Displays popup or pulldown menu items when activated.
<code>tk.OptionMenu</code>	Creates a popup menu, and a button to display it.
<code>tk.Entry, ttk.Entry</code>	Enter one line of text.
<code>tk.Text</code>	Display and edit formatted text, possibly with multiple lines.
<code>tk.Checkbutton, ttk.Checkbutton</code>	Set on-off, True-False selections.
<code>tk.Radiobutton, ttk.Radiobutton</code>	Allow one-of-many selections.
<code>tk.Listbox</code>	Choose one or more alternatives from a list.
<code>ttk.Combobox</code>	Combines a text field with a pop-down list of values.
<code>tk.Scale, ttk.Scale</code>	Select a numerical value by moving a “slider” along a scale.

The following figure shows examples of these widgets. You can download and run this python program, [all\\_user\\_input\\_widgets.py](#)<sup>1</sup>, to interact with the widgets.

**Figure 15.3.2** Examples of user input widgets

The following widgets display information to a user, but have no user interaction:

**Table 15.3.3**

Widget	Purpose
<code>tk.Label, ttk.Label</code>	Display static text or an image.
<code>tk.Message</code>	Display static multi-line text.
<code>ttk.Separator</code>	Displays a horizontal or vertical separator bar.
<code>ttk.Progressbar</code>	Shows the status of a long-running operation.
<code>ttk.Treeview</code>	Displays a hierarchical collection of items.

You do not need to memorize the above lists, but you should probably re-read the lists again so that you are familiar with what is possible in a TKinter GUI interface. (Note that the TKinter module is customizable, which means that you can create your own widgets, but that is beyond what we will study in these lessons.)

## 15.4 Layout Managers

A widget will not be visible in a window until you assign it a size and location within its parent widget. Assigning a specific size and location to every widget is tedious and error-prone. In addition, the desired behaviour for most GUI interfaces is that the widgets resize and relocate in reasonable ways if their parent window is re-sized. Trust me, you don't want to write code to resize

<sup>1</sup> [..../static/Programs/all\\_user\\_input\\_widgets.py](#)

and relocate widgets every time you develop a GUI program! Therefore, **layout managers** are included in the Tkinter module to do this work for you. You just have to give some basic positioning information to a **layout manager** so it can calculate a position and a size for each widget.

There are three **layout managers** in the Tkinter module:

**Table 15.4.1**

Layout Manager	Description
place	You specify the exact size and position of each widget.
pack	You specify the size and position of each widget <i>relative to each other</i> .
grid	You place widgets in a cell of a 2-dimensional table defined by rows and columns.

You should **never** mix and match these layout managers. Use only one of them for the widget layout within a particular “parent widget”. (Widgets are organized in a hierarchy, which is explained in the next lesson.)

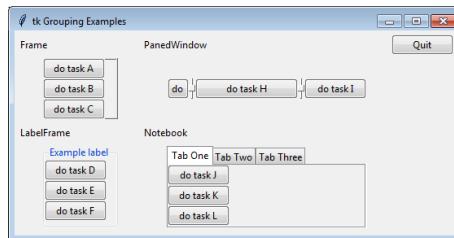
## 15.5 Widget Groupings

You will design a more user friendly interface if you group and organize your widgets in a coherent design. Tkinter has four basic ways to group widgets. These are described in the following table. They are often referred to as “container” widgets because in the widget hierarchy of a GUI program they are the parent widget of a group of related widgets.

**Table 15.5.1**

Widget	Purpose
tk.Frame, ttk.Frame	Create a container for a set of widgets to be displayed as a unit.
ttk.LabelFrame	Group a number of related widgets using a border and a title.
tk.PanedWindow	Group one or more widgets into “panes”, where the “panes” can be re-sized by the user.
ttk.Notebook	A tabbed set of frames, only one of which is visible at any given time.

Widgets are always organized as a hierarchy, where the main application window is the root of the hierarchy. Typically, the child widgets of an application window are a combination of “frames”. The “frames” hold other widgets. A “frame” will not be visible until it is assigned a size and location using a layout manager. The image below shows examples of the four types of widget “containers”. The “containers” in this example used a **grid** layout manager on a 2x2 grid.



**Figure 15.5.2** Examples of grouping widgets

For the **Frame** and **LabelFrame** groups, the frame is the “parent” of the widgets displayed inside the frame. That is, when the buttons were created, the frame was the first parameter to the `tk.Button()` function.

For the **PanedWindow** and **Notebook** groups, you use an `.add(widget)` function to add your widgets to the group. You are still creating a hierarchy of widgets, but the syntax is different.

You can download and run this example program, [all\\_frame\\_widgets.py](#)<sup>1</sup> that allows you to interact with the four types of “containers.”

## 15.6 Command Events

When a user clicks on certain types of widgets, like a button, you typically want a specific action to be performed. This is accomplished by setting the `command` attribute of a widget to a specific event handler function. This can be any function that receives no arguments as parameters. You can set the event handler function using a “named parameter” when you create the widget, or set the widget’s `command` attribute using a dictionary lookup. For example:

```
def my_function():
    print("my_function was called.")

my_button = tk.Button(application_window, text="Example",
                      command=my_function)

# or

my_button = tk.Button(application_window, text="Example")
my_button['command'] = my_function
```

Note that you are setting the `command` property of the widget to a *function reference* – you are not calling the function! Therefore, do not put parentheses after the function name when you make the assignment.

The following widgets define a `command` property which defines a function that is called from the application’s event loop whenever a specific, predefined event is performed by a user.

**Table 15.6.1**

Widget	The user event that causes the <code>command</code> function to be executed:
Button	The user places their pointing device cursor over the button and presses and releases the left mouse button.
Checkbutton	If the state of the check box changes, the function is called.
Radiobutton	If the state of the radio box changes, the function is called.
Scale	The function is called if the slider moves. The function is passed one argument, the new scale value.

The following widgets do **not** have a `command` property, but they use other properties to respond to user events:

**Table 15.6.2**

Widget	Property	User events:
Menu	<code>postcommand</code>	Every time someone brings up this menu.
Combobox	<code>postcommand</code>	When the user clicks on the down-arrow.
Combobox	<code>validatecommand</code>	Dynamically validate the widget’s text content.
Entry	<code>validatecommand</code>	Dynamically validate the widget’s text content.

Note that the `Label`, `Message`, and `Separator` widgets do not respond to user events and therefore have no associated event handlers.

## 15.7 Low-Level Event Processing

In the previous lessons you learned how to associate simple user actions with the most common user events, such as clicking on a button. For simple GUI

---

<sup>1</sup> [../static/Programs/all\\_frame\\_widgets.py](#)

programs this is typically sufficient. But for more complex programs you might need to process “lower level” events, such as recognizing when the user’s cursor is over a widget, or when a widget becomes the focus of user input. To handle these types of events you need to implement **event binding**. This lesson provides an overview of how to process any event that happens on a computer. We will not use these techniques in most of the simple GUI programs we discuss, but you should be aware of what is possible.

## 15.8 The Design of GUI Programs

For very simple GUI programs, no special program design is needed, as demonstrated in the previous “Hello World” example programs. However, any non-trivial GUI program will require extensive use of global variables if the structure of the code does not use a Python **class**. You have learned in previous lessons that global variables are bad because they make debugging programs more difficult. Therefore we want a design for GUI programs that avoids global variables as much as possible.

To demonstrate this, let’s look at two versions of a simple program that increments a counter each time a user clicks a button. The first version of this code does not use a **class** definition and requires that a global variable called `my_counter` be used. This is because the label that represents the counter is created in the `create_user_interface` function but it must be accessed in the event handler function `increment_counter`. In fact, the event handlers of a GUI program almost always need access to multiple widgets in the program’s interface and the values can’t be passed as parameters because an `command` event handler function receives no parameters and a `bind` event handler function receives exactly one parameter – an event object. Study the following example and pay close attention to where the `my_counter` global variable is used.

```
import tkinter as tk
from tkinter import ttk

global my_counter

def create_user_interface(application_window):
    global my_counter

    my_counter = ttk.Label(application_window, text="0")
    my_counter.grid(row=0, column=0)

    increment_button = ttk.Button(application_window,
        text="Add 1 to counter")
    increment_button.grid(row=1, column=0)
    increment_button['command'] = increment_counter

    quit_button = ttk.Button(application_window,
        text="Quit")
    quit_button.grid(row=2, column=0)
    quit_button['command'] = window.destroy

def increment_counter():
    global my_counter
    my_counter['text'] = str(int(my_counter['text']) + 1)
```

```

# Create the application window
window = tk.Tk()

create_user_interface(window)

# Start the GUI event loop
window.mainloop()

```

Let's compare the above program to an identical application that is designed as a Python `class`. The `class` encapsulates all of the values needed for the GUI interface and the event handlers and we don't need global variables!

```

import tkinter as tk
from tkinter import ttk

def main():
    # Create the entire GUI program
    program = CounterProgram()

    # Start the GUI event loop
    program.window.mainloop()

class CounterProgram:

    def __init__(self):
        self.window = tk.Tk()
        self.my_counter = None # All attributes should be
                               # initialize in init
        self.create_widgets()

    def create_widgets(self):
        self.my_counter = ttk.Label(self.window, text="0")
        self.my_counter.grid(row=0, column=0)

        increment_button = ttk.Button(self.window,
                                      text="Add 1 to counter")
        increment_button.grid(row=1, column=0)
        increment_button['command'] = self.increment_counter

        quit_button = ttk.Button(self.window, text="Quit")
        quit_button.grid(row=2, column=0)
        quit_button['command'] = self.window.destroy

    def increment_counter(self):
        self.my_counter['text'] =
            str(int(self.my_counter['text']) + 1)

if __name__ == "__main__":
    main()

```

Notice the following about this design:

- The application's window is created in the constructor (`__init__`) of the `CounterProgram` class and then the interface widgets are created by a call to `create_widgets`.
- The event handler, `increment_counter` can access the label `self.my_counter` using the object's attributes.

- The code creates an instance of the class CounterProgram and starts the GUI event-loop.

It is recommended that you develop all of your GUI programs as Python Classes. For complex designs, a Python Class can help manage the complexity of the code and the scoping of variables.

## 15.9 Common Widget Properties

Each widget has a set of properties that define its visual appearance on the computer screen and how it responds to user events. There is a set of properties that all tk widgets have in common. Some of these are shown in the following table. Remember that ttk widgets match the “look and feel” of the computer that is running your program, so there are limited attributes you can change for ttk widgets. See the ttk style attribute information at <https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-style-layer.html><sup>1</sup> if you want to modify ttk widgets.)

**Table 15.9.1**

Common Widget Properties	Description
bg	Background color.
fg	Foreground color.
width	Width in pixels
height	Height in pixels
borderwidth	The size of the border in pixels.
text	Text displayed on the widget.
font	The font used for text on the widget.
cursor	The shape of the cursor when the cursor is over the widget.
activeforeground	The color of the text when the widget is activated.
activebackground	The color of the background when the widget is activated.
image	An image to be displayed on the widget.

You can treat a widget object as a dictionary and use the property names as keys to access and change the property values. For example, to change the background color and width of a widget whose object variable is named sam, you could do this:

```
sam = tk.Button(application_window, text="Sam's Button")
sam['bg'] = 'red'
sam['width'] = 60 # pixels
```

## 15.10 Timer Events

### 15.10.1 Introduction

GUI programs run an “event loop” that continuously receive events from the operating system and “dispatches” those events to appropriate callback functions. Nothing happens in a GUI program without an event. The application logic for some problems requires that specific events happen at some specific times. For this reason tkinter includes a feature to generate events under software control. These are often referred to as *timer events*.

---

<sup>1</sup><https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/ttk-style-layer.html>

Every widget has an `after` method that will generate an event at a specific time interval from the time it is called. The method takes at least 2 arguments: the amount of time (in milliseconds) to wait before generating the event, and the callback function to call after the time has elapsed. In the example below, the function `a_callback_function` will be called one second (1000 milliseconds) after the timer-event was created.

```
def a_callback_function():
    print("a_callback_function was called.")

my_button = tk.Button(application_window, text="Example")
my_button.after(1000, a_callback_function)
```

### 15.10.2 Animations and Repeated Tasks

If you want a specific task to be repeated on a regular interval, then the callback function that performs the task should create a new timer event each time it is called. The following example creates a callback function that is called 30 times per second. (Note that 1/30th of a second is 0.033 seconds, or 33 milliseconds.)

```
def animate():
    # Draw something
    my_button.after(33, animate)

my_button = tk.Button(application_window, text="Example")
my_button.after(33, animate)
```

A widget can have more than one timer event active at any time. In fact there is no limit to the number of timer events you can create.

Note that you should never use a loop to repeat a task in a GUI program. If you use a loop, the event-loop will be prevented from execution and no events will be processed while the loop is running. Always use a timer event for repeating a task, especially if a single execution of the task takes a considerable amount of CPU time.

### 15.10.3 Canceling Timer Events

In some cases you may need to cancel a timer event to prevent it from executing. This is straightforward using the `after_cancel` method. Remember that a widget can have multiple active timers, so the `after_cancel` method requires one parameter which specifies which timer event to cancel. If you need to cancel an event, you must capture the return value from the call to `after` when you created the event. Here is an example:

```
def do_something():
    # Some processing

my_button = tk.Button(application_window, text="Example")
timer_object = my_button.after(1000, do_something)
...
my_button.after_cancel(timer_object)
```

### 15.10.4 Multiple Parameters to Timer Callbacks

Timer callback function can have zero or more arguments passed to them when they are called. You specify the arguments when you create the event. This makes timer callback functions extremely flexible. Below is an example of

three different callback functions, each of which receives a different number of arguments. You must specify the correct number of arguments for the callback function when you create the timer event.

```
def task1():
    # Do some processing

def task2(alpha):
    # Do some processing

def task3(beta, gamma):
    # Do some processing

my_button = tk.Button(application_window, text="Example")
my_button.after(1000, task1)
my_button.after(2000, task2, 3)      # 3 gets passed to the
                                    # parameter alpha
my_button.after(5000, task3, a, b) # a gets passed to the
                                    # parameter beta
                                    # b gets passed to the
                                    # parameter gamma
```

## 15.11 A Programming Example

### 15.11.1 Introduction

Let's develop a non-trivial GUI program to demonstrate the material presented in the previous lessons. We will develop a GUI Whack-a-mole game where a user tries to click on "moles" as they randomly pop up out of the "ground."

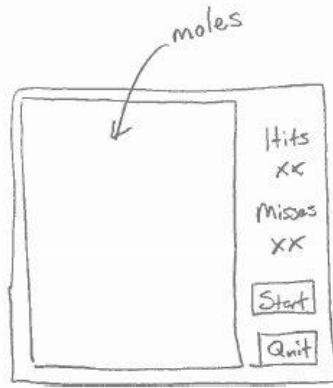
This discussion will take you through an *incremental development* cycle that creates a program in well-defined stages. While you might read a finished computer program from "top to bottom," that is not how it was developed. For a typical GUI program development, you are encouraged to go through these stages:

- 1 Using scratch paper, physically draw a rough sketch of your user interface.
- 2 Create the basic structure of your program and create the major frames that will hold the widgets needed for your program's interface. Give the frames an initial size and color so that you can visually see them, given that there are no widgets inside of them to determine their size.
- 3 Incrementally add all of the widgets you need for your program and size and position them appropriately.
- 4 Create your callback functions, stub them out, and assign them to appropriate events. Verify that the events are executing the correct functions.
- 5 Incrementally implement the functionality needed for each callback function.

When you develop code using *incremental development* your program should always be executable. You continually add a few lines of code and then test them. If errors occur you almost always know where the errors came from! They came from the lines of code you just added.

### 15.11.2 A Whack-a-mole Game

Step 1: Make sure you have a reasonable GUI design and implementation plan before you start coding. Draw a sketch of your initial design on paper and consider how a user will interact with your program.



**Figure 15.11.1** Initial design of a Whack-a-mole game

Step 2: Create the basic structure of your interface using appropriate `frame` widgets. You will need to give a size to the frames because they will contain no widgets, which is how a frame typically gets its size. It is also suggested that you give each frame a unique color so it is easy to see the area of the window it covers. Here is a basic start for our whack-a-mole game ([whack\\_a\\_mole\\_v1.py](#)<sup>1</sup>):

```
def main():
    # Create the entire GUI program
    program = WhackAMole()

    # Start the GUI event loop
    program.window.mainloop()

class WhackAMole:

    def __init__(self):
        self.window = tk.Tk()
        self.mole_frame, self.status_frame =
            self.create_frames()

    def create_frames(self):
        mole_frame = tk.Frame(self.window, bg='red',
                              width=300, height=300)
        mole_frame.grid(row=1, column=1)

        status_frame = tk.Frame(self.window, bg='green',
                               width=100, height=300)
        status_frame.grid(row=1, column=2)

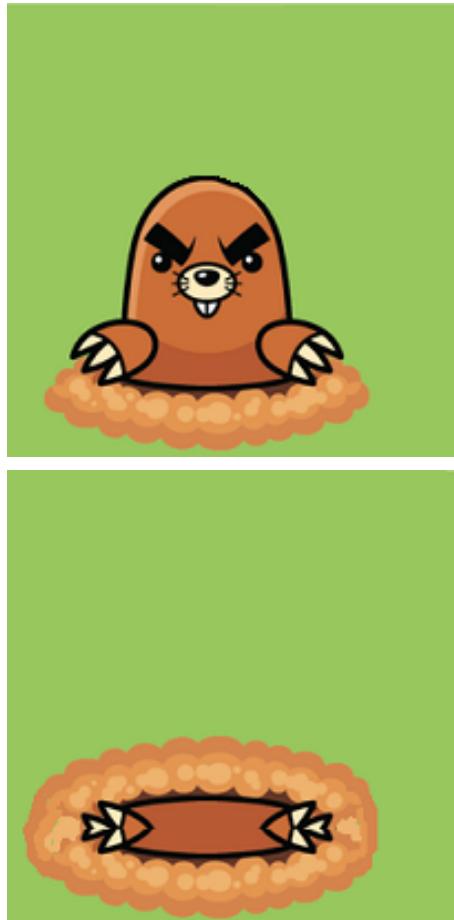
        return mole_frame, status_frame

if __name__ == "__main__":
    main()
```

---

<sup>1</sup>[..../static/Programs/whack\\_a\\_mole\\_v1.py](#)

Step 3: Incrementally add appropriate widgets to each frame. Don't attempt to add all the widgets at once. The initial design conceptualized the moles as buttons, so a grid of buttons was added to the left frame, one button for each mole. The exact size of the "mole field" needs to be determined at a future time, so initialize a CONSTANT that can be used to easily change it later. ([whack\\_a\\_mole\\_v2.py<sup>2</sup>](#))



```
import tkinter as tk
from tkinter import PhotoImage

def main():
    # Create the entire GUI program
    program = WhackAMole()

    # Start the GUI event loop
    program.window.mainloop()

class WhackAMole:
    NUM_MOLES_ACROSS = 4

    def __init__(self):
        self.window = tk.Tk()
        self.mole_frame, self.status_frame =
            self.create_frames()
        self.mole_photo = PhotoImage(file="mole.png")
```

<sup>2</sup>../\_static/Programs/whack\_a\_mole\_v2.py

```

        self.mole_buttons = self.create_moles()

    def create_frames(self):
        mole_frame = tk.Frame(self.window, bg='red')
        mole_frame.grid(row=1, column=1)

        status_frame = tk.Frame(self.window, bg='green',
                               width=100)
        status_frame.grid(row=1, column=2, sticky=tk.E +
                          tk.W + tk.N + tk.S)

        return mole_frame, status_frame

    def create_moles(self):
        # Source of mole image:
        # https://play.google.com/store/apps/details?id=genergame.molehammer

        mole_buttons = []
        for r in range(WhackAMole.NUM_MOLES_ACROSS):
            row_of_buttons = []
            for c in range(WhackAMole.NUM_MOLES_ACROSS):
                mole_button = tk.Button(self.mole_frame,
                                       image=self.mole_photo)
                mole_button.grid(row=r, column=c, padx=8,
                                 pady=8)

                row_of_buttons.append(mole_button)

            mole_buttons.append(row_of_buttons)

        return mole_buttons

if __name__ == "__main__":
    main()

```

Continue to add appropriate widgets for the right frame. The final result is shown below, but recognize that it was developed little by little. ([whack\\_a\\_mole\\_v3.py](#)<sup>3</sup>)

```

import tkinter as tk
from tkinter import PhotoImage

def main():
    # Create the entire GUI program
    program = WhackAMole()

    # Start the GUI event loop
    program.window.mainloop()

class WhackAMole:
    STATUS_BACKGROUND = "white"
    NUM_MOLES_ACROSS = 4

    def __init__(self):
        self.window = tk.Tk()
        self.mole_frame, self.status_frame =
            self.create_frames()
        self.mole_photo = PhotoImage(file="mole.png")
        self.mole_buttons = self.create_moles()

```

---

<sup>3</sup>../\_static/Programs/whack\_a\_mole\_v3.py

```
        self.hit_counter, self.miss_counter,
        self.start_button \
    = self.create_status_widgets()

def create_frames(self):
    mole_frame = tk.Frame(self.window, bg='red')
    mole_frame.grid(row=1, column=1)

    status_frame = tk.Frame(self.window,
                           bg=WhackAMole.STATUS_BACKGROUND)
    status_frame.grid(row=1, column=2, sticky=tk.N +
                      tk.S + tk.W + tk.E)

    return mole_frame, status_frame

def create_moies(self):
    # Source of mole image:
    # https://play.google.com/store/apps/details?id=genergame.molehammer

    mole_buttons = []
    for r in range(WhackAMole.NUM_MOLES_ACROSS):
        row_of_buttons = []
        for c in range(WhackAMole.NUM_MOLES_ACROSS):
            mole_button = tk.Button(self.mole_frame,
                                   image=self.mole_photo)
            mole_button.grid(row=r, column=c, padx=8,
                             pady=8)

        row_of_buttons.append(mole_button)
        mole_buttons.append(row_of_buttons)

    return mole_buttons

def create_status_widgets(self):
    spacer = tk.Label(self.status_frame, text="",
                      bg=WhackAMole.STATUS_BACKGROUND)
    spacer.pack(side="top", fill=tk.Y, expand=True)

    hit_label = tk.Label(self.status_frame,
                         text="Number of Hits",
                         bg=WhackAMole.STATUS_BACKGROUND)
    hit_label.pack(side="top", fill=tk.Y, expand=True)

    hit_counter = tk.Label(self.status_frame, text="0",
                           bg=WhackAMole.STATUS_BACKGROUND)
    hit_counter.pack(side="top", fill=tk.Y, expand=True)

    spacer = tk.Label(self.status_frame, text="",
                      bg=WhackAMole.STATUS_BACKGROUND)
    spacer.pack(side="top", fill=tk.Y, expand=True)

    miss_label = tk.Label(self.status_frame,
                          text="Number of Misses",
                          bg=WhackAMole.STATUS_BACKGROUND)
    miss_label.pack(side="top", fill=tk.Y, expand=True)

    miss_counter = tk.Label(self.status_frame,
```

```

        text="0", bg=WhackAMole.STATUS_BACKGROUND)
miss_counter.pack(side="top", fill=tk.Y,
                  expand=True)

spacer = tk.Label(self.status_frame, text="",
                  bg=WhackAMole.STATUS_BACKGROUND)
spacer.pack(side="top", fill=tk.Y, expand=True)

start_button = tk.Button(self.status_frame,
                        text="Start")
start_button.pack(side="top", fill=tk.Y,
                  expand=True, ipadx=10)

spacer = tk.Label(self.status_frame, text="",
                  bg=WhackAMole.STATUS_BACKGROUND)
spacer.pack(side="top", fill=tk.Y, expand=True)

quit_button = tk.Button(self.status_frame,
                        text="Quit")
quit_button.pack(side="top", fill=tk.Y,
                  expand=True, ipadx=10)

spacer = tk.Label(self.status_frame, text="",
                  bg=WhackAMole.STATUS_BACKGROUND)
spacer.pack(side="top", fill=tk.Y, expand=True)

return hit_counter, miss_counter, start_button

if __name__ == "__main__":
    main()

```

Step 4: Create a callback function for each event that will cause something to happen in your program. Stub these functions out with a single print statement in each one. Bind an event to each callback function. Now test your program and make sure each event causes the correct print-line in the Python console. ([whack\\_a\\_mole\\_v4.py<sup>4</sup>](#))

```

import tkinter as tk
from tkinter import PhotoImage

def main():
    # Create the entire GUI program
    program = WhackAMole()

    # Start the GUI event loop
    program.window.mainloop()

class WhackAMole():
    STATUS_BACKGROUND = "white"
    NUM_MOLES_ACROSS = 4

    def __init__(self):
        self.window = tk.Tk()
        self.mole_frame, self.status_frame =
            self.create_frames()
        self.mole_photo = PhotoImage(file="mole.png")
        self.mole_buttons = self.create_molets()

```

---

<sup>4</sup>../\_static/Programs/whack\_a\_mole\_v4.py

```
        self.hit_counter, self.miss_counter,
        self.start_button, self.quit_button \
    = self.create_status_widgets()

    self.set_callbacks()

def create_frames(self):
    mole_frame = tk.Frame(self.window, bg='red')
    mole_frame.grid(row=1, column=1)

    status_frame = tk.Frame(self.window,
                           bg=WhackAMole.STATUS_BACKGROUND)
    status_frame.grid(row=1, column=2, sticky=tk.E +
                      tk.W + tk.N + tk.S)

    return mole_frame, status_frame

def create_molets(self):
    # Source of mole image:
    # https://play.google.com/store/apps/details?id=genergame.molehammer

    mole_buttons = []
    for r in range(WhackAMole.NUM_MOLES_ACROSS):
        row_of_buttons = []
        for c in range(WhackAMole.NUM_MOLES_ACROSS):
            mole_button = tk.Button(self.mole_frame,
                                   image=self.mole_photo)
            mole_button.grid(row=r, column=c, padx=8,
                             pady=8)

        row_of_buttons.append(mole_button)
        mole_buttons.append(row_of_buttons)

    return mole_buttons

def create_status_widgets(self):
    spacer = tk.Label(self.status_frame, text="",
                      bg=WhackAMole.STATUS_BACKGROUND)
    spacer.pack(side="top", fill=tk.Y, expand=True)

    hit_label = tk.Label(self.status_frame,
                         text="Number of Hits",
                         bg=WhackAMole.STATUS_BACKGROUND)
    hit_label.pack(side="top", fill=tk.Y, expand=True)

    hit_counter = tk.Label(self.status_frame, text="0",
                           bg=WhackAMole.STATUS_BACKGROUND)
    hit_counter.pack(side="top", fill=tk.Y, expand=True)

    spacer = tk.Label(self.status_frame, text="",
                      bg=WhackAMole.STATUS_BACKGROUND)
    spacer.pack(side="top", fill=tk.Y, expand=True)

    miss_label = tk.Label(self.status_frame,
                          text="Number of Misses",
                          bg=WhackAMole.STATUS_BACKGROUND)
    miss_label.pack(side="top", fill=tk.Y, expand=True)
```

```

miss_counter = tk.Label(self.status_frame,
    text="0", bg=WhackAMole.STATUS_BACKGROUND)
miss_counter.pack(side="top", fill=tk.Y,
    expand=True)

spacer = tk.Label(self.status_frame, text="",
    bg=WhackAMole.STATUS_BACKGROUND)
spacer.pack(side="top", fill=tk.Y, expand=True)

start_button = tk.Button(self.status_frame,
    text="Start")
start_button.pack(side="top", fill=tk.Y,
    expand=True, ipadx=10)

spacer = tk.Label(self.status_frame, text="",
    bg=WhackAMole.STATUS_BACKGROUND)
spacer.pack(side="top", fill=tk.Y, expand=True)

quit_button = tk.Button(self.status_frame,
    text="Quit")
quit_button.pack(side="top", fill=tk.Y,
    expand=True, ipadx=10)

spacer = tk.Label(self.status_frame, text="",
    bg=WhackAMole.STATUS_BACKGROUND)
spacer.pack(side="top", fill=tk.Y, expand=True)

return hit_counter, miss_counter, start_button,
    quit_button

def set_callbacks(self):
    # Set the same callback for each mole button
    for r in range(WhackAMole.NUM_MOLES_ACROSS):
        for c in range(WhackAMole.NUM_MOLES_ACROSS):
            self.mole_buttons[r][c]['command'] =
                self.mole_hit

            self.start_button['command'] = self.start
            self.quit_button['command'] = self.quit

def mole_hit(self):
    print("mole_button_hit")

def start(self):
    print("start_button_hit")

def quit(self):
    print("quit_button_hit")

if __name__ == "__main__":
    main()

```

Step 5: Add appropriate functionality to the callback functions. This is where the functional logic of your particular application resides. In the case of our whack-a-mole game, we need to be able to count the number of times a user clicks on a mole when it is visible. And we need the moles to appear and disappear at random intervals. Originally each mole was a button widget, but the border around each button was distracting, so they were changed to label widgets. Two images were used to represent a mole: one image is a solid color

that matches the frame's background, and the other image is a picture of a mole. By replacing the image used for each label we can make the moles visible or invisible. A label normally does not have an associated callback, so we bind a left mouse click event, "<ButtonPress-1>" to each label. We can determine whether the mouse click is a "hit" or a "miss" by examining the label under the click to see which image is currently set to the label. We use timer events to change the image on each label. Also notice the use of a messagebox to protect the program from accidental quitting. The end result is shown below. ([whack\\_a\\_mole\\_v5.py](#)<sup>5</sup>)

```

import tkinter as tk
from tkinter import PhotoImage
from tkinter import messagebox
from random import randint

def main():
    # Create the entire GUI program
    program = WhackAMole()

    # Start the GUI event loop
    program.window.mainloop()

class WhackAMole:
    STATUS_BACKGROUND = "white"
    NUM_MOLES_ACROSS = 4
    MIN_TIME_DOWN = 1000
    MAX_TIME_DOWN = 5000
    MIN_TIME_UP = 1000
    MAX_TIME_UP = 3000

    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Whack-a-mole")

        self.mole_frame, self.status_frame =
            self.create_frames()

        self.mole_photo = PhotoImage(file="mole.png")
        self.mole_cover_photo =
            PhotoImage(file="mole_cover.png")
        self.label_timers = {}

        self.mole_labels = self.create_moles()

        self.hit_counter, self.miss_counter,
            self.start_button, self.quit_button \
            = self.create_status_widgets()

        self.set_callbacks()
        self.game_is_running = False

    def create_frames(self):
        mole_frame = tk.Frame(self.window)
        mole_frame.grid(row=0, column=0)

        status_frame = tk.Frame(self.window,
                               bg=WhackAMole.STATUS_BACKGROUND)

```

---

<sup>5</sup>../\_static/Programs/whack\_a\_mole\_v5.py

```
status_frame.grid(row=0, column=1, sticky=tk.E +
    tk.W + tk.N + tk.S,
    ipadx=6)

return mole_frame, status_frame

def create_moles(self):
    # Source of mole image:
    # https://play.google.com/store/apps/details?id=genergame.molehammer

    mole_labels = []
    for r in range(WhackAMole.NUM_MOLES_ACROSS):
        row_of_labels = []
        for c in range(WhackAMole.NUM_MOLES_ACROSS):
            mole_label = tk.Label(self.mole_frame,
                image=self.mole_photo)
            mole_label.grid(row=r, column=c,
                sticky=tk.E + tk.W + tk.N + tk.S)
            self.label_timers[id(mole_label)] = None

        row_of_labels.append(mole_label)

    mole_labels.append(row_of_labels)

    return mole_labels

def create_status_widgets(self):
    spacer = tk.Label(self.status_frame, text="",
                      bg=WhackAMole.STATUS_BACKGROUND)
    spacer.pack(side="top", fill=tk.Y, expand=True)

    hit_label = tk.Label(self.status_frame,
        text="Number of Hits",
        bg=WhackAMole.STATUS_BACKGROUND)
    hit_label.pack(side="top", fill=tk.Y, expand=True)

    hit_counter = tk.Label(self.status_frame, text="0",
                           bg=WhackAMole.STATUS_BACKGROUND)
    hit_counter.pack(side="top", fill=tk.Y, expand=True)

    spacer = tk.Label(self.status_frame, text="",
                      bg=WhackAMole.STATUS_BACKGROUND)
    spacer.pack(side="top", fill=tk.Y, expand=True)

    miss_label = tk.Label(self.status_frame,
        text="Number of Misses",
        bg=WhackAMole.STATUS_BACKGROUND)
    miss_label.pack(side="top", fill=tk.Y, expand=True)

    miss_counter = tk.Label(self.status_frame, text="0",
                           bg=WhackAMole.STATUS_BACKGROUND)
    miss_counter.pack(side="top", fill=tk.Y,
                      expand=True)

    spacer = tk.Label(self.status_frame, text="",
                      bg=WhackAMole.STATUS_BACKGROUND)
    spacer.pack(side="top", fill=tk.Y, expand=True)

    start_button = tk.Button(self.status_frame,
```

```
        text="Start")
start_button.pack(side="top", fill=tk.Y,
                  expand=True, ipadx=10)

spacer = tk.Label(self.status_frame, text="",
                   bg=WhackAMole.STATUS_BACKGROUND)
spacer.pack(side="top", fill=tk.Y, expand=True)

quit_button = tk.Button(self.status_frame,
                       text="Quit")
quit_button.pack(side="top", fill=tk.Y,
                 expand=True, ipadx=10)

spacer = tk.Label(self.status_frame, text="",
                   bg=WhackAMole.STATUS_BACKGROUND)
spacer.pack(side="top", fill=tk.Y, expand=True)

return hit_counter, miss_counter, start_button,
       quit_button

def set_callbacks(self):
    # Set the same callback for each mole label
    for r in range(WhackAMole.NUM_MOLES_ACROSS):
        for c in range(WhackAMole.NUM_MOLES_ACROSS):
            self.mole_labels[r][c].bind("<ButtonPress-1>",
                                         self.mole_hit)

    self.start_button['command'] = self.start
    self.quit_button['command'] = self.quit

def mole_hit(self, event):

    if self.game_is_running:
        hit_label = event.widget
        if hit_label['image'] ==
            self.mole_cover_photo.name:
            # MISSED! Update the miss counter
            self.miss_counter['text'] =
                str(int(self.miss_counter['text']) + 1)
        else:
            # HIT! Update the hit counter
            self.hit_counter['text'] =
                str(int(self.hit_counter['text']) + 1)
            # Remove the mole and don't update the miss
            # counter
            self.put_down_mole(hit_label, False)

def start(self):
    if self.start_button['text'] == 'Start':
        # Change all the mole images to a blank image
        # and
        # set a random time for the moles to re-appear
        # on each label.
        for r in range(WhackAMole.NUM_MOLES_ACROSS):
            for c in range(WhackAMole.NUM_MOLES_ACROSS):
                the_label = self.mole_labels[r][c]
                the_label['image'] =
                    self.mole_cover_photo
                time_down =
```

```

        randint(WhackAMole.MIN_TIME_DOWN,
                WhackAMole.MAX_TIME_DOWN)
    timer_object =
        the_label.after(time_down,
                        self.pop_up_mole,
                        the_label)
    self.label_timers[id(the_label)] =
        timer_object

    self.game_is_running = True
    self.start_button['text'] = "Stop"

    self.hit_counter['text'] = "0"
    self.miss_counter['text'] = "0"

else: # The game is running, so stop the game and
    reset everything
    # Show every mole and stop all the timers
    for r in range(WhackAMole.NUM_MOLES_ACROSS):
        for c in range(WhackAMole.NUM_MOLES_ACROSS):
            the_label = self.mole_labels[r][c]
            # Show the mole
            the_label['image'] = self.mole_photo
            # Delete any timer that is associated
            # with the mole
            the_label.after_cancel(self.label_timers[id(the_label)])

    self.game_is_running = False
    self.start_button['text'] = "Start"

def put_down_mole(self, the_label, timer_expired):

    if self.game_is_running:
        if timer_expired:
            # The mole is going down before it was
            # clicked on, so update the miss counter
            self.miss_counter['text'] =
                str(int(self.miss_counter['text']) + 1)
        else:
            # The timer did not expire, so manually
            # stop the timer
            the_label.after_cancel(self.label_timers[id(the_label)])

        # Make the mole invisible
        the_label['image'] = self.mole_cover_photo

        # Set a call to pop up the mole in the future
        time_down = randint(WhackAMole.MIN_TIME_DOWN,
                            WhackAMole.MAX_TIME_DOWN)
        timer_object = the_label.after(time_down,
                                       self.pop_up_mole,
                                       the_label)
        # Remember the timer object so it can be
        # canceled later, if need be
        self.label_timers[id(the_label)] = timer_object

def pop_up_mole(self, the_label):
    # Show the mole on the screen
    the_label['image'] = self.mole_photo

```

```

        if self.game_is_running:
            # Set a call to make the mole disappear in the
            # future
            time_up = randint(WhackAMole.MIN_TIME_UP,
                               WhackAMole.MAX_TIME_UP)
            timer_object = the_label.after(time_up,
                                           self.put_down_mole,
                                           the_label, True)
            self.label_timers[id(the_label)] = timer_object

    def quit(self):
        really_quit = messagebox.askyesno("Quiting?", "Do you really want to quit?")
        if really_quit:
            self.window.destroy()

if __name__ == "__main__":
    main()

```

### 15.11.3 Summary

We developed a complete GUI application in 5 well-designed stages. Hopefully you see the value in incremental software development.

However, the end result is not necessarily easy to understand or modify for future enhancements. The next lesson will introduce a scheme for breaking complete software into more manageable pieces.

## 15.12 Managing GUI Program Complexity

### 15.12.1 Introduction

As we explained in a previous lesson, GUI programs are best implemented as Python classes because it allows you to manage the scope of the variables in your GUI interface and callback functions. However, as GUI programs become more complex, it can become overwhelming to implement them as a single class. If a single class has over than 2,000 lines of code it is probably getting too big to effectively manage. What are some ways to effectively break down complex problems into manageable pieces?

One of the most widely used ways to break down a GUI program into manageable pieces is called the Model-View-Controller software design pattern. This is often abbreviated as **MVC** (Model-View-Controller). It divides a problem into three pieces:

- Model - the *model* directly manages an application's data and logic. If the model changes, the *model* sends commands to update the user's view.
- View - the *view* presents the results of the application to the user. It is in charge of all program output.
- Controller - the *controller* accepts all user input and sends commands to the model to change the model's state.

To says this in more general terms, the *controller* manages the applications input, the *model* manages the application's “state” and enforces application consistency, and the *view* updates the output, which is what the user sees on the screen. This is basically identical to what all computer processing is composed of, which is:

```
|   input --> processing --> output
```

The MVC design pattern renames the pieces and restricted which part of the code can talk to the other parts of code. For MVC design:

```
|   controller (input) --> model (processing) --> view (output)
```

From the perspective of a GUI program, this means that the callback functions, which are called when a user causes events, are the *controller*, the *model* should perform all of the application logic, and the building and modification of the GUI widgets composes the *view*.

Let's develop a Whack-a-mole game program using this design strategy. Instead of creating one Python Class for the entire game, the code will be developed as a set of cooperating objects. So where should we begin? I would suggest that the same stages of development we used in the previous lesson are a good approach, but we will create a separate Python *class* for most of the stages. Let's walk through the code development.

### 15.12.2 Creating the *View*

Let's create a Python *class* that builds the user interface for a Whack-a-mole game. The emphasis for this code is the creation of the widgets we need to display to the user. For this version let's allow the moles to be placed at random locations inside the left frame. To do this we must specify an exact size for the left frame. Otherwise the code is the same as the previous version.

```
|   Code
```

### 15.12.3 Creating the *Model*

The *model* for this Whack-a-mole game is fairly simple. We need to keep a counter for the number of user hits on moles that are visible, and a counter for the number of times a user clicks on a mole that is not visible (or just clicks on the left frame and not a mole widget.)

### 15.12.4 Creating the *Controller*

The *controller* receives user events and sends messages to the *controller* to update the *model*'s state. For our Whack-a-mole game, we have the following four basic commands we need to send to the *model*:

- A user clicked on something on the left frame.
- The user wants to start a new game. (The user clicked on the “Start” button.)
- The user wants to stop playing a game. (The user clicked on the “Stop” button.)
- The user wants to quit the application. (The user clicked on the “Quit” button.)

The *controller\** needs to recognize these events and send them to appropriate methods in the *model*. The *controller* needs to define callback functions for these events and register the appropriate event with the appropriate callback. Therefore, the *controller* needs access to the widgets in the *view* object. This can easily be accomplished by passing a reference to the *view* object to the *controller* when it is created. Summary ——

## 15.13 Exercises

This page left intentionally blank

## 15.14 Glossary

### Glossary

**icon.** A small picture that represents some functionality in a computer program. A user clicks on an icon to cause an action to be performed by a program.

**widget.** A visual element of a graphical user interface that allows a user to give commands to an executing program. Example widgets include a `command button`, a `slider bar`, or a `list box`.

**graphical user interface.** A user interacts with a computer program by pointing, clicking, and dragging icons and widgets in a window on a computer screen.

**GUI.** An abbreviation for a “graphical user interface.”

**event-driven programming.** A program that only executes tasks when a user specially requests a task.

**event loop.** A built-in function of a GUI toolkit that “listens” for operating system events and then calls an appropriate event-handler for each event.

**event-handler.** A function that processes an event. These functions are also called `callback functions`.

# Chapter 16

## Recursion

### 16.1 What Is Recursion?

**Recursion** is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

### 16.2 Calculating the Sum of a List of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a list of numbers such as: [1, 3, 5, 7, 9]. An iterative function that computes the sum is shown below. The function uses an accumulator variable (`theSum`) to compute a running total of all the numbers in the list by starting with 0 and adding each number in the list.

```
def listsum(numList):
    theSum = 0
    for i in numList:
        theSum = theSum + i
    return theSum

print(listsum([1,3,5,7,9]))
```

Pretend for a minute that you do not have `while` loops or `for` loops. How would you compute the sum of a list of numbers? If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers. To redefine the problem from adding a list to adding pairs of numbers, we could rewrite the list as a fully parenthesized expression. Such an expression looks like this: (((1 + 3) + 5) + 7) + 9) We can also parenthesize the expression the other way around,(1 + (3 + (5 + (7 + 9)))) Notice that the innermost set of parentheses, (7 + 9), is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.  
total = \ (1 + (3 + (5 + (7 + 9)))) \\ total = \ (1 + (3 + (5 + 16))) \\ total = \ (1 + (3 + 21)) \\ total = \ (1 + 24) \\ total = \ 25 How can we take this idea and turn it into a Python program? First, let's restate the sum problem in terms of

Python lists. We might say the the sum of the list `numList` is the sum of the first element of the list (`numList[0]`), and the sum of the numbers in the rest of the list (`numList[1:]`). To state it in a functional form:  

$$\text{listSum}(\text{numList}) = \text{first}(\text{numList}) + \text{listSum}(\text{rest}(\text{numList}))$$

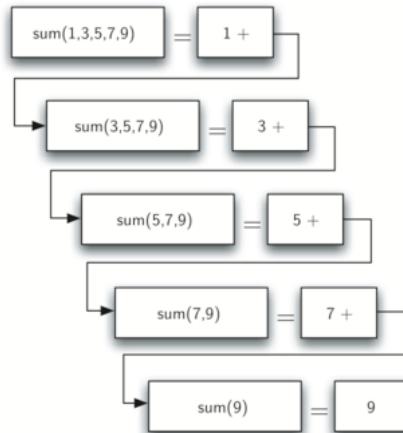
In this equation `first(numList)` returns the first element of the list and `rest(numList)` returns a list of everything but the first element. This is easily expressed in Python.

```
def listsum(numList):
    if len(numList) == 1:
        return numList[0]
    else:
        return numList[0] + listsum(numList[1:])

print(listsum([1,3,5,7,9]))
```

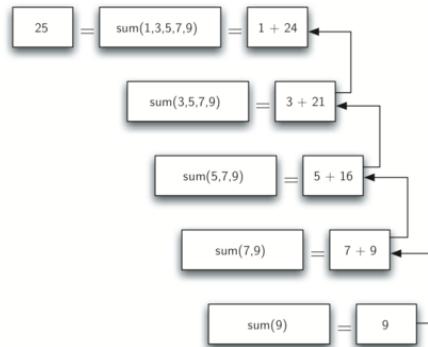
There are a few key ideas in this listing to look at. First, on line 2 we are checking to see if the list is one element long. This check is crucial and is our escape clause from the function. The sum of a list of length 1 is trivial; it is just the number in the list. Second, on line 5 our function calls itself! This is the reason that we call the `listsum` algorithm recursive. A recursive function is a function that calls itself.

[Figure 16.2.1](#) shows the series of **recursive calls** that are needed to sum the list [1, 3, 5, 7, 9]. You should think of this series of calls as a series of simplifications. Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller.



**Figure 16.2.1** Figure 1: Series of Recursive Calls Adding a List of Numbers

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. [Figure 16.2.2](#) shows the additions that are performed as `listsum` works its way backward through the series of calls. When `listsum` returns from the topmost problem, we have the solution to the whole problem.



**Figure 16.2.2** Figure2: Series of Recursive Returns from Adding a List of Numbers

### 16.3 The Three Laws of Recursion

Like the robots of Asimov, all recursive algorithms must obey three important laws:

- 1 A recursive algorithm must have a **base case**.
- 2 A recursive algorithm must change its state and move toward the base case.
- 3 A recursive algorithm must call itself, recursively.

Let's look at each one of these laws in more detail and see how it was used in the `listsum` algorithm. First, a base case is the condition that allows the algorithm to stop recursing. A base case is typically a problem that is small enough to solve directly. In the `listsum` algorithm the base case is a list of length 1.

To obey the second law, we must arrange for a change of state that moves the algorithm toward the base case. A change of state means that some data that the algorithm is using is modified. Usually the data that represents our problem gets smaller in some way. In the `listsum` algorithm our primary data structure is a list, so we must focus our state-changing efforts on the list. Since the base case is a list of length 1, a natural progression toward the base case is to shorten the list. This is exactly what happens on line 5 of the ActiveCode in [Section 16.2](#) when we call `listsum` with a shorter list.

The final law is that the algorithm must call itself. This is the very definition of recursion. Recursion is a confusing concept to many beginning programmers. As a novice programmer, you have learned that functions are good because you can take a large problem and break it up into smaller problems. The smaller problems can be solved by writing a function to solve each problem. When we talk about recursion it may seem that we are talking ourselves in circles. We have a problem to solve with a function, but that function solves the problem by calling itself! But the logic is not circular at all; the logic of recursion is an elegant expression of solving a problem by breaking it down into a smaller and easier problems.

In the remainder of this chapter we will look at more examples of recursion. In each case we will focus on designing a solution to a problem by using the three laws of recursion.

**Checkpoint 16.3.1** How many recursive calls are made when computing the sum of the list [2,4,6,8,10]?

- A. 6
- B. 5
- C. 4
- D. 3

**Checkpoint 16.3.2** Suppose you are going to write a recursive function to calculate the factorial of a number. `fact(n)` returns  $n * n-1 * n-2 * \dots * 1$ , and the factorial of zero is defined to be 1. What would be the most appropriate base case?

- A.  $n == 0$
- B.  $n == 1$
- C.  $n >= 0$
- D.  $n <= 1$

## 16.4 Converting an Integer to a String in Any Base

Suppose you want to convert an integer to a string in some base between binary and hexadecimal. For example, convert the integer 10 to its string representation in decimal as "10", or to its string representation in binary as "1010". While there are many approaches one can take to solve this problem, the recursive formulation of the problem is very elegant.

Let's look at a concrete example using base 10 and the number 769. Suppose we have a sequence of characters corresponding to the first 10 digits, like `convString = "0123456789"`. It is easy to convert a number less than 10 to its string equivalent by looking it up in the sequence. For example, if the number is 9, then the string is `convString[9]` or "9". If we can arrange to break up the number 769 into three single-digit numbers, 7, 6, and 9, then converting it to a string is simple. A number less than 10 sounds like a good base case.

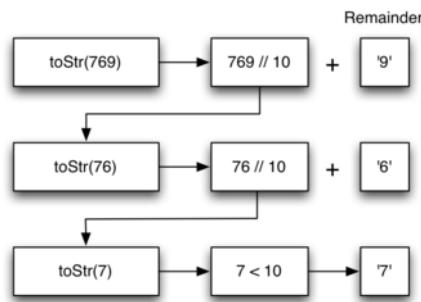
Knowing what our base is suggests that the overall algorithm will involve three components:

- 1 Reduce the original number to a series of single-digit numbers.
- 2 Convert the single digit-number to a string using a lookup.
- 3 Concatenate the single-digit strings together to form the final result.

The next step is to figure out how to change state and make progress toward the base case. Since we are working with an integer, let's consider what mathematical operations might reduce a number. The most likely candidates are division and subtraction. While subtraction might work, it is unclear what we should subtract from what. Integer division with remainders gives us a clear direction. Let's look at what happens if we divide a number by the base we are trying to convert to.

Using integer division to divide 769 by 10, we get 76 with a remainder of 9. This gives us two good results. First, the remainder is a number less than our

base that can be converted to a string immediately by lookup. Second, we get a number that is smaller than our original and moves us toward the base case of having a single number less than our base. Now our job is to convert 76 to its string representation. Again we will use integer division plus remainder to get results of 7 and 6 respectively. Finally, we have reduced the problem to converting 7, which we can do easily since it satisfies the base case condition of  $n < \text{base}$ , where  $\text{base} = 10$ . The series of operations we have just performed is illustrated in [Figure 16.4.1](#). Notice that the numbers we want to remember are in the remainder boxes along the right side of the diagram.



**Figure 16.4.1** Figure 3: Converting an Integer to a String in Base 10

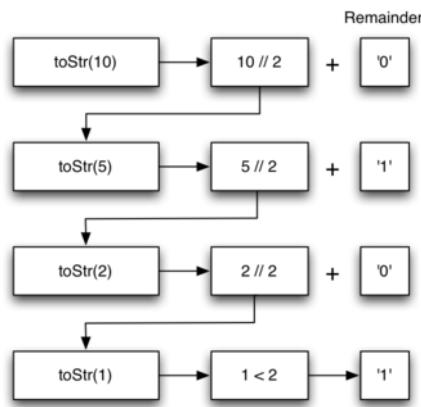
The activecode below shows the Python code that implements the algorithm outlined above for any base between 2 and 16.

```
def toStr(n,base):
    convertString = "0123456789ABCDEF"
    if n < base:
        return convertString[n]
    else:
        return toStr(n//base,base) + convertString[n%base]

print(toStr(1453,16))
```

Notice that in line 3 we check for the base case where  $n$  is less than the base we are converting to. When we detect the base case, we stop recursing and simply return the string from the `convertString` sequence. In line 6 we satisfy both the second and third laws—by making the recursive call and by reducing the problem size—using division.

Let's trace the algorithm again; this time we will convert the number 10 to its base 2 string representation ("1010").



**Figure 16.4.2** Figure 4: Converting the Number 10 to its Base 2 String Representation

Figure 16.4.2 shows that we get the results we are looking for, but it looks like the digits are in the wrong order. The algorithm works correctly because we make the recursive call first on line 6, then we add the string representation of the remainder. If we reversed returning the `convertString` lookup and returning the `toString` call, the resulting string would be backward! But by delaying the concatenation operation until after the recursive call has returned, we get the result in the proper order.

**Note 16.4.3 Self Check.** Write a function that takes a string as a parameter and returns a new string that is the reverse of the old string.

```

from test import testEqual
def reverse(s):
    return s

testEqual(reverse("hello"), "olleh")
testEqual(reverse("l"), "l")
testEqual(reverse("follow"), "wollof")
testEqual(reverse(""), "")
  
```

Write a function that takes a string as a parameter and returns True if the string is a palindrome, False otherwise. Remember that a string is a palindrome if it is spelled the same both forward and backward. for example: radar is a palindrome. for bonus points palindromes can also be phrases, but you need to remove the spaces and punctuation before checking. for example: madam i'm adam is a palindrome. Other fun palindromes include:

- kayak
- aibohphobia
- Live not on evil
- Reviled did I live, said I, as evil I did deliver
- Go hang a salami; I'm a lasagna hog.
- Able was I ere I saw Elba
- Kanakanak – a town in Alaska
- Wassamassaw – a town in South Dakota

```

from test import testEqual
def removeWhite(s):
    return s

def isPal(s):
    return False

testEqual(isPal(removeWhite("x")), True)
testEqual(isPal(removeWhite("radar")), True)
testEqual(isPal(removeWhite("hello")), False)
testEqual(isPal(removeWhite("")), True)
testEqual(isPal(removeWhite("hannah")), True)
testEqual(isPal(removeWhite("madam\u00b7ui'\u00b7adam")), True)

```

## 16.5 Visualizing Recursion

Some problems are easy to solve using recursion; however, it can still be difficult to find a mental model or a way of visualizing what is happening in a recursive function. This can make recursion difficult for people to grasp. In this section we will look at using recursion to draw some interesting pictures. As you watch these pictures take shape you will get some new insight into the recursive process that may be helpful in cementing your understanding of recursion.

For our next program we are going to draw a fractal tree. Fractals come from a branch of mathematics, and have much in common with recursion. The definition of a fractal is that when you look at it the fractal has the same basic shape no matter how much you magnify it. Some examples from nature are the coastlines of continents, snowflakes, mountains, and even trees or shrubs. The fractal nature of many of these natural phenomenon makes it possible for programmers to generate very realistic looking scenery for computer generated movies. In our next example we will generate a fractal tree.

To understand how this is going to work it is helpful to think of how we might describe a tree using a fractal vocabulary. Remember that we said above that a fractal is something that looks the same at all different levels of magnification. If we translate this to trees and shrubs we might say that even a small twig has the same shape and characteristics as a whole tree. Using this idea we could say that a *tree* is a trunk, with a smaller *tree* going off to the right and another smaller *tree* going off to the left. If you think of this definition recursively it means that we will apply the recursive definition of a tree to both of the smaller left and right trees.

Let's translate this idea to some Python code. Listing 1 below shows how we can use our turtle to generate a fractal tree. Let's look at the code a bit more closely. You will see that on lines 5 and 7 we are making a recursive call. On line 5 we make the recursive call right after the turtle turns to the right by 20 degrees; this is the right tree mentioned above. Then in line 7 the turtle makes another recursive call, but this time after turning left by 40 degrees. The reason the turtle must turn left by 40 degrees is that it needs to undo the original 20 degree turn to the right and then do an additional 20 degree turn to the left in order to draw the left tree. Also notice that each time we make a recursive call to `tree` we subtract some amount from the `branchLen` parameter; this is to make sure that the recursive trees get smaller and smaller. You should also recognize the initial `if` statement on line 2 as a check for the base case of `branchLen` getting too small.

**Listing 1.**

```

def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-10,t)
        t.right(20)
        t.backward(branchLen)

```

The complete program for this tree example is shown below. Before you run the code think about how you expect to see the tree take shape. Look at the recursive calls and think about how this tree will unfold. Will it be drawn symmetrically with the right and left halves of the tree taking shape simultaneously? Will it be drawn right side first then left side?

### Complete Tree Example Program.

```

import turtle

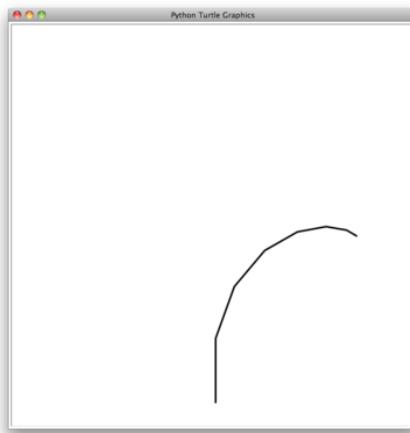
def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-15,t)
        t.right(20)
        t.backward(branchLen)

def main():
    t = turtle.Turtle()
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75,t)
    myWin.exitonclick()

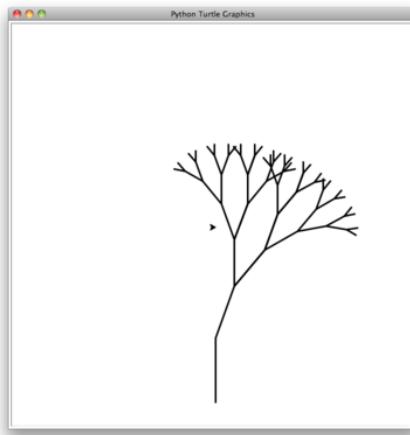
main()

```

Notice how each branch point on the tree corresponds to a recursive call, and notice how the tree is drawn to the right all the way down to its shortest twig. You can see this in [Figure 16.5.1](#). Now, notice how the program works its way back up the trunk until the entire right side of the tree is drawn. You can see the right half of the tree in [Figure 16.5.2](#). Then the left side of the tree is drawn, but not by going as far out to the left as possible. Rather, once again the entire right side of the left tree is drawn until we finally make our way out to the smallest twig on the left.



**Figure 16.5.1** Figure 1: The Beginning of a Fractal Tree



hello world

**Figure 16.5.2** Figure 2: The First Half of the Tree

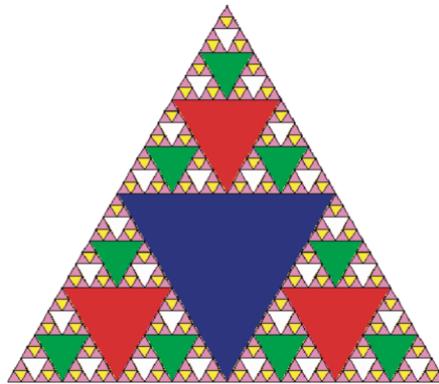
This simple tree program is just a starting point for you, and you will notice that the tree does not look particularly realistic because nature is just not as symmetric as a computer program. Here are a few ideas for how to explore some interesting options to make your tree look more realistic.

**Note 16.5.3 Self Check.** Modify the recursive tree program using one or all of the following ideas:

- Modify the thickness of the branches so that as the `branchLen` gets smaller, the line gets thinner.
- Modify the color of the branches so that as the `branchLen` gets very short it is colored like a leaf.
- Modify the angle used in turning the turtle so that at each branch point the angle is selected at random in some range. For example choose the angle between 15 and 45 degrees. Play around to see what looks good.
- Modify the `branchLen` recursively so that instead of always subtracting the same amount you subtract a random amount in some range.

## 16.6 Sierpinski Triangle

Another fractal that exhibits the property of self-similarity is the Sierpinski triangle. An example is shown in Figure 16.6.1. The Sierpinski triangle illustrates a three-way recursive algorithm. The procedure for drawing a Sierpinski triangle by hand is simple. Start with a single large triangle. Divide this large triangle into four new triangles by connecting the midpoint of each side. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles. You can continue to apply this procedure indefinitely if you have a sharp enough pencil. Before you continue reading, you may want to try drawing the Sierpinski triangle yourself, using the method described.



**Figure 16.6.1** Figure 3: The Sierpinski Triangle

Since we can continue to apply the algorithm indefinitely, what is the base case? We will see that the base case is set arbitrarily as the number of times we want to divide the triangle into pieces. Sometimes we call this number the “degree” of the fractal. Each time we make a recursive call, we subtract 1 from the degree until we reach 0. When we reach a degree of 0, we stop making recursive calls. The code that generated this Sierpinski Triangle is shown below.

```

import turtle

def drawTriangle(points,color,myTurtle):
    myTurtle.fillcolor(color)
    myTurtle.up()
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.down()
    myTurtle.begin_fill()
    myTurtle.goto(points[1][0],points[1][1])
    myTurtle.goto(points[2][0],points[2][1])
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.end_fill()

def getMid(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

def sierpinsky(points,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow',
                'violet','orange']

```

```

drawTriangle(points, colormap[degree], myTurtle)
if degree > 0:
    sierpinski([points[0],
                getMid(points[0], points[1]),
                getMid(points[0], points[2])],
               degree-1, myTurtle)
    sierpinski([points[1],
                getMid(points[0], points[1]),
                getMid(points[1], points[2])],
               degree-1, myTurtle)
    sierpinski([points[2],
                getMid(points[2], points[1]),
                getMid(points[0], points[2])],
               degree-1, myTurtle)

def main():
    myTurtle = turtle.Turtle()
    myWin = turtle.Screen()
    myPoints = [[-100,-50],[0,100],[100,-50]]
    sierpinski(myPoints,3,myTurtle)
    myWin.exitonclick()

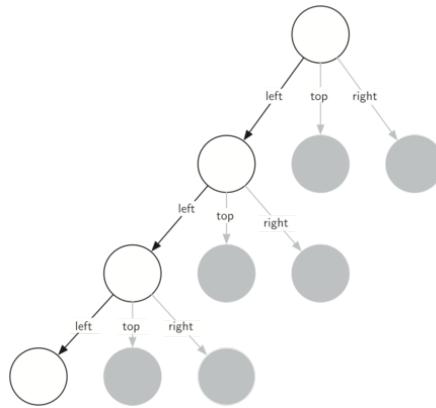
main()

```

This program follows the ideas outlined above. The first thing `sierpinski` does is draw the outer triangle. Next, there are three recursive calls, one for each of the new corner triangles we get when we connect the midpoints.

Look at the code and think about the order in which the triangles will be drawn. While the exact order of the corners depends upon how the initial set is specified, let's assume that the corners are ordered lower left, top, lower right. Because of the way the `sierpinski` function calls itself, `sierpinski` works its way to the smallest allowed triangle in the lower-left corner, and then begins to fill out the rest of the triangles working back. Then it fills in the triangles in the top corner by working toward the smallest, topmost triangle. Finally, it fills in the lower-right corner, working its way toward the smallest triangle in the lower right.

Sometimes it is helpful to think of a recursive algorithm in terms of a diagram of function calls. [Figure 16.6.2](#) shows that the recursive calls are always made going to the left. The active functions are outlined in black, and the inactive function calls are in gray. The farther you go toward the bottom of [Figure 16.6.2](#), the smaller the triangles. The function finishes drawing one level at a time; once it is finished with the bottom left it moves to the bottom middle, and so on.



**Figure 16.6.2** Figure 4: Building a Sierpinski Triangle

The `sierpinski` function relies heavily on the `getMid` function. `getMid` takes as arguments two endpoints and returns the point halfway between them. In addition, this program has a function that draws a filled triangle using the `begin_fill` and `end_fill` turtle methods.

## 16.7 Glossary

### Glossary

**base case.** A branch of the conditional statement in a recursive function that does not give rise to further recursive calls.

**data structure.** An organization of data for the purpose of making it easier to use.

**immutable data type.** A data type which cannot be modified. Assignments to elements or slices of immutable types cause a runtime error.

**infinite recursion.** A function that calls itself recursively without ever reaching the base case. Eventually, an infinite recursion causes a runtime error.

**mutable data type.** A data type which can be modified. All mutable types are compound types. Lists and dictionaries (see next chapter) are mutable data types; strings and tuples are not.

**recursion.** The process of calling the function that is already executing.

**recursive call.** The statement that calls an already executing function. Recursion can even be indirect — function f can call g which calls h, and h could make a call back to f.

**recursive definition.** A definition which defines something in terms of itself. To be useful it must include *base cases* which are not recursive. In this way it differs from a *circular definition*. Recursive definitions often provide an elegant way to express complex data structures.

**tuple.** A data type that contains a sequence of elements of any type, like a list, but is immutable. Tuples can be used wherever an immutable type is required, such as a key in a dictionary (see next chapter).

**tuple assignment.** An assignment to all of the elements in a tuple using a single assignment statement. Tuple assignment occurs in parallel rather than in sequence, making it useful for swapping values.

## 16.8 Programming Exercises

**Checkpoint 16.8.1** Write a recursive function to compute the factorial of a number.

```
def computeFactorial(number):
    #your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(computeFactorial(0),1,"Tested\u
            computeFactorial\uon\uinput\u0")
        self.assertEqual(computeFactorial(1),1,"Tested\u
            computeFactorail\uon\uinput\u1")
        self.assertEqual(computeFactorial(2),2,"Tested\u
            computeFactorial\uon\uinput\u2")
        self.assertEqual(computeFactorial(3),6,"Tested\u
            computeFactorial\uon\uinput\u3")
        self.assertEqual(computeFactorial(4),24,"Tested\u
            computeFactorial\uon\uinput\u4")
        self.assertEqual(computeFactorial(8),40320,"Tested\u
            computeFactorial\uon\uinput\u8")
        self.assertEqual(computeFactorial(-5),None,"Tested\u
            computeFactorial\uon\uanegative\unumber\u-make\u
            sure\uuto\uhandle\uthis\ucase")

myTests().main()
```

**Checkpoint 16.8.2** Write a recursive function to reverse a list.

```
def reverseList(lst):
    #your code here

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(reverseList([1,2,3,4,5]),
            [5,4,3,2,1], "Your\ufunction\ufailed\uwith\uinput\u
            [1,2,3,4,5]")
        self.assertEqual(reverseList(['Hello','World','!']),
            ['!', 'World', 'Hello'], "Your\ufunction\ufailed\u
            with\uinput\u['Hello', 'World', '!']")
        self.assertEqual(reverseList(['Python',100,'35','Computer\u
            Science']), ['Computer\uScience', '35', 100,
            'Python'], "Your\ufunction\ufailed\uwith\uinput\u
            ['Python',100,'35','Computer\uScience'])"

myTests().main()
```

**Checkpoint 16.8.3** Modify the recursive tree program using one or all of the following ideas:

- Modify the thickness of the branches so that as the `branchLen` gets smaller, the line gets thinner.
- Modify the color of the branches so that as the `branchLen` gets very short it is colored like a leaf.
- Modify the angle used in turning the turtle so that at each branch point the angle is selected at random in some range. For example choose the angle between 15 and 45 degrees. Play around to see what looks good.
- Modify the `branchLen` recursively so that instead of always subtracting the same amount you subtract a random amount in some range.

If you implement all of the above ideas you will have a very realistic looking tree.

**Checkpoint 16.8.4** Find or invent an algorithm for drawing a fractal mountain. Hint: One approach to this uses triangles again.

**Checkpoint 16.8.5** Write a recursive function to compute the Fibonacci sequence. How does the performance of the recursive function compare to that of an iterative version?

**Checkpoint 16.8.6** Implement a solution to the Tower of Hanoi using three stacks to keep track of the disks.

**Checkpoint 16.8.7** Using the turtle graphics module, write a recursive program to display a Hilbert curve.

**Checkpoint 16.8.8** Using the turtle graphics module, write a recursive program to display a Koch snowflake.

**Checkpoint 16.8.9** Write a program to solve the following problem: You have two jugs: a 4-gallon jug and a 3-gallon jug. Neither of the jugs have markings on them. There is a pump that can be used to fill the jugs with water. How can you get exactly two gallons of water in the 4-gallon jug?

**Checkpoint 16.8.10** Generalize the problem above so that the parameters to your solution include the sizes of each jug and the final amount of water to be left in the larger jug.

**Checkpoint 16.8.11** Write a program that solves the following problem: Three missionaries and three cannibals come to a river and find a boat that holds two people. Everyone must get across the river to continue on the journey. However, if the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. Find a series of crossings that will get everyone safely to the other side of the river.

**Checkpoint 16.8.12** Modify the Tower of Hanoi program using turtle graphics to animate the movement of the disks. Hint: You can make multiple turtles and have them shaped like rectangles.

**Checkpoint 16.8.13** Pascal's triangle is a number triangle with numbers arranged in staggered rows such that  $a_{nr} = \frac{n!}{r!(n-r)!}$ . This equation is the equation for a binomial coefficient. You can build Pascal's triangle by adding the two numbers that are diagonally above a number in the triangle. An example of Pascal's triangle is shown below.

	1	1		
1	2	1		
1	3	3	1	
1	4	6	4	1

Write a program that prints out Pascal's triangle. Your program should accept a parameter that tells how many rows of the triangle to print.

## 16.9 Exercises

Adding this blank exercises page to hold instructor written questions.

# Chapter 17

## Classes and Objects - the Basics

### 17.1 Object-oriented programming

Python is an **object-oriented programming language**. That means it provides features that support **object-oriented programming**<sup>1</sup> (**OOP**).

Object-oriented programming has its roots in the 1960s, but it wasn't until the mid 1980s that it became the main **programming paradigm**<sup>2</sup> used in the creation of new software. It was developed as a way to handle the rapidly increasing size and complexity of software systems and to make it easier to modify these large and complex systems over time.

Up to now, some of the programs we have been writing use a **procedural programming**<sup>3</sup> paradigm. In procedural programming the focus is on writing functions or *procedures* which operate on data. In object-oriented programming the focus is on the creation of **objects** which contain both data and functionality together. Usually, 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.

### 17.2 A change of perspective

Throughout the earlier chapters, we wrote functions and called them using a syntax such as `drawCircle(tess)`. This suggests that the function is the active agent. It says something like, “*Hey, drawCircle! Here’s a turtle object for you to use to draw with.*”

In object-oriented programming, the objects are considered the active agents. For example, in our early introduction to turtles, we used an object-oriented style. We said `tess.forward(100)`, which asks the turtle to move itself forward by the given number of steps. An invocation like `tess.circle()` says “*Hey tess! Please use your circle method!*”

This change in perspective is sometimes considered to be a more “polite” way to write programming instructions. However, it may not initially be obvious that it is useful. It turns out that often times shifting responsibility

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Object-oriented\\_programming](http://en.wikipedia.org/wiki/Object-oriented_programming)

<sup>2</sup>[http://en.wikipedia.org/wiki/Programming\\_paradigm](http://en.wikipedia.org/wiki/Programming_paradigm)

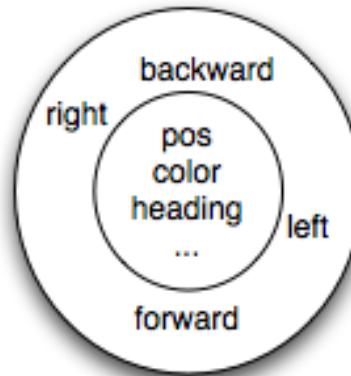
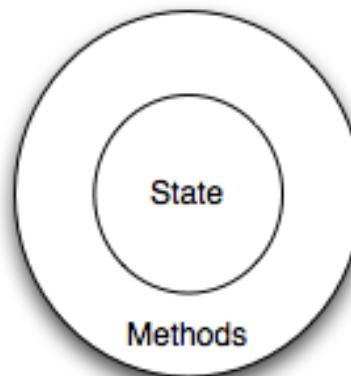
<sup>3</sup>[http://en.wikipedia.org/wiki/Procedural\\_programming](http://en.wikipedia.org/wiki/Procedural_programming)

from the functions onto the objects makes it possible to write more versatile functions and makes it easier to maintain and reuse code.

The most important advantage of the object-oriented style is that it fits our mental chunking and real-life experience more accurately. In real life our `cook` method is part of our microwave oven — we don't have a `cook` function sitting in the corner of the kitchen, into which we pass the microwave! Similarly, we use the cellphone's own methods to send an sms, or to change its state to silent. The functionality of real-world objects tends to be tightly bound up inside the objects themselves. OOP allows us to accurately mirror this when we organize our programs.

### 17.3 Objects Revisited

In Python, every value is actually an object. Whether it be a turtle, a list, or even an integer, they are all objects. Programs manipulate those objects either by performing computation with them or by asking them to perform methods. To be more specific, we say that an object has a **state** and a collection of **methods** that it can perform. The state of an object represents those things that the object knows about itself. For example, as we have seen with turtle objects, each turtle has a state consisting of the turtle's position, its color, its heading and so on. Each turtle also has the ability to go forward, backward, or turn right or left. Individual turtles are different in that even though they are all turtles, they differ in the specific values of the individual state attributes (maybe they are in a different location or have a different heading).

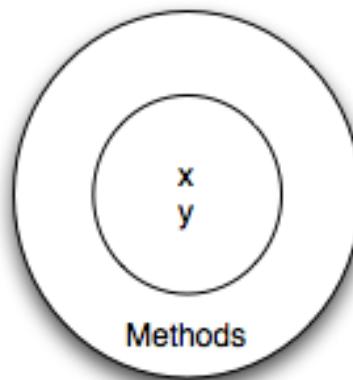


## 17.4 User Defined Classes

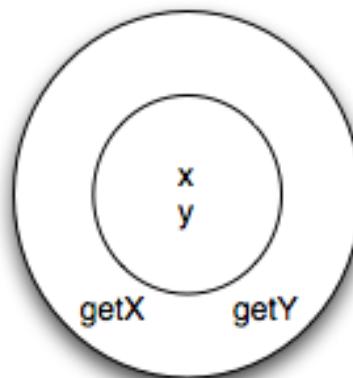
We've already seen classes like `str`, `int`, `float` and `Turtle`. These were defined by Python and made available for us to use. However, in many cases when we are solving problems we need to create data objects that are related to the problem we are trying to solve. We need to create our own classes.

As an example, consider the concept of a mathematical point. In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. 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. This  $(x, y)$  is the state of the point.

Thinking about our diagram above, we could draw a `point` object as shown here.



Some of the typical operations that one associates with points might be to ask the point for its `x` coordinate, `getX`, or to ask for its `y` coordinate, `getY`. You may also wish to calculate the distance of a point from the origin, or the distance of a point from another point, or find the midpoint between two points, or answer the question as to whether a point falls within a given rectangle or circle. We'll shortly see how we can organize these together with the data.



Now that we understand what a `point` object might look like, we can define a new **class**. We'll want our points to each have an `x` and a `y` attribute, so our

first class definition looks like this.

```
class Point:
    """Point class for representing and manipulating x,y coordinates."""

    def __init__(self):
        """Create a new point at the origin"""
        self.x = 0
        self.y = 0
```

Class definitions can appear anywhere in a program, but they are usually near the beginning (after the `import` statements). The syntax rules for a class definition are the same as for other compound statements. There is a header which begins with the keyword, `class`, followed by the name of the class, and ending with a colon.

If the first line after the class header is a string, it becomes the docstring of the class, and will be recognized by various tools. (This is also the way docstrings work in functions.)

Every class should have a method with the special name `__init__`. This **initializer method**, often referred to as the **constructor**, is automatically called whenever a new instance of `Point` is created. It gives the programmer the opportunity to set up the attributes required within the new instance by giving them their initial state values. The `self` parameter (you could choose any other name, but nobody ever does!) is automatically set to reference the newly-created object that needs to be initialized.

So let's use our new `Point` class now.

```
class Point:
    """Point class for representing and manipulating x,y coordinates."""

    def __init__(self):
        """Create a new point at the origin"""
        self.x = 0
        self.y = 0

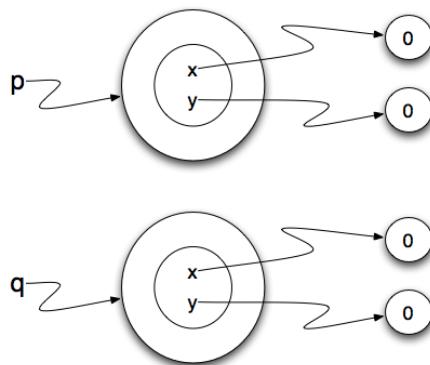
    p = Point()          # Instantiate an object of type Point
    q = Point()          # and make a second point

    print("Nothing seems to have happened with the points")
```

During the initialization of the objects, we created two attributes called `x` and `y` for each, and gave them both the value 0.

**Note 17.4.1** The assignments are not to `x` and `y`, but to `self.x` and `self.y`. The attributes `x` and `y` are *always* attached to a particular instance. The instance is always explicitly referenced with dot notation.

You will note that when you run the program, nothing happens. It turns out that this is not quite the case. In fact, two `Points` have been created, each having an `x` and `y` coordinate with value 0. However, because we have not asked the point to do anything, we don't see any other result.



You can see this for yourself, via codelens:

```
class Point:
    """A class for representing and manipulating x,y coordinates."""

    def __init__(self):
        """Create a new point at the origin"""
        self.x = 0
        self.y = 0

p = Point()      # Instantiate an object of type Point
q = Point()      # and make a second point

print("Nothing seems to have happened with the points")
```

The following program adds a few print statements. You can see that the output suggests that each one is a `Point` object. However, notice that the `is` operator returns `False` meaning that they are different objects (we will have more to say about this in a later chapter).

```
class Point:
    """A class for representing and manipulating x,y coordinates."""

    def __init__(self):
        """Create a new point at the origin"""
        self.x = 0
        self.y = 0

p = Point()      # Instantiate an object of type Point
q = Point()      # and make a second point

print(p)
print(q)

print(p is q)
```

This should look familiar — we've used classes before to create more than one object:

```
from turtle import Turtle

tess = Turtle()      # Instantiate objects of type Turtle
alex = Turtle()
```

The variables `p` and `q` are assigned references to two new `Point` objects.

A function like `Turtle` or `Point` that creates a new object instance is called a **constructor**. Every class automatically uses the name of the class as the name of the constructor function. The definition of the constructor function is done when you write the `__init__` function.

It may be helpful to think of a class as a factory for making objects. The class itself isn't an instance of a point, but it contains the machinery to make point instances. Every time you call the constructor, you're asking the factory to make you a new object. As the object comes off the production line, its initialization method is executed to get the object properly set up with its factory default settings.

The combined process of "make me a new object" and "get its settings initialized to the factory default settings" is called **instantiation**.

### Check Your Understanding

**Checkpoint 17.4.2** What is the output of the following print code?

```
class Car:

    def __init__(self):
        self.color = "Red"
        self.size = "Big"
BMW = Car()
Tesla = Car()

x = BMW is Tesla
y = type(BMW)==type(Tesla)

print(x, y)
```

- A. True True
- B. True False
- C. False True
- D. False False

## 17.5 Improving our Constructor

Our constructor so far can only create points at location  $(0,0)$ . To create a point at position  $(7, 6)$  requires that we provide some additional capability for the user to pass information to the constructor. Since constructors are simply specially named functions, we can use parameters (as we've seen before) to provide the specific information.

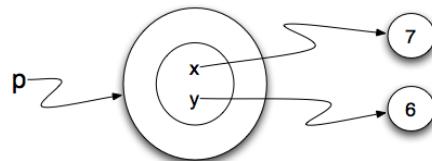
We can make our class constructor more general by putting extra parameters into the `__init__` method, as shown in this codelens example.

```
class Point:
    """Point class for representing and manipulating x,y coordinates"""

    def __init__(self, initX, initY):
        """Create a new point at the given coordinates."""
        self.x = initX
        self.y = initY

p = Point(7, 6)
```

Now when we create new points, we supply the x and y coordinates as parameters. When the point is created, the values of `initX` and `initY` are assigned to the state of the object.



## 17.6 Adding Other Methods to our Class

The key advantage of using a class like `Point` rather than something like a simple tuple `(7, 6)` now becomes apparent. We can add methods to the `Point` class that are sensible operations for points. Had we chosen to use a simple tuple to represent the point, we would not have this capability. Creating a class like `Point` brings an exceptional amount of “organizational power” to our programs, and to our thinking. We can group together the sensible operations, and the kinds of data they apply to, and each instance of the class can have its own state.

A **method** behaves like a function but it is invoked on a specific instance. For example, with a turtle named `tess`, `tess.right(90)` asks the `tess` object to perform its `right` method and turn 90 degrees. Methods are accessed using dot notation.

Let's add two simple methods to allow a point to give us information about its state. The `getX` method, when invoked, will return the value of the x coordinate. The implementation of this method is straight forward since we already know how to write functions that return values. One thing to notice is that even though the `getX` method does not need any other parameter information to do its work, there is still one formal parameter, `self`. As we stated earlier, all methods defined in a class that operate on objects of that class will have `self` as their first parameter. Again, this serves as reference to the object itself which in turn gives access to the state data inside the object.

```
class Point:
    """A Point class for representing and manipulating x,y coordinates."""
    def __init__(self, initX, initY):
        """Create a new point at the given coordinates."""
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

p = Point(7, 6)
print(p.getX())
print(p.getY())
```

Note that the `getX` method simply returns the value of `self.x` from the

object itself. In other words, the implementation of the method is to go to the state of the object itself and get the value of `x`. Likewise, the `getY` method looks the same.

Let's add another method, `distanceFromOrigin`, to see better how methods work. This method will again not need any additional information to do its work. It will perform a more complex task.

```
class Point:
    """Point class for representing and manipulating x,y coordinates"""

    def __init__(self, initX, initY):
        """Create a new point at the given coordinates."""
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

p = Point(7, 6)
print(p.distanceFromOrigin())
```

Notice that the caller of `distanceFromOrigin` does not explicitly supply an argument to match the `self` parameter. This is true of all method calls. The definition will always have one additional parameter as compared to the invocation.

## 17.7 Objects as Arguments and Parameters

You can pass an object as an argument in the usual way. We've already seen this in some of the turtle examples where we passed the turtle to some function like `drawRectangle` so that the function could control and use whatever turtle instance we passed to it.

Here is a simple function called `distance` involving our new `Point` objects. The job of this function is to figure out the distance between two points.

```
import math

class Point:
    """Point class for representing and manipulating x,y coordinates"""

    def __init__(self, initX, initY):
        """Create a new point at the given coordinates."""
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
```

```

        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def distance(point1, point2):
        xdiff = point2.getX() - point1.getX()
        ydiff = point2.getY() - point1.getY()

        dist = math.sqrt(xdiff**2 + ydiff**2)
        return dist

p = Point(4, 3)
q = Point(0, 0)
print(distance(p, q))

```

`distance` takes two points and returns the distance between them. Note that `distance` is **not** a method of the `Point` class. You can see this by looking at the indentation pattern. It is not inside the class definition. The other way we can know that `distance` is not a method of `Point` is that `self` is not included as a formal parameter. In addition, we do not invoke `distance` using the dot notation.

## 17.8 Converting an Object to a String

When we're working with classes and objects, it is often necessary to print an object (that is to print the state of an object). Consider the example below.

```

class Point:
    """Point class for representing and manipulating x,y coordinates"""

    def __init__(self, initX, initY):
        """Create a new point at the given coordinates."""
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

p = Point(7, 6)
print(p)

```

The `print` function shown above produces a string representation of the `Point` `p`. The default functionality provided by Python tells you that `p` is an object of type `Point`. However, it does not tell you anything about the specific state of the point.

We can improve on this representation if we include a special method call `__str__`. Notice that this method uses the same naming convention as the constructor, that is two underscores before and after the name. It is common that Python uses this naming technique for special methods.

The `__str__` method is responsible for returning a string representation as defined by the class creator. In other words, you as the programmer, get to choose what a `Point` should look like when it gets printed. In this case, we have decided that the string representation will include the values of `x` and `y` as well as some identifying text. It is required that the `__str__` method create and *return* a string.

```
class Point:
    """A Point class for representing and manipulating x,y coordinates"""

    def __init__(self, initX, initY):
        """Create a new point at the given coordinates."""
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def __str__(self):
        return "x=" + str(self.x) + ",y=" + str(self.y)

p = Point(7, 6)
print(p)
```

When we run the program above you can see that the `print` function now shows the string that we chose.

Now, you ask, don't we already have an `str` type converter that can turn our object into a string? Yes we do!

And doesn't `print` automatically use this when printing things? Yes again!

But, as we saw earlier, these automatic mechanisms do not do exactly what we want. Python provides many default implementations for methods that we as programmers will probably want to change. When a programmer changes the meaning of a special method we say that we **override** the method. Note also that the `str` type converter function uses whatever `__str__` method we provide.

## 17.9 Instances as Return Values

Functions and methods can return objects. This is actually nothing new since everything in Python is an object and we have been returning values for quite some time. The difference here is that we want to have the method create an object using the constructor and then return it as the value of the method.

Suppose you have a point object and wish to find the midpoint halfway between it and some other target point. We would like to write a method, call it `halfway` that takes another `Point` as a parameter and returns the `Point` that is halfway between the point and the target.

```
class Point:

    def __init__(self, initX, initY):
```

```

    """_Create_a_new_point_at_the_given_coordinates._"""
    self.x = initX
    self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def __str__(self):
        return "x=" + str(self.x) + ",y=" + str(self.y)

    def halfway(self, target):
        mx = (self.x + target.x) / 2
        my = (self.y + target.y) / 2
        return Point(mx, my)

p = Point(3, 4)
q = Point(5, 12)
mid = p.halfway(q)

print(mid)
print(mid.getX())
print(mid.getY())

```

The resulting Point, `mid`, has an x value of 4 and a y value of 8. We can also use any other methods since `mid` is a Point object.

In the definition of the method `halfway` see how the requirement to always use dot notation with attributes disambiguates the meaning of the attributes `x` and `y`: We can always see whether the coordinates of Point `self` or `target` are being referred to.

**Note 17.9.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 17.10 Glossary

### Glossary

**attribute.** One of the named data items that makes up an instance.

**class.** A class can be thought of as a template for the objects that are instances of it. It defines a data type. A class can be provided by the Python system or be user-defined.

**constructor.** Every class has a “factory”, called by the same name as the class, for making new instances. If the class has an *initializer method*, this method is used to get the attributes (i.e. the state) of the new object properly set up.

**initializer method.** A special method in Python (called `__init__`) that is invoked automatically to set a newly-created object’s attributes to their initial (factory-default) state.

**instance.** An object whose type is of some class. Instance and object are used interchangeably.

**instantiate.** To create an instance of a class, and to run its initializer.

**method.** A function that is defined inside a class definition and is invoked on instances of that class.

**object.** A compound form of data that is often used to model a thing or concept in the real world. It bundles together the data and the operations that are relevant for that thing or concept. It has the type of its defining class. Instance and object are used interchangeably.

**object-oriented programming.** A powerful style of programming in which data and the operations that manipulate it are organized into classes and methods.

**object-oriented language.** A language that provides features, such as user-defined classes and inheritance, that facilitate object-oriented programming.

## 17.11 Exercises

1. Add a `distanceFromPoint` method that works similar to `distanceFromOrigin` except that it takes a `Point` as a parameter and computes the distance between that point and `self`.
2. Add a method `reflect_x` to `Point` which returns a new `Point`, one which is the reflection of the point about the x-axis. For example, `Point(3, 5).reflect_x()` is `(3, -5)`
3. Add a method `slope_from_origin` which returns the slope of the line joining the origin to the point. For example,

```
>>> Point(4, 10).slope_from_origin()  
2.5
```

What cases will cause your method to fail? Return `None` when it happens.

4. The equation of a straight line is “ $y = ax + b$ ”, (or perhaps “ $y = mx + c$ ”). The coefficients  $a$  and  $b$  completely describe the line. Write a method in the `Point` class so that if a point instance is given another point, it will compute the equation of the straight line joining the two points. It must return the two coefficients as a tuple of two values. For example,

```
>>> print(Point(4, 11).get_line_to(Point(6, 15)))  
>>> (2, 3)
```

This tells us that the equation of the line joining the two points is “ $y = 2x + 3$ ”. When will your method fail?

5. Add a method called `move` that will take two parameters, call them `dx` and `dy`. The method will cause the point to move in the x and y direction the number of units given. (Hint: you will change the values of the state of the point)
6. Given three points that fall on the circumference of a circle, find the center and radius of the circle.

# Chapter 18

## Classes and Objects - Digging a Little Deeper

### 18.1 Fractions

We have all had to work with fractions when we were younger. Or, perhaps you do a lot of cooking and have to manage measurements for ingredients. In any case, fractions are something that we are familiar with. In this chapter we will develop a class to represent a fraction including some of the most common methods that we would like fractions to be able to do.

A fraction is most commonly thought of as two integers, one over the other, with a line separating them. The number on the top is called the numerator and the number on the bottom is called the denominator. Sometimes people use a slash for the line and sometimes they use a straight line. The fact is that it really does not matter so long as you know which is the numerator and which is the denominator.

To design our class, we simply need to use the analysis above to realize that the state of a fraction object can be completely described by representing two integers. We can begin by implementing the `Fraction` class and the `__init__` method which will allow the user to provide a numerator and a denominator for the fraction being created.

```
class Fraction:

    def __init__(self, top, bottom):
        self.num = top          # the numerator is on top
        self.den = bottom        # the denominator is on the
                                bottom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

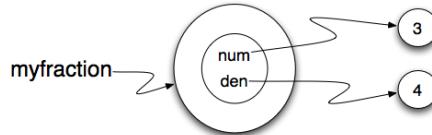
    def getNum(self):
        return self.num

    def getDen(self):
        return self.den

myfraction = Fraction(3, 4)
print(myfraction)
```

```
print(myfraction.getNum())
print(myfraction.getDen())
```

Note that the `__str__` method provides a “typical” looking fraction using a slash between the numerator and denominator. The figure below shows the state of `myfraction`. We have also added a few simple accessor methods, `getNum` and `getDen`, that can return the state values for the fraction.



## 18.2 Objects are Mutable

We can change the state of an object by making an assignment to one of its instance variables. For example, we could change the numerator of the fraction by assigning a new value to `self.num`. Likewise, we could do the same thing for `self.den`.

One place where this type of modification makes sense is when we place a fraction in **lowest terms**. Lowest terms simply means that the numerator and denominator do not share any common factors. For example,  $12/16$  is a fraction but it is not in lowest terms since 2 can divide into both 12 and 16. We call 2 a **common divisor**. If we divide the numerator and the denominator by a common divisor, we get an equivalent fraction. If we divide by the **greatest common divisor**, we will get the lowest terms representation. In this case 4 would be the greatest common divisor and the lowest terms representation would be  $3/4$ .

There is a very nice iterative method for computing the greatest common divisor of two integers. Try to run the function on a number of different examples.

```
def gcd(m, n):
    while m % n != 0:
        oldm = m
        oldn = n

        m = oldn
        n = oldm % oldn

    return n

print(gcd(12, 16))
```

Now that we have a function that can help us with finding the greatest common divisor, we can use that to implement a fraction method called `simplify`. We will ask the fraction “to put itself in lowest terms”.

The `simplify` method will pass the numerator and the denominator to the `gcd` function to find the greatest common divisor. It will then modify itself by dividing its `num` and its `den` by that value.

```
def gcd(m, n):
    while m % n != 0:
        oldm = m
        oldn = n
```

```

    m = oldn
    n = oldm % oldn

    return n

class Fraction:

    def __init__(self, top, bottom):

        self.num = top          # the numerator is on top
        self.den = bottom        # the denominator is on the
                                bottom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def simplify(self):
        common = gcd(self.num, self.den)

        self.num = self.num // common
        self.den = self.den // common

myfraction = Fraction(12, 16)

print(myfraction)
myfraction.simplify()
print(myfraction)

```

There are two important things to note about this implementation. First, the `gcd` function is not a method of the class. It does not belong to `Fraction`. Instead it is a function that is used by `Fraction` to assist in a task that needs to be performed. This type of function is often called a **helper function**. Second, the `simplify` method does not return anything. Its job is to modify the object itself. This type of method is known as a **mutator** method because it mutates or changes the internal state of the object.

### 18.3 Sameness

The meaning of the word *same* seems perfectly clear until you give it some thought and then you realize there is more to it than you expected.

For example, if you say, Chris and I have the same car, you mean that his car and yours are the same make and model, but that they are two different cars. If you say, Chris and I have the same mother, you mean that his mother and yours are the same person.

When you talk about objects, there is a similar ambiguity. For example, if two `Fractions` are the same, does that mean they represent the same rational number or that they are actually the same object?

We've already seen the `is` operator in the chapter on lists, where we talked about aliases. It allows us to find out if two references refer to the same object.

```

class Fraction:

    def __init__(self, top, bottom):

        self.num = top          # the numerator is on top
        self.den = bottom        # the denominator is on the

```

```

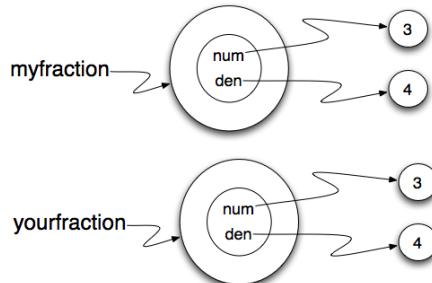
    bottom

def __str__(self):
    return str(self.num) + "/" + str(self.den)

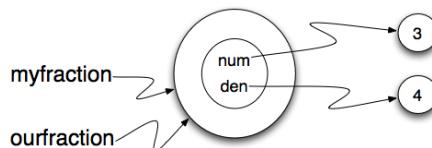
myfraction = Fraction(3, 4)
yourfraction = Fraction(3, 4)
print(myfraction is yourfraction)

ourfraction = myfraction
print(myfraction is ourfraction)
```

Even though `myfraction` and `yourfraction` refer to the same rational number, they are not the same object.



If we assign `myfraction` to `ourfraction`, then the two variables are aliases of the same object.



This type of equality is called **shallow equality** because it compares only the references, not the contents of the objects. Using the `==` operator to check equality between two user defined objects will return the shallow equality result. In other words, the `Fraction` objects are equal (`==`) if they are the same object.

Of course, we could define equality to mean the fractions are the same in that they represent the same rational number. Recall from algebra that  $a/b = c/d$  is equivalent to  $a*d = b*c$ . Here is a boolean function that performs this check.

```

def sameRational(f1, f2):
    return f1.getNum()*f2.getDen() == f2.getNum() *
        f1.getDen()
```

This type of equality is known as **deep equality** since it compares the values “deep” in the object, not just the reference to the object.

```

def sameRational(f1, f2):
    return f1.getNum()*f2.getDen() == f2.getNum() *
        f1.getDen()

class Fraction:
```

```

        self.num = top          # the numerator is on top
        self.den = bottom       # the denominator is on the
                                bottom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def getNum(self):
        return self.num

    def getDen(self):
        return self.den

myfraction = Fraction(3, 4)
yourfraction = Fraction(3, 4)
print(myfraction is yourfraction)
print(sameRational(myfraction, yourfraction))
notInLowestTerms = Fraction(15, 20)
print(sameRational(myfraction, notInLowestTerms))

```

Of course, if the two variables refer to the same object, they have both shallow and deep equality.

**Note 18.3.1 Beware of ==.** “When I use a word,” Humpty Dumpty said, in a rather scornful tone, “it means just what I choose it to mean — neither more nor less.” *Alice in Wonderland*

Python has a powerful feature that allows a designer of a class to decide what an operation like == or < should mean. (We’ve just shown how we can control how our own objects are converted to strings, so we’ve already made a start!) We’ll cover more detail later. But sometimes the implementors will attach shallow equality semantics, and sometimes deep equality, as shown in this little experiment:

```

p = Point(4, 2)
s = Point(4, 2)
print("== on Points returns", p == s) # by default, ==
                                         does a shallow equality test here

a = [2, 3]
b = [2, 3]
print("== on lists returns", a == b) # by default, ==
                                         does a deep equality test on lists

```

This outputs:

```

== on Points returns False
== on lists returns True

```

So we conclude that even though the two lists (or tuples, etc.) are distinct objects with different memory addresses, in one case the == operator tests for deep equality, while in the case of points it makes a shallow test.

## 18.4 Arithmetic Methods

We will conclude this chapter by adding a few more methods to our `Fraction` class. In particular, we will implement arithmetic. To begin, consider what it means to add two fractions together. Remember that you can only add frac-

tions if they have the same denominator. The easiest way to find a common denominator is to multiply the two individual denominators together. Anything we do to the denominator needs to be done to the numerator. This gives us the following equation for fraction addition:

$$\frac{a}{b} + \frac{c}{d} = \frac{(ad + cb)}{bd}$$

Our add method will take a `Fraction` as a parameter. It will return a new `Fraction` representing the sum. We will use the equation shown above to compute the new numerator and the new denominator. Since this equation will not give us lowest terms, we will utilize a similar technique as was used in the `simplify` method to find the greatest common divisor and then divide each part of the new fraction.

```
def add(self, otherfraction):

    newnum = self.num*otherfraction.den +
             self.den*otherfraction.num
    newden = self.den * otherfraction.den

    common = gcd(newnum, newden)

    return Fraction(newnum//common, newden//common)
```

You can try the addition method and then modify the fractions and retry.

```
def gcd(m, n):
    while m % n != 0:
        oldm = m
        oldn = n

        m = oldn
        n = oldm % oldn

    return n

class Fraction:

    def __init__(self, top, bottom):

        self.num = top          # the numerator is on top
        self.den = bottom        # the denominator is on the
                                # bottom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def simplify(self):
        common = gcd(self.num, self.den)

        self.num = self.num // common
        self.den = self.den // common

    def add(self, otherfraction):

        newnum = self.num*otherfraction.den +
                 self.den*otherfraction.num
        newden = self.den * otherfraction.den

        common = gcd(newnum, newden)
```

```

        return Fraction(newnum // common, newden // common)

f1 = Fraction(1, 2)
f2 = Fraction(1, 4)

f3 = f1.add(f2)
print(f3)

```

One final modification to this method will be quite useful. Instead invoking the `add` method, we can use the addition operator “`+`”. This requires that we implement another special method, this time called `__add__`. The details of the method are the same.

```

def __add__(self, otherfraction):

    newnum = self.num*otherfraction.den +
             self.den*otherfraction.num
    newden = self.den * otherfraction.den

    common = gcd(newnum, newden)

    return Fraction(newnum // common, newden // common)

```

However, now we can perform addition in the same manner that we are used to with other numeric data.

```

f1 = Fraction(1, 2)
f2 = Fraction(1, 4)

f3 = f1 + f2      # calls the __add__ method of f1
print(f3)

```

**Note 18.4.1** This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

## 18.5 Glossary

### Glossary

**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.

**deep equality.** Equality of values, or two references that point to objects that have the same value.

**shallow copy.** To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

**shallow equality.** Equality of references, or two references that point to the same object.

## 18.6 Exercises

1. We can represent a rectangle by knowing three things: the location of its lower left corner, its width, and its height. Create a class definition for a `Rectangle` class using this idea. To create a `Rectangle` object at location (4,5) with width 6 and height 5, we would do the following:

```
r = Rectangle(Point(4, 5), 6, 5)
```

2. Add the following accessor methods to the Rectangle class: `getWidth`, `getHeight`, `__str__`.
3. Add a method `area` to the Rectangle class that returns the area of any instance:

```
r = Rectangle(Point(0, 0), 10, 5)
test(r.area(), 50)
```

4. Write a `perimeter` method in the Rectangle class so that we can find the perimeter of any rectangle instance:

```
r = Rectangle(Point(0, 0), 10, 5)
test(r.perimeter(), 30)
```

5. Write a `transpose` method in the Rectangle class that swaps the width and the height of any rectangle instance:

```
r = Rectangle(Point(100, 50), 10, 5)
test(r.width, 10)
test(r.height, 5)
r.transpose()
test(r.width, 5)
test(r.height, 10)
```

6. Write a new method in the Rectangle class to test if a Point falls within the rectangle. For this exercise, assume that a rectangle at (0,0) with width 10 and height 5 has *open* upper bounds on the width and height, i.e. it stretches in the x direction from [0 to 10), where 0 is included but 10 is excluded, and from [0 to 5) in the y direction. So it does not contain the point (10, 2). These tests should pass:

```
r = Rectangle(Point(0, 0), 10, 5)
test(r.contains(Point(0, 0)), True)
test(r.contains(Point(3, 3)), True)
test(r.contains(Point(3, 7)), False)
test(r.contains(Point(3, 5)), False)
test(r.contains(Point(3, 4.99999)), True)
test(r.contains(Point(-3, -3)), False)
```

7. Write a new method called `diagonal` that will return the length of the diagonal that runs from the lower left corner to the opposite corner.

8. In games, we often put a rectangular “bounding box” around our sprites in the game. We can then do *collision detection* between, say, bombs and spaceships, by comparing whether their rectangles overlap anywhere.

Write a function to determine whether two rectangles collide. *Hint: this might be quite a tough exercise! Think carefully about all the cases before you code.*

# Chapter 19

## Inheritance

### 19.1 Pillars of OOP

Object-oriented programming involves four key ideas: encapsulation, information hiding, inheritance, and polymorphism. **Encapsulation** is the idea that a class can package some data together with the methods that manipulate the data. This is a powerful capability, and the chief idea that distinguishes OOP from structured programming. **Information Hiding** promotes quality code by allowing objects to protect their state against direct manipulation by code using the object. Python, like many languages, provides mechanisms to achieve information hiding, but we do not cover them in this book. **Inheritance** and **polymorphism** are mechanisms that help to enable code reuse and contract-based programming, and are the subject of this chapter.

### 19.2 Introduction to Inheritance

Recall the Point class from earlier in the book:

```
class Point:

    def __init__(self, initX, initY):
        self.x = initX
        self.y = initY

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def __str__(self):
        return "x=" + str(self.x) + ",y=" + str(self.y)

p = Point(7, 6)
print(p)
```

Now, suppose we want to create a class that works like `Point` in every respect, but also keeps track of a short description for the point. We could create a `LabeledPoint` class by copying and pasting the definition for `Point`, changing the name to `LabeledPoint`, and modifying the class to suit our purposes. However, any time you copy and paste code, keep in mind that you are copying and pasting bugs that may exist in the code. Inheritance provides a way to reuse the definition of `Point` without having to copy and paste.

We begin like this:

```

class LabeledPoint(Point):
    pass

```

This example defines a class named `LabeledPoint` that inherits from the `Point` class. Putting the name `Point` in parenthesis tells Python that the new class, `LabeledPoint`, begins with all of the methods defined in its parent, `Point`. For example, we can instantiate `LabeledPoint` using the `Point` constructor, and invoke any `Point` methods we want to on it:

```

p = LabeledPoint(7,6)
dist = p.distanceFromOrigin()

```

Now, let's refine `LabeledPoint` so that it holds a label, along with the `x` and `y` coordinates:

```

class LabeledPoint(Point):

    def __init__(self, initX, initY, label):
        self.x = initX
        self.y = initY
        self.label = label

    def __str__(self):
        return "x=" + str(self.x) + ",y=" + str(self.y) +
               "(" + self.label + ")"

labeledPt = LabeledPoint(7,6,"Here")
print(labeledPt)

```

Here, we have redefined two of the methods that `LabeledPoint` inherits from `Point`: `__init__()` and `__str__()`. This is called *overriding*. When a child class redefines methods that are defined in its parent, we say that the child *overrides* the functionality inherited from its parent. When both the parent class and child class have a method with the same name, an invocation of the method on an instance of the child class executes code in the child's class; an invocation of the method on an instance of the parent class executes code in the parent's class. For example, consider the following:

```

pt = Point(5,10)
labeledPt = LabeledPoint(7, 6, "Here")

ptStr = str(pt)
labeledPtStr = str(labeledPt)

```

In Line 4, the call to `str(pt)` invokes the `__str__()` method in `Point`, because `pt` refers to an instance of `Point`. In Line 5, the call to `str(labeledPt)` invokes the `__str__()` method in `LabeledPoint`, because `labeledPt` refers to an instance of `LabeledPoint`.

### 19.3 Extending

If you compare the code in the `__init__` methods of `Point` and `LabeledPoint`, you can see that there is some duplication—the initialization of `x` and `y`. We can eliminate the duplication by having `LabeledPoint`'s `__init__()` method invoke `Point`'s `__init__()` method. That way, each class will be responsible for initializing its own instance variables.

A method in a child class that overrides a method in the parent can invoke the overridden method using `super()`, like this:

```

class LabeledPoint(Point):

    def __init__(self, initX, initY, label):
        super().__init__(initX, initY)
        self.label = label

```

In this example, line 4 invokes the `__init__()` method in `Point`, passing the values of `initX` and `initY` to be used in initializing the `x` and `y` instance variables.

Here is a complete code listing showing both classes, with a version of `__str__()` for `LabeledPoint` that invokes its parent's implementation using `super()` to avoid duplicating the functionality provided in `Point`.

```

class Point:

    def __init__(self, initX, initY):
        self.x = initX
        self.y = initY

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def __str__(self):
        return "x=" + str(self.x) + ",y=" + str(self.y)

class LabeledPoint(Point):

    def __init__(self, initX, initY, label):
        super().__init__(initX, initY)
        self.label = label

    def __str__(self):
        return super().__str__() + "(" + self.label + ")"

p = LabeledPoint(7,6,"Here")
print(p)
print(p.distanceFromOrigin())

```

## 19.4 Reuse Through Composition

Inheritance is not the only way to reuse code. *Composition* occurs when an object stores a reference to one or more objects in one of its instance variables. The object is thus able to reuse code in the objects it embeds within itself.

For example, our `LabeledPoint` example could have been implemented as follows:

```

class Point:

    def __init__(self, initX, initY):
        self.x = initX
        self.y = initY

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def __str__(self):
        return "x=" + str(self.x) + ",y=" + str(self.y)

```

```

class LabeledPoint:

    def __init__(self, initX, initY, label):
        self.point = Point(initX, initY)
        self.label = label

    def distanceFromOrigin(self):
        return self.point.distanceFromOrigin()

    def __str__(self):
        return str(self.point) + " (" + self.label + ")"

p = LabeledPoint(7,6,"Here")
print(p)
print(p.distanceFromOrigin())

```

The first thing to notice about this version of `LabeledPoint` is that it does not inherit from `Point`. Instead, its constructor instantiates a `Point` and stores a reference to it in its `point` instance variable so that it can be used by the other methods.

Next, notice how the `distanceFromOrigin()` method reuses the code in `Point`` by invoking it. We say that ```LabeledPoint`'s `distanceFromOrigin()` delegates its implementation to `Point`'s implementation.

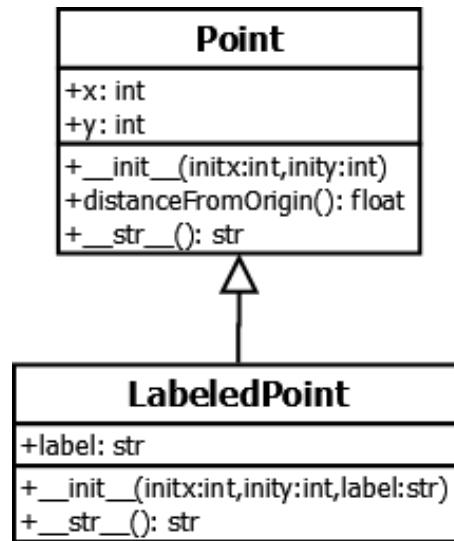
Finally, notice how the `__str__()` method also reuses the code in `Point` by calling `str(self.point)`.

## 19.5 Class Diagrams

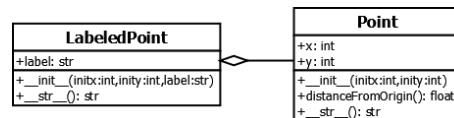
When two classes are involved in an inheritance or composition relationship, we say that an association exists between them. Often it is helpful to depict the associations between classes on a graphical diagram, so that developers working on the code can see at a glance how the classes are related to each other. The *Unified Modeling Language* (UML) is a graphical notation that provides a standard for depicting classes and their associations on various types of diagrams. It gives software engineers a way to record both their proposed designs, as well as the design of the final product. Put another way, UML provides the notation for constructing blueprints for software, and since it is widely used in both industry and academia, it is important for you to have some familiarity with it.

UML defines notations for several different kinds of diagrams. Here, we will introduce class diagrams, one of the most common UML diagrams. Class diagrams can contain a number of different elements, but we will focus on the basics: depicting classes with their instance variables and methods, and relationships to other classes.

On a UML class diagram, a class is depicted as a rectangle with three sections. The top section contains the class name; the middle section, the instance variables; and the bottom section, the methods. The following diagram shows the classes `Point` and `LabeledPoint` in their original inheritance relationship.



To depict composition, UML uses a different type of association, as shown in this figure:



## 19.6 Composition vs. Inheritance

Now you have seen two ways for a class to reuse code in another class. So, is one better than the other? When do you use inheritance, and when is composition the better choice?

Although the subject of this chapter is inheritance, the truth is that composition is usually a better choice than inheritance to reuse code. Perhaps 95% of cases where you are debating about choosing inheritance or composition, you should choose composition. It's hard to go wrong with composition, but you can get into all kinds of trouble if you go with inheritance and inheritance is not an appropriate choice.

So, it's easier to address the question of which technique to use by defining when inheritance is an appropriate choice. Inheritance is appropriate when the proposed child class (the one reusing the functionality in its parent) represents a *specialization* of its parent. Class A is a specialization of Class B if class A represents a specific type of class B. This is generally the case if you can fill in the following sentence with the names of the proposed child and parent classes:

**(child class) is a type of (parent class).**

Let's try some examples. Using the LabeledPoint example from the previous section: “*LabeledPoint* is a type of *Point*.” Since a *LabeledPoint* is a specific type of *Point*—a *Point* that has a label—that sentence makes sense. *LabeledPoint* is a specialization of *Point*, and inheritance is an appropriate choice.

Now, suppose you wanted to define a class that represents a rectangle. Like a *Point*, a *Rectangle* would need to keep track of an x and y location to determine its position, and might also have a width and a height. You're thinking about defining *Rectangle* to inherit from *Point*, so that it reuses all of *Point*'s functionality (like knowing its position and calculating its distance from origin), and adding just the two new instance variables it needs for its width

and height. From a pure code reuse standpoint, inheritance seems plausible. But wait—let’s apply the “is-a” linguistic test. Filling in the blanks in the sentence template above, we get: “*Rectangle* is a type of *Point*.” Most people would feel there is something wrong with that statement. A rectangle is *not* a more specific type of a point. A rectangle *contains* points and *consists of* points, but is not itself a point. Thus, it fails the linguistic test; composition is the better choice here.

So what happens if you decide to ignore the linguistic test and go ahead and make *Rectangle* inherit from *Point*? In some cases, you won’t run into trouble right away. Often, the difficulties don’t start to crop up until later, when you decide to add more methods to *Point* (the parent) that aren’t appropriate for *Rectangle* (the child). This leads to a program that is confusing to understand and contains bugs that occur when methods intended for *Point* are invoked on *Rectangle* instances by mistake. Also, since inheritance is the strongest form of relationship between classes, changes to code in a parent class have a stronger likelihood of breaking code in its children than would tend to occur if composition were used.

Inheritance is a powerful feature and, when used appropriately, a terrific way to reuse code. But, like most power tools, it can cut you up pretty badly if you don’t know what you are doing. Use it with caution and respect.

## 19.7 Case Study: Structured Postal Addresses

### 19.7.1 Introduction

Postal addresses are interesting things. Every country has its own format for postal addresses, and sometimes one country can have multiple address formats.

Postal addresses generally consist of a few standard elements: the recipient name; street address; city or locality; state or province; and a postal code. However, other elements are often included, such as neighborhood, district, post office identifier, and so on. For example, the following is

Mr. Abe Jones  
Acme Corporation  
123 Somewhere Ln  
Greenville, SC 29609  
USA

This same address would be written as follows for delivery to the Netherlands (in the example, the street, city, and state are unchanged, even though they do not exist in the Netherlands):

Acme Corporation  
Mr. Abe Jones  
Somewhere Ln 123  
29609 SC Greenville  
NETHERLANDS

Addresses in Ireland are complex, having up to 12 parts (such as building name and number, primary and secondary thoroughfare, primary and secondary locality, town, county, ...) plus an Eircode, a unique identifier assigned to each of the ~2 million addresses in Ireland. For example, Abe might live at the following address (English translation is given in parentheses, and would be omitted):

Abe Jones  
 Cnoc na Sceiche (The Hill of the Thorn)  
 Leac an Anfa (The Flagstone of the Storm)  
 Cathair na Mart (The City of the Beeves)  
 Co. Mhaigh Eo (The County of the Plain of the Yews)  
 A65 F4E2  
 IRELAND

(One would think that since each address has its own unique Eircode, it ought to be possible to address mail to Abe Jones, A65 F4E2, IRELAND. On second thought, perhaps that is not such a great idea. Can you imagine the practical concerns with such a scheme?)

### 19.7.2 Storing Postal Addresses

Suppose we want to write a contact management application. Among other things, the application stores names and addresses. What would be the best way to design a class that holds the information for an address? One approach would be to store the parts of the address that are consistent, such as the recipient name and the country, in instance variables, and store the rest of the address as a list of address lines:

```
class Address:
    def __init__(self, recipient, addressLines, country):
        self.country = country
        self.recipient = recipient
        self.addressLines = addressLines

addr = Address('Abe Jones', ['123 Somewhere Ln',
    'Greenville, SC 29609'], 'USA')
print(addr)
```

This approach treats an address as a collection of unstructured bits of information. If we want to look up an address, we can search by full name or country, but if we want to find all addresses in Greenville, or all addresses in zip code 29609, we can't do it very easily, since information such as city and zip code is mashed together in an unstructured address line along with the state abbreviation.

An approach that stores addresses as structured pieces of information might look like this:

```
class StructuredAddress:
    def __init__(self, country, recipient, street, city,
        state, postalCode):
        self.country = country
        self.recipient = recipient
        self.street = street
        self.city = city
        self.state = state
        self.postalCode = postalCode

    def display(self):
        print(self.recipient)
        print(self.street)
        print(self.city + ", " + self.state + " " +
            self.postalCode)
        print(self.country)
```

```

addr = StructuredAddress('USA', 'Abe Jones', '103 Anywhere Ln',
                        'Greenville', 'SC', '29609')
addr.display()

```

Now, if we have a list of StructuredAddress objects and we want to find all of the ones that hold addresses in Greenville, we can do it much more easily:

```

for addr in addrList:
    if addr.city == 'Greenville':
        addr.display()

```

### 19.7.3 Storing International Addresses

But now we have another problem. Our StructuredAddress works fine for U.S. addresses, but not for those of other countries. Suppose we want to handle Irish and Italian addresses. We might enhance the `display()` method to handle these with appropriate logic:

```

def display(self):
    print(self.recipient)

    if self.country == 'USA':
        print(self.street)
        print(self.city + ", " + self.state + " " +
              self.postalCode)
    elif self.country == 'IRELAND':
        print(self.postalCode)
    elif self.country == 'ITALY':
        print(self.street)
        print(self.postalCode + ' ' + self.city + ' ' +
              self.state)
    else:
        pass

    print(self.country)

```

This example works for Italian addresses because they conveniently have the same elements as U.S. addresses (just displayed in a slightly different order). For Irish addresses, we ignore the complicated address format and assume that the Irish post office will get mail to the recipient because of Ireland's unique Eircode scheme. But what if we wanted to include the additional elements of Irish addresses? We might create additional instance variables for those elements in our StructuredAddress class. However, you can probably see that approach will quickly become unwieldy.

### 19.7.4 Inheritance Applied

Let's apply inheritance to the problem of managing structured postal addresses. We will define a base class that contains the attributes in common to all postal addresses: `recipient` and `country`.

```

class BasePostalAddress:
    def __init__(self, country, recipient):
        self.country = country
        self.recipient = recipient

    def display(self):

```

```

        print(self.recipient)
        print(self.country)

    def validate(self):
        return self.recipient != '' and self.country != ''

```

This class isn't very useful by itself; relatively few people in the world could receive mail addressed to them using only their name and country. But it does establish two methods to perform functionality we want all addresses to perform: display themselves, and check whether the required information is present and of an appropriate length.

Next, we build on BasePostalAddress by creating a separate class for each country that inherits from it:

```

class IrishPostalAddress(BasePostalAddress):
    def __init__(self, recipient, postalCode):
        super().__init__("IRELAND", recipient)
        self.postalCode = postalCode

    def display(self):
        print(self.recipient)
        print(self.postalCode)
        print(self.country)

    def validate(self):
        return super().validate() and len(self.postalCode)
            == 7

class USPostalAddress(BasePostalAddress):
    def __init__(self, recipient, street, city, state,
                 zipcode):
        super().__init__("USA", recipient)
        self.street = street
        self.city = city
        self.state = state
        self.zip = zipcode

    def display(self):
        print(self.recipient)
        print(self.street)
        print(self.city + ", " + self.state + " " +
              self.zip)
        print(self.country)

    def validate(self):
        return (super().validate() and self.city != '' and
                len(self.state) == 2 and
                (len(self.postalCode) == 5 or
                 len(self.postalCode) == 9))

```

### 19.7.5 A List of Addresses

Now, let's construct a list containing both US and Irish addresses, and display them using a loop:

```

addrList = [IrishPostalAddress("Alf Jones", "A26F4G9"),
            USPostalAddress("Abe Jones", "103 Anywhere Ln",
                           "Greenville", "SC", "29609"),
            IrishPostalAddress("Gabe Jones", "A65F4E2")]

```

```

for addr in addrList:
    addr.display()

```

Normally, if a program iterates over a list that contains different types of objects, it has to be careful about making assumptions about the methods and operations that it can invoke on the different objects in the list, since an attempt to invoke a method or apply an operator to an object that does not support the method or operator will result in a runtime error. In this case, we know that all of the objects in the list inherit from `BasePostalAddress`. It is safe to invoke any methods defined in `BasePostalAddress`, since all children of `BasePostalAddress` are guaranteed to contain those methods. Programs that use inheritance often contain loops like this.

Notice something else. As the loop iterates over the list, each time the `display()` method is invoked, the computer will execute the one that is defined for the specific object referenced by `addr`. The first time through the loop, `addr` references an `IrishPostalAddress`, so the `display()` method for Irish addresses is invoked. The second time through the loop, the `display()` method in `USPostalAddress` is invoked. This behavior—where the computer always executes the method that is defined for the object being referenced—is called *polymorphism*. Python exhibits this behavior whether or not the objects in question utilize inheritance, but languages like Java and C++, polymorphism is available only through inheritance.

### 19.7.6 Using `isinstance`

Let's try something else with our list of addresses. Suppose we wanted to display all addresses with a given city. We might write some code like this:

```

for addr in addrList:
    if addr.city == 'Greenville':
        addr.display()

```

However, we would get into trouble on the first iteration of the loop. The first address is an Irish address, which does not have a `city` attribute. Python would raise an error. We want to perform this test only for US addresses.

In this case, since all addresses have a `country` attribute, we could write the loop this way:

```

for addr in addrList:
    if addr.country == 'USA' and addr.city == 'Greenville':
        addr.display()

```

Another way to test the address is to find out if the object belongs to a specific class. Python provides the `isinstance()` function for this purpose. `isinstance()` is designed for situations where you want to access a field or invoke a method on an object, but you want to do so only if the object provides the needed functionality. Given an object `obj` and a class `cls`, `isinstance(obj, cls)` returns True if `obj` is an instance of `cls` (or a subclass of `cls`), and False if it is not. Here is how we might use it in our loop:

```

for addr in addrList:
    if isinstance(addr, USPostalAddress) and addr.city ==
        'Greenville':
        addr.display()

```

In this version of the code, the `city` attribute will be tested only if `addr` references an instance of `USPostalAddress`, or a child of `USPostalAddress` (which would also have a `city` attribute).

Now that you've learned about `isinstance()`, you should know that, like inheritance itself, `isinstance()` should be used sparingly. Code that invokes `isinstance()` is often performing work on an object that the object should be designed to do itself, and is not utilizing inheritance and polymorphism to its full potential.

To make this loop better utilize inheritance and polymorphism, we need a way to test each address to see if it is in a given city. Let's add a method to `BasePostalAddress` for this purpose. It will return a boolean indicating whether the address is in a certain city.

```
class BasePostalAddress:  
    ...  
  
    def isInCity(self, city):  
        return False
```

`BasePostalCodes` do not have a `city` attribute, so they just return `False`. `USPostalCodes` do have a `city`, so we'll override this method for that class:

```
class USPostalAddress:  
    ...  
  
    def isInCity(self, city):  
        return self.city == city
```

Now, we rewrite our loop to use `isInCity()` to perform the test:

```
for addr in addrList:  
    if addr.isInCity('Greenville'):  
        addr.display()
```

Notice how we've eliminated the `isinstance()` test. Also, notice how this test works for `IrishPostalAddress` objects, even though we didn't define `isInCity()` for `IrishPostalAddress`, since `IrishPostalAddress` inherits its version from `BasePostalAddress`.

# Chapter 20

## Unit Testing

### 20.1 Introduction: Unit Testing

Testing plays an important role in the development of software. To this point, most of the testing you have done has probably involved running your program and fixing errors as you notice them. In this chapter, you will learn about a more methodical approach to testing. Along the way, you will pick up some design techniques. So, let's get started!

### 20.2 Checking Assumptions With `assert`

#### 20.2.1 Introduction

Many functions work correctly only for certain parameter values, and produce invalid results (or crash) if given others. Consider the following function, which computes the sum of the numbers in a range specified by its parameters:

```
def sumnums(lo, hi):
    """returns the sum of the numbers in the range
    [lo..hi]"""

    sum = 0
    for i in range(lo, hi+1):
        sum += i
    return sum

print(sumnums(1, 3))
print(sumnums(3, 1))
```

Notice that the first call to `sumnums` produces the correct answer (6), while the second call produces an incorrect answer. `sumnums` works correctly only if `lo` has a value that is less than, or equal to, `hi`.

This function trusts the calling code to provide parameter values that are valid. If the caller provides a second parameter that is lower than the first parameter, the function does not produce a correct result. That's not the fault of the function; the function isn't designed to work correctly if `lo > hi`.

To make it clear that the function is designed to work correctly only if `lo <= hi`, it's a good idea to state that as a precondition in the function documentation, like this:

```
def sumnums(lo, hi):
    """returns the sum of the numbers in the range [lo..hi]
```

```
Precondition: lo <= hi
"""

```

**Note 20.2.1 Precondition.** A **precondition** specifies a condition that must be True if the function is to produce correct results.

A precondition places a constraint on the values of the parameters that the caller can pass and expect to receive a valid result. Preconditions are boolean expressions – comparisons that you might write in an `if` statement. We'll have more to say about preconditions later in the chapter.

Code that calls a function is responsible for passing parameters that satisfy the function's preconditions. If the calling code passes values that violate the function's preconditions, the function isn't expected to work correctly. That's not the function's fault: it's the caller's fault for passing parameters to the function that the function is not designed to handle correctly. However, it might be a good idea if we designed the function to check for invalid values, and when it detects them, somehow report that it was called incorrectly.

### 20.2.2 Designing Defensive Functions

A defensive function is a function that checks its parameters to see if they are valid, and responds in an appropriate way if they are invalid. That raises the question: what should a defensive function do if it receives invalid values? Should it print an error? Silently ignore the problem and return a default value? Return a special value that indicates an error? Exit the program? There are several options.

As an example, here is one way we could make `sumnums` defensive:

```
def sumnums(lo, hi):
    """returns the sum of the numbers in the range [lo..hi]

    Precondition: lo <= hi
    """

    if lo > hi:
        print('Alert: Invalid parameters to sumnums.')
        return -1

    sum = 0
    for i in range(lo, hi+1):
        sum += i
    return sum

print(sumnums(1, 3))
print(sumnums(3, 1))
```

In this version, the function checks to see if the preconditions are violated, and if so, it complains by printing a message and returns the value -1 to the caller.

**Note 20.2.2 Defensive Programming.** The strategy of designing functions that check their parameters embodies a principle of software design called **defensive programming**, in which software checks for invalid inputs and responds in an appropriate way. Defensive programming is especially important for mission critical systems, but it can be a helpful strategy in regular software projects, as we'll soon see.

This is an improvement over the original function, because now, if the function is called with invalid data, the user will see a message that something

is wrong. However, the `if` statement adds three lines of code to the function. That may not seem like much, but it clutters the code and, in a typical program with several functions, those `if` statements will start to feel like undesirable baggage. There's a better way.

### 20.2.3 The `assert` Statement

Python provides a statement called the `assert` statement that can be used to check function preconditions. An `assert` statement checks the value of a boolean expression. If the expression is `True`, the `assert` statement allows the program to proceed normally. But if the expression is `False`, the `assert` statement signals an error and stops the program.

Here's an example of an `assert` statement:

```
x = 1 + 1
assert x == 2
print(x)
```

To see it in action, run the example above. You'll see the value `2` displayed. The boolean condition `x == 2` was `True`, and the `assert` statement allowed execution to continue.

Try changing the `assert` statement above as follows:

```
assert x == 3
```

Run this version of the code, and you'll see an `AssertionError` appear. That occurred because the value of the boolean expression was `False`.

Let's modify our `sumnums` function to use an `assert` statement to check the precondition:

```
def sumnums(lo, hi):
    """Returns the sum of the numbers in the range [lo..hi]

    Precondition: lo <= hi
    """

    assert lo <= hi

    sum = 0
    for i in range(lo, hi+1):
        sum += i
    return sum

print(sumnums(1, 3))
print(sumnums(3, 1))
```

In this version of `sumnums`, we've replaced the `if` statement with an `assert` statement. Notice that the boolean condition of the `assert` statement is the precondition, `lo <= hi`. When the function is called, if the condition is true, the function completes normally and returns its result. If the condition is false, the program stops with an `AssertionError`. So, the first call to `sumnums(1, 3)` succeeds and the result, `6`, appears. The second call to `sumnums(3, 1)` causes the `assert` to fail and an error appears.

Notice how much more streamlined this version of the function is than the version with the three-line `if` statement. Here, we've added just one line of code to the original version. Using assertions is a relatively low-effort way to create defensive functions.

**Note 20.2.3 Writing assert statements to check preconditions.** Writing assert statements to check preconditions is easy. They go at the **beginning** of the function. When you write an assert statement to check a precondition, if the function comment already contains a precondition, you often can simply take the precondition and put it directly into the `assert` statement (you might have to tweak it to make it syntactically legal). If there is no precondition in the function comment, think about how you would write an if statement to check that the values in the parameters are **correct**, and then put that condition after the word `assert`.

#### 20.2.4 More on `assert` and Preconditions

Let's discuss for a moment the question of what a defensive function should do when it receives invalid values in its parameters. By using an `assert` statement to check preconditions, we've designed the function to terminate the program if it is given bad data. Is this the right thing to do? If the program ends abruptly due to an assertion failure, the user will lose whatever work is in progress. That seems undesirable, to put it mildly.

Although a full discussion of defensive programming and assertions is outside the scope of an introductory programming textbook, think about this: an assertion error **indicates a bug in the program**. More specifically, the bug is a logic error that resulted in calling a function with inappropriate parameter values. If a computation is in progress and a logic error occurs, any results that computation might produce will be faulty. Logic errors often go silently undetected by users, because they aren't aware that the output is incorrect. It is better for a user to lose work than for a logic error to go undetected and produce an invalid result that might be unwittingly used. Therefore, using assert statements to check function preconditions is entirely appropriate.

Not only will adding assertions to your functions to check preconditions help expose logic errors in your program, it does so in a way that helps you track them down and fix them quickly. When you don't use assertions, a function that is called with incorrect parameters may produce erroneous results that aren't detected until much later in the program, and debugging the problem can be difficult to trace back to the source. When you use assertions to check preconditions, a function that detects a problem will stop immediately, helping you pinpoint the problem much faster. This behavior is called the **fail fast principle**. You want your program to fail as quickly after a logic error is detected as possible to help streamline the diagnostic work.

**Note 20.2.4 Debugging Assertion Failures.** When an `assert` statement that you have written to check a function precondition signals an error at runtime, your first thought will probably be: "what went wrong? where's the problem?" It will help if you remember that an assert that checks a function precondition is there to **catch bugs in code that calls the function**. After all, you put it on the first line of the function. So, it's not an indication of a problem in the function: instead, the calling code has a problem. So, look to see what code called the function. When you're running your program in a regular Python interpreter, the full error message will show the exact sequence of calls that triggered the error, and you can tell exactly which line of code is responsible for providing the incorrect values.

**Note 20.2.5 Functions that Cannot Fail.** An alternative approach to handling bad input for `sumnums` would be to design the function so that it works correctly regardless of whether the low end of the range is specified first or second. For example, we could design it so that both of the following calls

produce correct results:

```
print(sumnums(1, 3))
print(sumnums(3, 1))
```

It's not hard to do; I bet you could figure out how to tweak the function to work correctly for both of these calls without much effort. However, a more important question is: should we do that?

This question doesn't necessarily have a simple answer, but briefly, there are a couple of considerations that argue against it. First, refining the function to work correctly for both of these calls will result in a function that is slightly more complex, and therefore, perhaps more likely to contain bugs. Also, testing will be more involved; there are more cases to consider.

#### Check your understanding

**Checkpoint 20.2.6** An assert statement displays output if the condition is True.

A. True

B. False

**Checkpoint 20.2.7** Consider the following function. Which assert should be added to check its precondition?

```
def getfirst(msg):
    """returns first character of msg

    Precondition: len(msg) > 0
    """

    return msg[0]
```

A. assert len(msg) <= 0

B. assert len(msg) > 0

C. assert msg[0]

D. none of these

## 20.3 Testing Functions

### 20.3.1 Introduction

Melinda is writing a program that does some mathematical calculations. At the moment, she is working on adding some functionality to her program that requires rounding numbers to the nearest integer. She would normally use the built-in Python function `round` to do the job, but her program has a special requirement that numbers should be rounded up if the fractional portion is .6 or greater, instead of the usual .5 or greater. So, Melinda decides to write a function that rounds up numbers according to this requirement.

She defines a function `round6` to do the job:

```
def round6(num):
    """returns num rounded to nearest int if fractional part is >= .6"""

    return int(num + .6)
```

This function uses a valid approach to rounding, but is not quite correct (Melinda doesn't realize it yet — can you spot the bug?).

Now she needs to test the new code. There are two basic approaches Melinda could take to do her testing:

- 1 Put the function into the program, modify the program to call the function at the appropriate point, then run the program.
- 2 Test the function by itself, somehow.

Which do you think will be more efficient?

Melinda's program does complex mathematical calculations, and asks the user to enter 5 separate pieces of input before performing the calculations. If she goes with option 1, each time she runs the program to test the function, she must enter all 5 pieces of input. As you can imagine, that process is cumbersome and will not be very efficient. Also, if the program output is incorrect, it may be difficult to determine whether the fault is in the new function, or elsewhere in the program.

Melinda decides to write a separate, short program to help her test her new function. The test program is very simple — it contains only her new function and a bit of code to get some input, pass it to the function, and display the result. Here's what she writes:

```
def round6(num):
    """returns num rounded to nearest int if fractional
       part is >= .6"""

    return int(num + .6)

# ----- test program -----

x = float(input('Enter a number: '))
result = round6(x)
print('Result:', result)
```

Before running the program, she jots down some test cases to help her in her testing:

Input	Expected Output
-----	
Test Case 1:	3
Test Case 2:	4
Test Case 3:	4

Try running the program with the input values above. Notice that the output isn't quite right. Can you figure out how to correct the bug?

After analyzing her logic, Melinda corrects the bug by changing the return statement in the function as follows:

```
return int(num + .4)
```

She runs the test program again to verify that the function is working correctly. Then, she copies the `round6` function into her main program, confident that her rounding logic is correct.

The program Melinda wrote to help her test her `round6` function is an example of a unit test.

**Note 20.3.1 Unit Test.** A **unit test** is code that tests a function to determine if it works properly.

A unit test program like this one can dramatically reduce the effort it takes to test a new function, and can reduce the overall effort involved in adding functionality to a program. The savings tradeoff depends on the amount of effort required to write the test program, compared to the amount of effort required to test the function in the context of the main program for which the new function is being developed. Here, the function was relatively simple, and it probably wouldn't have taken Melinda too many iterations of testing the function in the context of the main program, with its five pieces of input. In this scenario, Melinda may not have saved much effort. However, if the function were more complex, writing a unit test would probably have helped reduce the overall effort. And, using some tricks I'll show you in the next sections, you can reduce the amount of effort required to write and run the unit test, making the case for writing unit tests even more compelling.

### 20.3.2 Automated Unit Tests

The unit test program above is a manual unit test. A **manual unit test** gets input from the user, invokes the code under test, providing the input supplied by the user, and displays the result. (In our example, `round6` is the code under test.) Manual unit tests are helpful, but they can be improved in two ways:

- 1 We can embed the test input directly within the unit test code, so the person running the test doesn't have to come up with the test input or take the time to enter it.
- 2 We can make the unit test report success or failure, instead of requiring the person running the test to look at the output and determine whether the function worked correctly.

We call a unit test that contains its own test input and produces a clear pass/fail indication an **automated unit test**. Take a look at the following example:

```
def round6(num):
    return int(num + .4)

# ---- automated unit test ----

result = round6(9.7)
if result == 10:
    print("Test 1: PASS")
else:
    print("Test 1: FAIL")

result = round6(8.5)
if result == 8:
    print("Test 2: PASS")
else:
    print("Test 2: FAIL")
```

This automated unit test invokes the `round6` function on predetermined test input, checks that the function produced the expected result, and displays a pass / fail message. Run it to see the test PASS messages.

Try editing the `round6` function above to introduce Melinda's original bug, then run it again to see the failure message. Notice the big advantage of an automated unit test: you can change the function being tested, run the unit test, and immediately see the test results for a whole series of tests. No hand-entry of test data, and no interpretation of the results. Clearly, once you have

the test written, you can dramatically speed up your edit-test-debug cycle. The downside, of course, is that the unit test program itself takes more time to develop.

### 20.3.3 Automated Unit Tests with `assert`

To help reduce the amount of effort required to develop an automated unit test, let's bring the `assert` statement into play. We can replace each `if` statement in the program above with an assert, as in the program below:

```
def round6(num: float) -> int:
    return int(num + .4)

# ---- automated unit test ----

result = round6(9.7)
assert result == 10

result = round6(8.5)
assert result == 8

print("All tests passed!")
```

Try running the program above to see the success message. Then, try altering the `round6` function to reintroduce the original bug, and see how the assertion failure pinpoints that the second test failed.

We can streamline this program even further by eliminating the `result` variable:

```
assert round6(9.7) == 10
assert round6(8.5) == 8

print("All tests passed!")
```

This is Really Nice. We have a short test program that contains its own test input and displays an automated pass or fail indication. Writing this program takes very little effort. We have the benefits of an automated test without having to write much code. Unit test programs are essentially “throw-away” programs that are used only during development, and it's important that they can be developed quickly and easily.

### 20.3.4 Unit Tests can have bugs

Unit tests, like the functions they test, can have bugs. So, when you run a unit test and it fails with an assert error, one of the first questions you need to ask yourself is: “Is the unit test correct?” If the unit test is incorrect, then you need to correct it, rather than spending time trying to find the bug in the function that the unit test is testing.

For example, consider the following assert:

```
assert round6(9.2) == 10
```

This unit test is incorrect, because `round6` should produce the value 9, not 10, when given the parameter 9.2.

#### Check your understanding

**Checkpoint 20.3.2** Rewrite the following 3 lines of code with a single assert:

```
result = engage_thruster(22)
if result != 'OK':
```

- ```
print("Test 2: FAIL")
```
- A. assert result != 'OK'
  - B. assert engage\_thruster(22) == result
  - C. assert engage\_thruster(22) != 'OK'
  - D. assert engage\_thruster(22) == 'OK'

**Checkpoint 20.3.3** Consider the following function which is supposed to return the first character of its argument:

```
def get_first(msg):
    return msg[1]
```

Now, consider this unit test:

```
assert get_first('Bells') == 'B'
```

This assertion fails. Is the unit test in error, or the function it is testing?

- A. Unit test
- B. Tested function
- C. Both are in error
- D. Both are correct

## 20.4 Designing Testable Functions

### 20.4.1 Introduction

Now that you know how to write unit tests using the `assert` statement, it's important for you to understand how to write testable functions. Not all functions can be tested.

Consider the following function:

```
def add(x, y):
    """Adds two numbers and displays the sum"""
    print(x + y)
```

How would you write an assert statement to check that this function works? Think about it a moment. Would this work?

```
assert add(2, 3) == 5
```

Answer: no. An assert statement cannot verify that what a function displays on the screen is correct. It can only check that the contents of variables are correct. This function is not testable.

A **testable function** is a function that produces a result that can be checked by an assert statement. Generally, it does so in one of three ways:

- 1 It returns its result
- 2 It stores its result in a global variable
- 3 It modifies the state of an object passed as a parameter

Functions that display their output using `print` are not testable functions.

**Check your understanding**

**Checkpoint 20.4.1** Is this a testable function?

```
sum = 0
def add(x, y):
    global sum
    sum = x + y
```

- A. Yes.
- B. No.

### 20.4.2 Design by Contract

In addition to producing a result that can be checked by an assert statement, a testable function must have a clear specification. In order to write unit tests for a function, you must have a precise understanding of what the function should do.

A function specification describes what value the function produces, given its parameter values, and is generally expressed in the form of a docstring. For example, consider the `sumnums` function given earlier in this chapter:

```
def sumnums(lo, hi):
    """returns the sum of the numbers in the range [lo..hi]

    Precondition: lo <= hi
    """
    ...
```

The docstring is this function's specification. Given this specification, you might write a unit test that contains the following assert:

```
assert sumnums(1, 3) == 6
```

An alternate way to write the docstring is as follows:

```
def sumnums(lo, hi):
    """computes the sum of a range of numbers

    Precondition: lo <= hi
    Postcondition: returns the sum of the numbers in the range [lo..hi]
    """
    ...
```

This docstring contains three elements: a brief description; a precondition; and a postcondition. We've discussed the concept of a precondition earlier in this chapter. The postcondition is new.

**Note 20.4.2 Postcondition.** A **postcondition** states the work the function completed by the function if the precondition is satisfied.

Functions that include a precondition and a postcondition in their docstring embody a software engineering idea called **design by contract**. The idea is that a function specification forms a contract between the function and the code calling the function. If the code calling the function passes parameters that satisfy the function's precondition, then the function should be expected to produce what it says it will produce. If the parameters do not satisfy the function's precondition, then the function does not have to produce a valid result. In the design by contract approach, a testable function is one where the function's postcondition can be verified by an assert statement.

In this example, you can think of the function’s docstring as promising to calling code: “If you give me two parameters, `lo` and `hi`, such that `lo` is less than or equal to `hi`, I promise to return the sum of the numbers in the range `lo..hi`, inclusive.”

To write a precondition, think about the parameter values that the function is designed to handle, and write a boolean expression that expresses what parameter values are valid. For example, consider a function that computes the average weight, given a total weight and a number of items:

```
def compute_average(total_weight: float, num_items: float) -> float:
    return total / num_items
```

This function will work if `num_items` is greater than zero, but will fail if `num_items` is zero. So, an appropriate precondition would be `num_items > 0`. A complete docstring would look like this:

```
def compute_average(total_weight: float, num_items: float) -> float:
    """computes the average weight, given 'total_weight' of items and 'num_items'

    Precondition: num_items > 0
    Postcondition: returns average item weight
    """
```

Sometimes, your precondition will be expressed more loosely, using English. Consider this function which extracts the first word from a string containing text:

```
def get_first_word(text: str) -> str:
    """extracts the first word from 'text'

    space_loc = text.find(' ')
    return text[0:space_loc]
```

This function will produce nonsense if the string doesn’t contain a space. So, an appropriate precondition might be “text contains 2 or more words separated by spaces”. The docstring might be:

```
def get_first_word(text: str) -> str:
    """extracts the first word from 'text'

    Precondition: 'text' contains 2 or more words separated by spaces
    Postcondition: returns the first word in 'text'
    """
```

Following the design by contract idea and writing function specifications that include preconditions and postconditions is an excellent way to design testable functions, because, as we’ll see in the next section, it makes it possible to reason precisely about what the function should do when given various parameter values. Even if you don’t use precondition and postcondition terminology in your docstrings, it helps to *think* in those terms.

### Check your understanding

**Checkpoint 20.4.3** Consider the following function. What would an appropriate precondition be?

```
def getfirst(msg):
    """returns first character of msg"""

    return msg[0]
```

- A. `len(msg) <= 0`
- B. `len(msg) > 0`
- C. `msg == ""`
- D. none of these

## 20.5 Writing Unit Tests

### 20.5.1 Introduction

Once you have designed a testable function, with a clear docstring specification, writing unit tests is not difficult. In this section, you'll learn how to do just that.

Let's start with our `sumnums` function:

```
def sumnums(lo, hi):
    """computes the sum of a range of numbers

    Precondition: lo <= hi
    Postcondition: returns the sum of the numbers in the range [lo..hi]
    """

    sum = 0
    for i in range(lo, hi+1):
        sum += i
    return sum
```

As we've seen, to write a unit test, you devise test cases for the function, and then write assert statements that call the function and check that the function produced the expected results. The following assert statements would be appropriate for a unit test for `sumnums` :

```
assert sumnums(1, 3) == 6
assert sumnums(1, 1) == 1
```

But what about the following?

```
assert sumnums(3, 1) == 0
```

Note that `sumnums` produces the value `0` for cases where the `lo` values exceeds the `hi` value, as is the case in this assert. So, like the first two asserts above, this assert would pass. However, it is not an appropriate assertion, because the specification says nothing about what the function produces if `lo` is greater than `hi`.

The unit test should be written such that it passes even if the function implementation is altered in a way that causes some other value than 0 to be returned if `lo` exceeds `hi`. For example, we might want to redesign the function to be more efficient — for example, use Gauss's formula for summing numbers, as in the following:

```
def sumnums(lo, hi):
    """computes the sum of a range of numbers

    Precondition: lo <= hi
    Postcondition: returns the sum of the numbers in the range [lo..hi]
    """
```

```
    return (hi * (hi + 1) / 2) - (lo * (lo - 1) / 2)
```

This version will produce correct results if the precondition is satisfied. Like the original function, it produces incorrect results if the precondition is violated — but unlike the original function, the values produced if the precondition is violated are not necessarily 0.

### 20.5.2 Specification-Based Testing

A key idea to remember when writing a unit test is that your test must always respect the function’s preconditions. The docstring states what the function should do, with the assumption that parameter values meet the preconditions. It does not state what the function should do if the parameter values violate the preconditions.

Writing an assert that violates the functions preconditions is not a good idea, because to determine what the function will produce for that case, you must look into the implementation of the function and analyze its behavior. That is called **implementation-based testing**, and it leads to brittle tests that are likely to fail if you rework the function implementation. When you write tests are based only on the function specification, without looking at the implementation, you are doing specification-based testing.

**Note 20.5.1 Specification-Based Tests.** Specification-based tests are tests that are designed based only on the information in the function specification, without considering any of the details in the function implementation.

Specification-based tests are preferred over implementation-based tests, because they are more resilient. They will continue to pass even if you rework the function implementation.

#### Check your understanding

**Checkpoint 20.5.2** Consider the following function. Indicate which of the asserts would be appropriate for a unit test.

```
def repeat(s: str, num: int) -> str:
    """Duplicates a string

    Precondition: `num` >= 0
    Postcondition: Returns a string containing `num` copies of `s`
    """
    if num >= 0:
        return s * num
    else:
        return ''
```

- A. assert repeat('\*', 0) == ''
- B. assert repeat('\*', -1) == ''
- C. assert repeat('-', 5) == '----'
- D. assert repeat('\*', 5) == '\*\*\*'

**Checkpoint 20.5.3** Write assert statements below to test a function with the following specification. Your asserts should check that the function produces an appropriate value for each of the three postcondition cases.

```

def grade(score):
    """Determines letter grade given a numeric score

Precondition: 0 <= score <= 100
Postcondition: Returns 'A' if 90 <= score <= 100,
'B' if 80 <= score < 90, 'F' if 0 <= score < 80
"""

```

Note: Line numbers in any assert error messages that appear while you are developing and testing your answer will not be accurate.

```

# Write assert statements to test grade()

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        code = self.getEditorText().replace(' ', '').replace('\'',
            '').replace('"', '')
        self.assertTrue(testA and '==A' in code, "Assert tested 90..100")
        self.assertTrue(testB and '==B' in code, "Assert tested 80..90")
        self.assertTrue(testF and '==F' in code, "Assert tested 0..80")
        self.assertFalse(illegal, "No asserts violated preconditions")

myTests().main()

```

## 20.6 Test-First Development

### 20.6.1 Introduction

The idea of unit tests has been around a long time, and most people agree that writing unit tests is a good idea. However, when deadlines loom and time is at a premium, the unit tests often don't get written. That's a problem, because studies have shown that projects with good unit tests often are more robust, with fewer bugs, than projects that don't have good unit tests.

In a traditional development process, when a programmer needs to create a new function, he writes the function, and then, if he has time, writes a unit test for it. If he doesn't have time, he doesn't write the unit test: he tests the function in the context of the program being developed. One day, someone decided that it might be a good idea to reverse the order: write the unit test *first*, and then write the function. That led to the idea of Test-First Development.

**Note 20.6.1 Test-First Development.** Test-First Development is an approach to writing software that involves writing a unit test for a function before writing the function.

In this section, we'll explore the idea of test-first development to see how it can help.

A programmer using Test-First Development writes a new function using the following steps:

- 1 First, create the function interface and docstring.
- 2 Next, create a unit test for the function.
- 3 Run the unit test. It should fail.
- 4 Write the body of the function.
- 5 Run the unit test. If it fails, debug the function, and run the test again.  
Repeat until the test passes.

As an example, suppose that we're going to write our `sumnums` function using the Test-First methodology. We begin by creating the interface and docstring:

```
def sumnums(lo, hi):
    """computes the sum of a range of numbers

    Precondition: lo <= hi
    Postcondition: returns the sum of the numbers in the range [lo..hi]
    """
```

Next, we write the unit test for it:

```
def sumnums(lo, hi):
    """computes the sum of a range of numbers

    Precondition: lo <= hi
    Postcondition: returns the sum of the numbers in the range [lo..hi]
    """

    assert sumnums(1, 3) == 6
    assert sumnums(1, 1) == 1
    print("All tests passed!")
```

We run the unit test and it fails.

Next, we implement the body of `sumnums`:

```
def sumnums(lo, hi):
    """computes the sum of a range of numbers

    Precondition: lo <= hi
    Postcondition: returns the sum of the numbers in the range [lo..hi]
    """

    sum = 0
    for i in range(lo, hi):
        sum += i
    return sum

    assert sumnums(1, 3) == 6
    assert sumnums(1, 1) == 1
    print("All tests passed!")
```

Now, run the tests. The tests indicate an assertion error, which points to a bug in the function logic. Fix the bug, and test again. (If you're not sure what the bug is, try using **Show in CodeLens** and stepping through the code to help you figure it out.)

The range function generates numbers in the range `lo .. hi - 1`. Our function should include `hi`. Try adjusting the `hi` parameter for range to `hi + 1`.

Suppose we're not creating a new function, but modifying an existing one. In Test-First Development, before making the modification to the function, we write a test for the new functionality. Then, we modify the function, and use the test to check that the modification worked.

### 20.6.2 Benefits of Test-First Development

There are several benefits to Test-First Development.

- 1 It ensures that unit tests are written. This tends to lead to higher-quality, robust code, with fewer bugs.
- 2 Writing the tests first helps the programmer to clarify the function specification. It's not possible to write an assert for a function that has a vague function docstring. This process forces the programmer to write a clear docstring and to practice specification-based testing, because when the tests are written, there is no function implementation to reference.
- 3 When the programmer writes the function and is ready to test it, the test is all ready to go. There is no internal struggle about whether a unit test should be written or not. The programmer runs the test, and gets instant feedback about whether the function is working or not.
- 4 If the function fails to pass the test, the benefits of unit testing in helping the programmer to quickly diagnose and fix the problem are instantly available. The test-debug cycle is rapid.
- 5 When a programmer modifies an existing function for which unit tests already exist, perhaps to add some more functionality, the existing unit tests serve as a safety net. They check that the modifications made by the programmer don't break any of the old functionality.
- 6 The overall development time tends to be reduced. Perhaps counter-intuitively, writing more code (the unit tests) actually speeds up the overall development process, because of the benefits imparted by unit testing.
- 7 Believe it or not, there are psychological benefits. As the programmer works on the project, creating little tests and then writing code that passes those tests, there is a sense of accomplishment and satisfaction that comes every time a new test passes. Instead of spending hours of frustration debugging a new function in the context of a complex program, with few visible results, the test-first progress leads to more visible and regular successes.

I hope you'll try out Test-First Development on your next assignment and experience some of these benefits for yourself!

#### Check your understanding

**Checkpoint 20.6.2** Test-First Development often involves writing more code than traditional development.

- A. True
- B. False

## 20.7 Testing with pytest

### 20.7.1 Introduction

Writing automated unit tests is very helpful in reducing the effort needed to build software. However, the simple approach described so far is inadequate to help programmers realize the full benefits of unit testing. In this section, we introduce a unit test framework which addresses some practical issues that crop up when you try to apply unit testing techniques in software development projects. Here are some of the issues with using plain assertion unit tests:

- Simple assert-based tests don't give very good diagnostic information when they fail. It would help to have better reporting. For example, when an assert fails, it would help us in diagnosing the error to see what value the function actually produced. An `AssertionError` doesn't give us that information.
- We need a better way to organize our unit test code. So far, I've suggested creating separate programs to hold the functions under test together with their unit test code. But that isn't practical for most projects. For example, functions often need to call other functions in the program in order to do their work. It's not convenient to bring those auxiliary functions over into separate test programs.
- We need a way to keep the unit test around and use it even after the function is first developed to help us catch bugs that are inadvertently introduced into the function when we make modifications to it.

Unit testing frameworks help to address these issues, by improving error reporting, providing a structure for programmers to organize their unit tests, and making it possible to leverage existing unit tests when making enhancements to functions. `pytest` is one unit testing framework that provides these benefits.

For our purposes, the attractive thing about `pytest` is that writing unit tests with `pytest` feels a lot like writing unit tests using plain assert statements. The only difference is that you put your assert statements into test functions. Here's an example of how it works:

```
def round6(num: float) -> int:
    """This function has a bug in it"""
    return int(num + .6)

# ---- automated unit test ----

def test_round6():
    assert round6(9.7) == 10
    assert round6(8.5) == 8

=====

from unittest.gui import TestCaseGui

class pyTests(TestCaseGui):

    def testOne(self):
        for item in globals():
            if item.startswith("test_"):
                try:
```

```

        globals()[item]()
        self.assertEqual(item + "passed", item
                        + "passed", item + "passed")
    except Exception as e:
        self.assertEqual(item + "failed", item
                        + "passed", item + "failed:" + str(e))

pyTests().main()

```

This code example defines two functions: the function to be tested, `round6`, and a function named `test_round6` that contains unit test code. When using the pytest approach, you write your unit test as a function whose name must start with the prefix `test_`. Inside the function, you write normal assert statements to test the desired function. Notice that you do not write a line to *call* the unit test function. Instead, when you launch pytest to run the unit tests, pytest scans your script and executes only the functions with the prefix `test_`.

This ActiveCode environment simulates pytest by scanning for and executing functions with a `test_` prefix when you click **Run**. Go ahead and try it - rename the `test_round6` function to `test_itworks` and try running the test again.

### 20.7.2 Organizing pytest Functions

The example above uses a single pytest function, with both asserts in the same pytest function. The disadvantage of that approach is that the first failing assert prevents the rest of the asserts from being tested.

If you want, you can write multiple pytest functions to test a single function. That way, when an assert fails in one test function, the rest of the pytest functions can still run and report success or failure.

You can name your pytest functions with names that indicate what they are testing. For example, try changing the ActiveCode example above so that it defines two test functions: one named `test_round6_rounds_up`, containing the first assert, and one named `test_round6_rounds_down`, containing the second assert. Your code should look like this:

```

def test_round6_rounds_up():
    assert round6(9.7) == 10

def test_round6_rounds_down():
    assert round6(8.5) == 8

```

If you use good pytest function names, when a pytest function has an assertion failure, you can easily tell what the problem was.

### 20.7.3 Using pytest

To use pytest, you must first install it using the `pip` command. Open your computer's command line window (not the Python interpreter) and enter the following command to install:

- Windows:

```
pip install pytest
```

- Mac/Linux:

```
pip3 install pytest
```

After you have installed pytest, you run pytest unit tests from the command line window. To run pytest unit tests, try copying the code from the ActiveCode example above and pasting it into a Python file named (ex.) **myround.py**. Then, use the **pytest** command to run your tests by opening a command window, navigating to the folder where you stored myround.py, and executing the following command:

```
pytest myround.py
```

#### 20.7.4 Understanding pytest Failure Reports

When you run the pytest command and an assertion fails, you see a report like this:

```
===== FAILURES =====
test_round6 -----
def test_round6():
    assert round6(9.7) == 10
>     assert round6(8.5) == 8
E     assert 9 == 8
E     +  where 9 = round6(8.5)

myround.py:8: AssertionError
```

Let's take a closer look at this report to understand what it's telling you.

- 1 First, notice the line with the > symbol:

```
>     assert round6(8.5) == 8
```

The > symbol points to the line with the assertion that failed.

- 2 Next, notice the lines marked E:

```
E     assert 9 == 8
E     +  where 9 = round6(8.5)
```

This indicates that the call to `round6(8.5)` returned the value 9, instead of the value 8. The value 9 is the actual result of the function. Knowing the value actually produced by the function can help you to troubleshoot the bug and correct the problem.

#### 20.7.5 Integrated Unit Testing with pytest

When you use the pytest framework, you can include pytest test functions in your main program, along with the rest of your program code. This allows you to keep your tests together with the functions that they test, and you can run either your program (using the `python` command) or the unit tests (using the `pytest` command).

Take a look at this example that shows a function (`round6`, containing a bug), together with a unit test function (`test_round6`), and a main program that uses `round6`:

```
def round6(num: float) -> int:
    return int(num + .6)

# ---- automated unit test ----

def test_round6():
```

```

    assert round6(9.7) == 10
    assert round6(8.5) == 8

    # ----- main program follows -----

    if __name__ == '__main__':
        num = float(input('Enter a value: '))
        print('The value rounded is: ' + str(round6(num)))

```

Notice how the main program is inside the `if` statement on line 12. This `if` condition is true when the program is run using the `python` command, and allows the main program to execute. When the unit tests are executed using the `pytest` command, any top-level code outside a function in the python file gets executed when `pytest` scans the script looking for unit test functions with a `test_` prefix. The `if` condition is false in this scenario, and that prevents the main program from executing when `pytest` is scanning the script. If that explanation didn't make total sense, just remember: in order for `pytest` to work correctly, any code that is part of the main program must be inside an `if` statement like the one in this example, so that it doesn't interfere with `pytests`'s unit testing process.

### Check your understanding

**Checkpoint 20.7.1** Write a `pytest` unit test function named `test_grade` to test a function with the following specification. Your asserts should check that the function produces an appropriate value for each of the three postcondition cases.

```

def grade(score):
    """Determines letter grade given a numeric score

    Precondition: 0 <= score <= 100
    Postcondition: Returns 'A' if 90 <= score <= 100,
    'B' if 80 <= score < 90, 'F' if 0 <= score < 80
    """

```

```

# Write a pytest unit test function named ``test_grade``

=====
from unittest.gui import TestCaseGui

testA = False
testB = False
testF = False
illegal = False

def grade(score):
    global illegal, testA, testB, testF

    if score > 100 or score < 0:
        illegal = True
        return ''
    elif score >= 90:
        testA = True
        return 'A'
    elif score >= 80:
        testB = True
        return 'B'
    else:
        testF = True
        return 'F'

class myTests(TestCaseGui):

    def testOne(self):
        code = self.getEditorText().replace('`')
        ' `').replace('`', '').replace('`', '')
        self.assertEqual(test_grade(), None, 'test_grade` function`defined`')
        self.assertTrue(testA and '==A' in code, "Assert` tested`90..100")
        self.assertTrue(testB and '==B' in code, "Assert` tested`80..90")
        self.assertTrue(testF and '==F' in code, "Assert` tested`0..80")

myTests().main()

```

## 20.8 Glossary

### Glossary

**assert statement.** A statement that verifies that a boolean condition is true.  
**design by contract.** An approach to designing functions that specifies function behavior using preconditions and postconditions.

**postcondition.** Part of a function specification that states the work the function completed by the function if the precondition is satisfied.

**precondition.** A boolean condition that the caller of a function must ensure is true in order for the function to produce correct results

**specification-based tests.** tests that are designed based only on the information in the function specification, without considering any of the details in

the function implementation.

**unit test.** Code that tests a function to determine if it works properly

## 20.9 Exercises

1. A function named `reverse` takes a string argument, reverses it, and returns the result:

```
1 def reverse(astring):  
    """Returns the reverse of 'astring'"""
```

Complete the assert statements in the ActiveCode editor below to create a unit test for `reverse`. Your asserts should check that `reverse` works properly for the following test cases (“Input” refers to the value passed as a parameter, and “Expected Output” is the result returned from `reverse`):

| Input              | Expected Output |
|--------------------|-----------------|
| -----              |                 |
| Test Case 1: 'abc' | 'cba'           |
| Test Case 2: 'b'   | 'b'             |
| Test Case 3: ''    | "               |

```

def test_reverse():
    # Complete the assert statements below

    assert _____
    assert _____
    assert _____
=====

from unittest.gui import TestCaseGui

testABC = False
testB = False
testEmpty = False

def reverse(astring):
    """Returns the reverse of astring"""
    global testABC, testB, testEmpty

    if astring == 'abc':
        testABC = True
    elif astring == 'b':
        testB = True
    elif astring == '':
        testEmpty = True

    l = list(astring)
    l.reverse()
    return ''.join(l)

class myTests(TestCaseGui):

    def testOne(self):
        test_reverse()
        code = self.getEditorText().replace('`',
            '').replace('`', '').replace('`', '')
        self.assertTrue(testABC and '==cba' in
            code, "Assert tested `abc`")
        self.assertTrue(testB and '==b' in code,
            "Assert tested `b`")
        self.assertTrue(testEmpty and '==\n' in
            code, "Assert tested ``")

myTests().main()

```

- <sup>2</sup> 2. A function named `striplatters` takes a string argument, removes all letters from it, and displays the result (see below). However, this function is not testable.

Modify the function so that it can be tested with the assert statements that follow.

```
def stripletters(msg):
    result = ''
    for ch in msg:
        if not ch.isalpha():
            result += ch

    print(result)

assert stripletters('ab12c') == '12'
assert stripletters('12') == '12'

=====

from unittest.gui import TestCaseGui

class myTests(TestCaseGui):

    def testOne(self):
        self.assertEqual(stripletters('ab12c'),
                        '12', "stripletters('ab12c') == '12'")
        self.assertEqual(stripletters('12'), '12',
                        "stripletters('12') == '12'")

myTests().main()
```

# Chapter 21

## Labs

### 21.1 Astronomy Animation

#### 21.1.1 Object-oriented programming

```
import turtle
import math

class SolarSystem:

    def __init__(self, width, height):
        self.thesun = None
        self.planets = []
        self.ssturtle = turtle.Turtle()
        self.ssturtle.hideturtle()
        self.ssscreen = turtle.Screen()
        self.ssscreen.setworldcoordinates(-width/2.0,-height/2.0,width/2.0,height/2.0)
        # self.ssscreen.tracer(50)

    def addPlanet(self, aplanet):
        self.planets.append(aplanet)

    def addSun(self, asun):
        self.thesun = asun

    def showPlanets(self):
        for aplanet in self.planets:
            print(aplanet)

    def freeze(self):
        self.ssscreen.exitonclick()

    def movePlanets(self):
        G = .1
        dt = .001

        for p in self.planets:
            p.moveTo(p.getXPos() + dt * p.getXVel(),
                     p.getYPos() + dt * p.getYVel())

            rx = self.thesun.getXPos() - p.getXPos()
            ry = self.thesun.getYPos() - p.getYPos()
```

```
r = math.sqrt(rx**2 + ry**2)

accx = G * self.thesun.getMass()*rx/r**3
accy = G * self.thesun.getMass()*ry/r**3

p.setXVel(p.getXVel() + dt * accx)

p.setYVel(p.getYVel() + dt * accy)

class Sun:
    def __init__(self, iname, irad, im, itemp):
        self.name = iname
        self.radius = irad
        self.mass = im
        self.temp = itemp
        self.x = 0
        self.y = 0

        self.sturtle = turtle.Turtle()
        self.sturtle.shape("circle")
        self.sturtle.color("yellow")

    def getName(self):
        return self.name

    def getRadius(self):
        return self.radius

    def getMass(self):
        return self.mass

    def getTemperature(self):
        return self.temp

    def getVolume(self):
        v = 4.0/3 * math.pi * self.radius**3
        return v

    def getSurfaceArea(self):
        sa = 4.0 * math.pi * self.radius**2
        return sa

    def getDensity(self):
        d = self.mass / self.getVolume()
        return d

    def setName(self, newname):
        self.name = newname

    def __str__(self):
        return self.name

    def getXPos(self):
        return self.x

    def getYPos(self):
        return self.y

class Planet:
```

```
def __init__(self, iname, irad, im, idist, ivx, ivy, ic):
    self.name = iname
    self.radius = irad
    self.mass = im
    self.distance = idist
    self.x = idist
    self.y = 0
    self.velx = ivx
    self.vely = ivy
    self.color = ic

    self.pturtle = turtle.Turtle()
#self.pturtle.speed('fast')
    self.pturtle.up()
    self.pturtle.color(self.color)
    self.pturtle.shape("circle")
    self.pturtle.goto(self.x, self.y)
    self.pturtle.down()

def getName(self):
    return self.name

def getRadius(self):
    return self.radius

def getMass(self):
    return self.mass

def getDistance(self):
    return self.distance

def getVolume(self):
    v = 4.0/3 * math.pi * self.radius**3
    return v

def getSurfaceArea(self):
    sa = 4.0 * math.pi * self.radius**2
    return sa

def getDensity(self):
    d = self.mass / self.getVolume()
    return d

def setName(self, newname):
    self.name = newname

def show(self):
    print(self.name)

def __str__(self):
    return self.name

def moveTo(self, newx, newy):
    self.x = newx
    self.y = newy
    self.pturtle.goto(newx, newy)

def getXPos(self):
```

```
        return self.x

    def getYPos(self):
        return self.y

    def getXVel(self):
        return self.velx

    def getYVel(self):
        return self.vely

    def setXVel(self, newvx):
        self.velx = newvx

    def setYVel(self, newvy):
        self.vely = newvy

def createSSAndAnimate():
    ss = SolarSystem(2,2)

    sun = Sun("SUN", 5000, 10, 5800)
    ss.addSun(sun)

    m = Planet("MERCURY", 19.5, 1000, .25, 0, 2, "blue")
    ss.addPlanet(m)

    m = Planet("EARTH", 47.5, 5000, 0.3, 0, 2.0, "green")
    ss.addPlanet(m)

    m = Planet("MARS", 50, 9000, 0.5, 0, 1.63, "red")
    ss.addPlanet(m)

    m = Planet("JUPITER", 100, 49000, 0.7, 0, 1, "black")
    ss.addPlanet(m)

    m = Planet("Pluto", 1, 500, 0.9, 0, .5, "orange")
    ss.addPlanet(m)

    m = Planet("Asteroid", 1, 500, 1.0, 0, .75, "cyan")
    ss.addPlanet(m)

    numTimePeriods = 10000
    for amove in range(numTimePeriods):
        ss.movePlanets()

    ss.freeze()

createSSAndAnimate()
```

## 21.2 Turtle Racing Lab

### 21.2.1 Random Numbers

Before we begin writing code for this lab, we need to introduce one more Python module. The `random` module allows us to generate random numbers. It's easy to use:

```

import random
x = random.randrange(1,10)
print(x)

```

The `randrange` function as called in the example above, generates a random number from 1 to 9. Even though we said 10 the `randrange` function works just like the `range` function when it comes to starting and stopping points. Now if you run the program over and over again you should see that each time you run it a different number is generated. Random numbers are the basis of all kinds of interesting programs we can write, and the `randrange` function is just one of many functions available in the `random` module.

### 21.2.2 Turtle Races

In this lab we are going to work step by step through the problem of racing turtles. The idea is that we want to create two or more turtles and have them race across the screen from left to right. The turtle that goes the farthest is the winner.

There are several different, and equally plausible, solutions to this problem. Let's look at what needs to be done, and then look at some of the options for the solution. To start, let's think about a solution to the simplest form of the problem, a race between two turtles. We'll look at more complex races later.

When you are faced with a problem like this in computer science it is often a good idea to find a solution to a simple problem first and then figure out how to make the solution more general.

Here is a possible sequence of steps that we will need to accomplish:

- 1 Import the modules we need
- 2 Create a screen
- 3 Create two turtles
- 4 Move the turtles to their starting positions
- 5 Send them moving across the screen

Here is the Python code for the first 4 steps above

```

import turtle           # 1. import the modules
import random
wn = turtle.Screen()      # 2. Create a screen
wn.bgcolor('lightblue')

lance = turtle.Turtle()    # 3. Create two turtles
andy = turtle.Turtle()
lance.color('red')
andy.color('blue')
lance.shape('turtle')
andy.shape('turtle')

andy.up()                  # 4. Move the turtles to their
                           starting point
lance.up()
andy.goto(-100,20)
lance.goto(-100,-20)

# your code goes here

wn.exitonclick()

```

Now, you have several choices for how to fill in code for step 5. Here are some possibilities to try. Try coding each of the following in the box above to see the different kinds of behavior.

- Use a single call to `forward` for each turtle, using a random number as the distance to move.
- Create a for loop, using a random number for the parameter passed to the `range` function. Inside the for loop move one of the turtles forward by some number of units.
- Create a single for loop using something like 150 or 200 as the range parameter. Then inside the for loop move each turtle forward using a random number as the parameter to `forward`.

So, which of these programs is better? Which of these programs is most correct? These are excellent questions. Program 1 is certainly the simplest, but it isn't very satisfying as far as a race is concerned. Each turtle simply moves their distance on their turn. That is not very satisfying as far as a simulated race goes. Program 2 ends up looking a lot like Program 1 when you run it. Program 3 is probably the most 'realistic' assuming realism is very important when we're talking about a simulated race of virtual turtles.

You may be thinking why can't each turtle just move forward until they cross some artificial finish line? Good question! We'll get to the answer to this, and look at the program in a later lesson when we learn about something called the `while` loop.

## 21.3 Drawing a Circle

In this lesson we are going to develop a function that uses a turtle to draw a circle. As we develop this function we will investigate several problem solving strategies that illustrate how computer scientists think as they solve problems.

In figuring out how to write this function we must realize that there are some limitations of our ability to draw a circle. First, we are limited by the turtle functions we already know about, for example, `left`, `right`, `forward`. There is not a method for drawing a curved line with a turtle (well, actually there is, but we're not going to use it).

One of the first strategies we will employ is called *simplification*. Let's look at something simpler than drawing a circle and see what we can learn, as we look at some simpler examples we'll try to *generalize* what we learn to help us build a more complex function.

One thing we already know how to do is to write a function to draw a square. Now a square doesn't look anything like a circle, but maybe if we look at some other simple shapes that will help us. Just to remind you, here is the code for the `drawSquare` function.

```
def drawSquare(t, sz):
    """Make a turtle draw a square of sz."""
    for i in range(4):
        t.forward(sz)
        t.left(90)
```

Although this isn't going to do anything if you click on the run button, you could fill in the rest of the code to make it work, right?

Now, see if you can modify the code for `drawSquare` to make it draw a triangle. Call this new function `drawTriangle`. When you finish `drawTriangle` write a third function called `drawOctagon`. When you are finished follow the link below to the next part of this lesson.

OK, they work now [Section 21.4](#)

## 21.4 Lessons from a Triangle

OK, let's look at what we have learned from writing a `drawTriangle` and `drawOctagon` function. The first thing you had to figure out was that you needed to modify the parameter to the `range` function based on the number of sides in the polygon. The next thing, and this may have been the most challenging part for you was to figure out how much to turn each time. The following table summarizes what you probably learned very nicely. If you didn't, look at the table and then go back to the previous page and see if you can finish `drawTriangle` and `drawOctagon`.

**Table 21.4.1**

| Shape    | Sides | range() | Angle         |
|----------|-------|---------|---------------|
| Triangle | 3     | 3       | $360/3 = 120$ |
| Square   | 4     | 4       | $360/4 = 90$  |
| Octagon  | 8     | 8       | $360/8 = 45$  |

Looking at the table above you can really see that there is a pattern. If you know the number of sides you want, the rest can be figured out from there. This leads us to the next problem solving stage of this exercise, generalization. Why write a separate function for every kind of polygon when you can just write a single function that can be used to draw many different polygons?

Our new function `drawPolygon` will have three parameters, a turtle and the length of the side just like we have in the previous functions, but now we will add an additional parameter: `numSides`.

Here's the starting point for the `drawPolygon` function, see if you can fill in the details on your own.

```
import turtle

def drawPolygon(t, sideLength, numSides):
    # your code here.

    wn = turtle.Screen()
    geo = turtle.Turtle()

    drawPolygon(geo, 30, 10) # draw a decagon

    wn.exitonclick()
```

## 21.5 Counting Letters

### 21.5.1 Simple Solution: Counting a Single Letter

We have previously discussed the solution to the problem of counting the number of times a specific letter appears in a string. In the case below, that specific letter is “a”.

```

def countA(text):
    count = 0
    for c in text:
        if c == 'a':
            count = count + 1
    return count

print(countA("banana"))

```

Of course, we could also solve this problem by using the `count` method provided for strings.

```

def countA(text):

    return text.count("a")

```

### 21.5.2 General Solution: Counting All Letters

Now we will generalize the counting problem and consider how to count the number of times each letter appears in a given string. In order to do this we need to realize that writing a function that returns a single integer will no longer work. Instead we will need to return some kind of collection that holds the counts for each character.

Although there may be many possible ways to do this, we suggest a dictionary where the keys of the dictionary will be the letters in the string and the associated values for each key will be the number of times that the letter appeared.

What about a letter that does not appear in the string? It will never be placed in the dictionary. By assumption, any key that is not in the dictionary has a count of 0.

If we call the function `countAll`, then a call to `countAll` would return the dictionary. For example,

```

| print(countAll("banana"))

would return the dictionary

| {"a":3, "b":1, "n":2}

```

Here is a start to the development of the function.

- 1 Define the function to require one parameter, the string.
- 2 Create an empty dictionary of counts.
- 3 Iterate through the characters of the string, one character at a time.

## 21.6 Letter Count Histogram

The previous lab wrote a function to return a dictionary of letter counts. In an earlier chapter, we wrote a turtle program that could draw a histogram.

Combine these two ideas together to create a function that will take a string and create a histogram of the number of times each letter occurs. Make sure it is in alphabetical order from left to right.

- 1 Count the number of times each letter occurs. Keep the count in a dictionary.
- 2 Get the keys from the dictionary, convert them to a list, and sort them.

- 3 Iterate through the keys, in alphabetical order, getting the associated value (count).
- 4 Make a histogram bar for each.

## 21.7 Approximating the Value of Pi

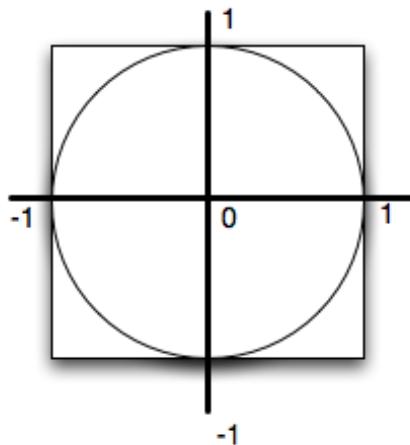
Almost everyone has heard of the famous mathematical constant called **Pi**. We use it most often to find the circumference or the area of a circle. For simplicity, the value that is commonly used for pi is 3.14. However, it turns out that pi is what mathematicians call an **irrational number**, meaning that that it has an infinite, nonrepeating number of decimal digits. The value is

3.1415926535897932384626433832795028841971693993751058209749445923078164062...

In this lab, we will approximate the value of pi using a technique known as **Monte Carlo Simulation**. This means that we will use random numbers to simulate a “game of chance”. The result of this game will be an approximation for pi.

### 21.7.1 Setup

The game that we will use for this simulation is “darts”. We will “randomly” throw a number of darts at a specially configured dartboard. The set up for our board is shown below. In the figure, you can see that we have a round dartboard mounted on a square piece of wood. The dartboard has a radius of one unit. The piece of wood is exactly two units square so that the round board fits perfectly inside the square.



But how will this help us to approximate pi? Consider the area of the circular dartboard. It has a radius of one so its area is  $\pi$ . The area of the square piece of wood is 4 ( $2 \times 2$ ). The ratio of the area of the circle to the area of the square is  $\pi/4$ . If we throw a whole bunch of darts and let them randomly land on the square piece of wood, some will also land on the dartboard. The number of darts that land on the dartboard, divided by the number that we throw total, will be in the ratio described above ( $\pi/4$ ). Multiply by 4 and we have  $\pi$ .

## 21.7.2 Throwing Darts

Now that we have our dartboard setup, we can throw darts. We will assume that we are good enough at throwing darts that we always hit the wood. However, sometimes the darts will hit the dartboard and sometimes they will miss.

Inorder to simulate throwing the darts, we can generate two random numbers between zero and one. The first will be the “x coordinate” of the dart and the second will be the “y coordinate”. However, we have a problem. The coordinates for the dartboard go from -1 to 1.

How can we turn a random number between 0 to 1 into a random number between -1 and 1?

The program has been started for you. You need to fill in the part that will “throw the dart”. Once you know the x,y coordinate, have the turtle move to that location and make a `dot`. Note that the tail is already up so it will not leave a line.

```
import turtle
import math
import random

fred = turtle.Turtle()

wn = turtle.Screen()
wn.setworldcoordinates(-1,-1,1,1)

fred.up()

numdarts = 10
for i in range(numdarts):
    randx = random.random()
    randy = random.random()

    x =
    y =

wn.exitonclick()
```

## 21.7.3 Counting Darts

We already know the total number of darts being thrown. The variable `numdarts` keeps this for us. What we need to figure out is how many darts land in the circle? Since the circle is centered at (0,0) and it has a radius of 1, the question is really simply a matter of checking to see whether the dart has landed within 1 unit of the center. Luckily, there is a turtle method called `distance` that will return the distance from the turtle to any other position. It needs the x,y for the other position.

For example, `fred.distance(12,5)` would return the distance from fred’s current position to position (12,5).

Now we simply need to use this method in a conditional to ask whether fred is within 1 unit from the center. If so, color the dart `red`, otherwise, color it `blue`. Also, if we find that it is in the circle, count it. Create an accumulator variable, call it `insideCount`, initialize it to zero, and then increment it when necessary. Remember that the increment is a form of the accumulator pattern using reassignment.

### 21.7.4 The Value of Pi

After the loop has completed and visualization has been drawn, we still need to actually compute pi and print it. Use the relationship given above.

Run your program with larger values of `numdarts` to see if the approximation gets better. If you want to speed things up for large values of `numdarts`, set the tracer to be 100 using `wn.tracer(100)`.

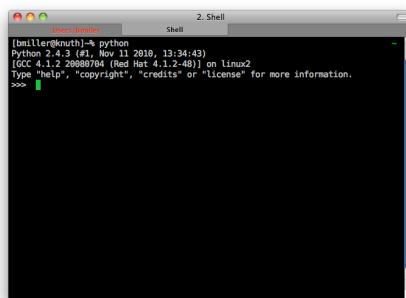
## 21.8 Python Beyond the Browser

Although having Python available for you to use right in the browser is convenient, there are some limitations, and Python is a real programming language used for real applications by some very large and impressive corporations. In this Advanced Topic we'll show you how to use the Python shell along with a simple text editor. In a later advanced topic we will introduce you to something called an Integrated Development Environment or IDE that will make life even better for you when working on a larger Python project.

### 21.8.1 The Python Shell

One of the most important ways you can learn computer science is by simply experimenting or trying things. Unlike chemistry where you can explode things and do real damage, in computer science you really can't go wrong by just experimenting a bit. In the worst case you might have to reboot your computer, but its pretty hard to do any long lasting damage.

For those of you used to fancy graphical user interfaces the Python shell make look a bit primitive, but don't be fooled by the lack of fancy interface, you can do a lot of powerful stuff in the shell. On a Mac or with Linux Python is already installed for you. There is a short video link at the end of this topic that explains how to download and install Python on Windows. Here is an example of what the shell looks like once you have it started up.



To run the Python shell you will first need to start up the Terminal application, on the Mac you can find this under Utilities in the Applications folder, on most versions of Linux you can find it under the accessories menu. Once you have a terminal started, you simply type `python` and press the return key. To start the shell under windows simply go to the start menu and choose Python (command line) from the menu.

Now that you have the shell started you may wonder what you can do with it? In the shell you can do anything you would do in a Python program. Any of the examples you have seen in this chapter can be typed in directly to the shell. Any Python expression can be entered into the shell and you will see

the result printed out for you right underneath. Here are some examples using the functions and expressions introduced in this chapter.

```
bmillerm@chronos:~$ python3.1
Python 3.1.2 (r312:79147, Jul 22 2010, 08:34:07)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> type(4)
<class 'int'>
>>> x = "Hello world"
>>> x
'Hello world'
>>> x
'Hello world'
>>> y = 100.0
>>> y
100.0
>>> z = x+y
>>> z
104.5
>>> x
'Hello world'
>>> y
100.0
>>>
```

It's a good idea to get in the habit of using the shell. Very often if you have a question about how something works you can answer the question for yourself by simply trying it in the shell. If you need a really extensible calculator you can simply start up your Python shell and use it to calculate just about anything. Soon you'll find yourself writing little Python snippets for all kinds of things.

### 21.8.2 Running a Python Program

Of course the problem with the Python shell is that you can't save anything, so you always have to retype whatever you want to do over again. It's also difficult to do anything more than a few lines long because if you make a typo you end up retying everything. Fortunately there is a solution for that as well. Python allows you to write a program, and save it as a text file with the extension .py and then you can run that program right from the command line.

Here is a simple example of a Python program.

```
print('HelloWorld')
print('2+3=', 2 + 3)
```

Let's assume that you have typed in the lines above in Notepad or TextEdit or some similar editor, if you already know emacs or vi you are a awesome! Now save the file as testprog1.py and then head back to the terminal. Now at the command line of the terminal type python testprog1.py and you will see the following output

```
bmillerm@chronos:~$ python testprog1.py
Hello World
2 + 3 = 5
```

Anything we do in the following chapters that appears in the editing window in the web page can be done exactly the same in the terminal just like the small example above. This example illustrates one very important difference between entering expressions in the Python shell, and writing a Python program. The Python shell uses what we call a read eval print loop. That is, Python *reads* an expression from the command line, then *evaluates* that expression, and finally *prints* the result. In a python program that you run using the Python interpreter, you have to be explicit about what it is you want to print. That is why in the Python program we use the print function on both lines.

### 21.8.3 Installing Python on Windows

If you are using windows you will need to install Python for yourself. Here is a

video that explains how to do it.<iframe width="425" height="349" src="http://www.youtube.com/embed/9EfGpN1Pnsg" frameborder="0" allowfullscreen></iframe>We are assuming that you use Python 3.x, as of this writing the latest version is Python 3.2.1. This link will tell you how to update or install Python on Linux, Mac, or Windows. [Install Python<sup>1</sup>](#)

### 21.8.4 Glossary

#### Glossary

**terminal.** A terminal is a program now days, but not too many years ago computer scientists did their work at a hardware device called a terminal. The terminal was connected by wire, or phone line to a computer somewhere else. Yes, the internet has not always been here.

**command line.** The command line is often synonymous with terminal in that when you are using the terminal you are also using the command line. It's where you type in commands, and then the computer interprets those commands and responds to you with results.

## 21.9 Experimenting With the 3n+1 Sequence

In this lab we will try to gain a bit more information about the 3n+1 sequence. We will start with the code from the chapter and make modifications. Here is the function so far.

```
def seq3np1(n):
    """Print the 3n+1 sequence from n, terminating when it reaches 1."""
    while n != 1:
        print(n)
        if n % 2 == 0:          # n is even
            n = n // 2
        else:                  # n is odd
            n = n * 3 + 1
        print(n)                # the last print is 1

seq3np1(3)
```

- 1 Count the number of iterations it takes to stop.

Our program currently **prints** the values in the sequence until it stops at 1. Remember that one of the interesting questions is How many items are in the sequence before stopping at 1?. To determine this, we will need to count them.

First, comment out (or delete) the print statements that currently exist. Now we will need a local variable to keep track of the count. It would make sense to call it count. It will need to be initialized to 0 since before we begin the loop.

Once inside the loop, we will need to update count by 1 (increment), so that we can keep track of the number of iterations. It is very important that you put these statements in the right place. Notice that the previous location of the print statements can be very helpful in determining the location.

---

<sup>1</sup><http://www.diveintopython3.net/installing-python.html>

When the loop terminates (we get to 1), **return** the value of `count`.

This demonstrates an important pattern of computation called a **counter** (note that it is a type of accumulator). The variable `count` is initialized to 0 and then incremented each time the loop body is executed. When the loop exits, `count` contains the result — the total number of times the loop body was executed.

Since the function now returns a value, we will need to call the function inside of a print statement in order to see the result.

- 2 Repeat the call to `seq3np1` using a range of values, up to and including an upper bound.

Now that we have a function that can return the number of iterations required to get to 1, we can use it to check a wide range of starting values. In fact, instead of just doing one value at a time, we can call the function iteratively, each time passing in a new value.

Create a simple for loop using a loop variable called `start` that provides values from 1 up to 50. Call the `seq3np1` function once for each value of `start`. Modify the print statement to also print the value of `start`.

- 3 Use the turtle graphics to graph the number of iterations. This provides an interesting visual that allows you to see the relative number of iterations for each value. You will probably want to use `setworldcoordinates` to make your graph an appropriate scale. You should also use the turtle to write out the loop variable and the number of iterations if the number of iterations is more than 100.
- 4 Keep track of the maximum number of iterations.

Scanning this list of results causes us to ask the following question: What is the longest sequence? The easiest way to compute this is to keep track of the largest count seen so far. Each time we generate a new count, we check to see if it is larger than what we think is the largest. If it is greater, we update our largest so far and go on to the next count. At the end of the process, the largest seen so far is the largest of all.

Create a variable call `maxSoFar` and initialize it to zero. Place this initialization outside the **for loop** so that it only happens once. Now, inside the for loop, modify the code so that instead of printing the result of the `seq3np1` function, we save it in a variable, call it `result`. Then we can check `result` to see if it is greater than `maxSoFar`. If so, update `maxSoFar`.

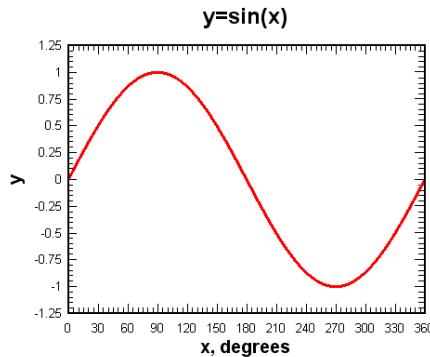
Experiment with different ranges of starting values.

## 21.10 Plotting a sine Wave

Have you ever used a graphing calculator? You can enter an equation, push a few buttons, and the calculator will draw a line. In this exercise, we will use our turtle to plot a simple math function, the sine wave.

### 21.10.1 What is the sine function?

The sine function, sometimes called the sine wave, is a smooth, repetitive oscillation that occurs often in many fields including mathematics, physics, and engineering. A single repetition is shown below. Note that the x axis is given in degrees.



For this lab, we will use the math library to generate the values that we need. To help you understand the sine function, consider the following Python program. As you can see, the `sin` function from the math library takes a single parameter. This parameter must be a value in “radians” (you may remember this from trigonometry class). Since most of us are used to stating the size of an angle in “degrees”, the math module provides a function, `radians` that will convert from degrees to radians for us.

```
import math

y = math.sin(math.radians(90))
print(y)
```

The program above shows us that the sine of 90 degrees is 1. Note that the figure above agrees with that. Try a few other values, like 0 degrees, 180 degrees, 38 degrees, and so on. You should be able to match the results up with the picture shown above.

It might be even more interesting to iterate through a sequence of angles and see the value of the sine function change. Try it for angles between 0 and 180 degrees. What do you notice about the results?

```
import math

for angle in range(????):
    y = math.sin(math.radians(angle))
    print(y)
```

Now try it for some other boundary values, like 270 or 360.

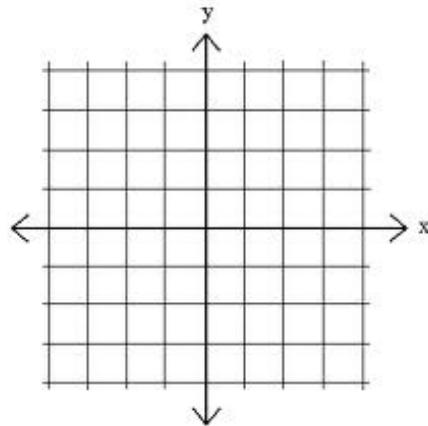
### 21.10.2 Making the Plot

In order to plot a smooth line, we will use the turtle’s `goto` method. `goto` takes two parameters, `x` and `y`, and moves the turtle to that location. If the tail is down, a line will be drawn from the previous location to the new location.

```
| fred.goto(50,60)
```

Recall that the default turtle screen starts with the turtle in the middle at position (0,0). You can think of the screen as a piece of graph paper. The x axis runs horizontally and the y axis runs vertically. The point where they

meet in the middle is (0,0). Positions to the left of the center have an x value that is negative. Positions that are below the center have a y value that is negative.



Let's try the `goto` method. Experiment with the method to make sure you understand the coordinate system of the screen. Try both positive and negative numbers.

```
import math
import turtle

wn = turtle.Screen()
wn.bgcolor('lightblue')

fred = turtle.Turtle()

fred.goto(50,60)

wn.exitonclick()
```

Now we can put the two previous programs together to complete our plot. Here is our sequence of steps.

- 1 Create and set up the turtle and the screen.
- 2 Iterate the angle from 0 to 360.
  - Generate the sine value for each angle.
  - Move the turtle to that position (leave a line behind).

Here is a partial program for you to complete.

```
import math
import turtle

wn = turtle.Screen()
wn.bgcolor('lightblue')

fred = turtle.Turtle()

#your code here

wn.exitonclick()
```

### 21.10.3 Making the Plot Better

You probably think that the program has errors since it does not draw the picture we expect. Maybe you think it looks a bit like a line? What do you think the problem is? Here is a hint...go back and take a look at the values for the sine function as they were calculated and printed in the earlier example.

Now can you see the problem? The value of `sin` always stays between -1 and 1. This does not give our turtle much room to run.

In order to fix this problem, we need to redesign our “graph paper” so that the coordinates give us more room to plot the values of the sine function. To do this, we will use a method of the `Screen` class called `setworldcoordinates`. This method allows us to change the range of values on the x and y coordinate system for our turtle. Take a look at the documentation for the turtle module to see how to use this method ([Global Module Index<sup>1</sup>](#)). Once you have an understanding of the parameters required to use the method, choose an appropriate coordinate system and retry your solution.

**Note 21.10.1 Now try this....** Now that you can plot a sine function, how about trying a different function, such as cosine or log?

---

<sup>1</sup><http://docs.python.org/py3k/py-modindex.html>

## 21.11 Operator precedence table

The following table summarizes the operator precedence of Python operators *in this book*, from highest precedence (most binding) to lowest precedence (least binding). Operators in the same box have the same precedence. Unless syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation, which groups from right to left). This is many of the entries from the complete Python table at <https://docs.python.org/3/reference/expressions.html#operator-precedence><sup>1</sup>.

In the row for comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature; for example `3 < x <= y != z`.

**Table 21.11.1**

| Operator                                                              | Description                       |
|-----------------------------------------------------------------------|-----------------------------------|
| (expressions...), [expressions...], {key: value...}, {expressions...} | Binding or tuple display, list    |
| x[index], x[index:index], x(arguments...), x.attribute                | Subscription, slicing, call, attr |
| **                                                                    | Exponentiation (groups right)     |
| -x                                                                    | Negation                          |
| *, /, //, %                                                           | Multiplication, real and integer  |
| +, -                                                                  | Addition and subtraction          |
| in, not in, is, is not, <, <=, >, >=, !=, ==                          | Comparisons, including mem        |
| not x                                                                 | Boolean NOT                       |
| and                                                                   | Boolean AND                       |
| or                                                                    | Boolean OR                        |

---

<sup>1</sup><https://docs.python.org/3/reference/expressions.html#operator-precedence>