REAL python™

Part 1: Introduction to Python

Table of Contents

Introduction

Whether you're new to programming or a professional code monkey looking to dive into a new language, this course will teach you all of the practical Python that you need to get started on projects on your own.

Real Python emphasizes real-world programming techniques, which are illustrated through interesting, useful examples. No matter what your ultimate goals may be, if you work with a computer at all, you will soon be finding endless ways to improve your life by automating tasks and solving problems through Python programs that you create.

Python is open-source freeware, meaning you can download it for free and use it for any purpose. It also has a great support community that has built a number of additional free tools. Need to work with PDF documents in Python? There's a free package for that. Want to collect data from webpages? No need to start from scratch!

Python was built to be easier to use than other programming languages. It's usually much easier to read Python code and MUCH faster to write code in Python than in other languages.

For instance, here's some simple code written in C, another commonly used programming language:

```
#include <stdio.h>

int main(void)
{
  printf ("Hello, world\n");
}
```

All the program does is print "Hello, world" on the screen. That was a lot of work to print one phrase! Here's the same code in Python:

```
print("Hello, world")
```

Simple, right? Easy, faster, more readable.

At the same time, Python has all the functionality of other languages and more. You might be surprised how many professional products are built on Python code: Gmail, Google Maps, YouTube, reddit, Spotify, turntable.fm, Yahoo! Groups, and the list goes on… And if it's powerful enough for both NASA and the NSA, it's good enough for us.

Why this course?

There are tons of books and tutorials out there for learning Python already. However, most of the resources out there generally have two main problems:

- They aren't practical.
- They aren't interesting.

Most books are so preoccupied with covering every last possible variation of every command that it's easy to get lost in the details. In the end, most of them end up looking more like the Python documentation pages. This is great as reference material, but it's a horrible way to learn a programming language. Not only do you spend most of your time learning things you'll never use, but it isn't any fun!

This course is built on the 80/20 principle. We will cover the commands and techniques used in the vast majority of cases and focus on how to program real-world solutions to problems that ordinary people actually want to solve.

This way, I guarantee that you will:

- Learn useful techniques much faster
- Spend less time struggling with unimportant complications
- Find more practical uses for Python in your own life
- Have more fun in the process!

If you want to become a serious, professional Python programmer, this course won't be enough by itself - but it will still be the best starting point. Once you've mastered the material in this course, you will have gained a strong enough foundation that venturing out into more advanced territory on your own will be a breeze.

So dive in! Learn to program in a widely used, free language that can do more than you ever thought was possible.

How to use this course

The first half of this course is a quick yet thorough overview of all the Python fundamentals. You do not need any prior experience with programming to get started. The second half, meanwhile, is focused on solving interesting, real-world problems in a practical manner.

For the most part, you should approach the topics in the first half of this course in the same order as they are presented. This is less true of the second half, which covers a number of mostly non-overlapping topics, although the chapters are generally increasing in difficulty throughout. If you are a more experienced programmer, then you may find yourself heading toward the back of the course right away - but don't neglect getting a strong foundation in the basics first!

Each chapter section is followed by review exercises to help you make sure that you've mastered all the topics covered. There are also a number of assignments, which are more involved and usually require you to tie together a number of different concepts from previous chapters. The practice files that accompany this course also include solution scripts to the assignments as well as some of the trickier exercises - but to get the most out of them, you should try your best to solve the assignment problems on your own before looking at the example solutions.

This course does move quickly, however, so if you're completely new to programming, you may want to supplement the first few chapters with additional practice. I highly recommend working through the beginning Python lessons available for free at the Codecademy site while you make your way through the beginning of this material as the best way to make sure that you have all the basics down.

Finally, if you have any questions or feedback about the course, you're always welcome to contact me directly.

Learning by doing

Since the underlying philosophy is learning by doing, do just that: Type in each and every code snippet presented to you. **Do not copy and paste**.

You will learn the concepts better and pick up the syntax faster if you type each line of code out yourself. Plus, if you screw up - which will happen over and over again - the simple act of correcting typos will help you learn how to debug your code. Finish all review exercises and give each homework assignment and the larger projects a try on your own before getting help from outside resources.

With enough practice, you will learn this material - and hopefully have fun along the way!

How long will it take to finish this course?

It depends on your background. Those familiar with another language could finish the course in as little as 35 to 40 hours, while those just starting, with no experience, may spend up to 100 hours. Take your time.

Course Repository

This course has an accompanying repository containing the course source code as well as the answers to exercises and assignments. Broken up by chapter, you can check your code against the code in the repository after you finish each chapter.

> You can download the course files directly from the repository. Press the 'Download ZIP' button which is located at the right-side of the page. This allows you to download the most recent version of the code as a zip archive. **Be sure to download the updated code for each release.**

License

Conventions

Formatting

Code blocks will be used to present example code.

```
print(“Hello world!”)
```

Terminal commands follow the Unix format:

```
$ python hello-world.py
```

(dollar signs are not part of the command)

Italic text will be used to denote a file name:

hello-world.py.

Bold text will be used to denote a new or important term:

Important term: This is an example of what an important term should look like.

NOTES, WARNINGS, and SEE ALSO boxes appear as follows:

> **NOTE:** This is a note filled in with bacon impsum text. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.
>
> **WARNING:** This is a warning also filled in with bacon impsum. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.
>
> **SEE ALSO:** This is a see also box with more tasty impsum. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

Errata

I welcome ideas, suggestions, feedback, and the occasional rant. Did you find a topic confusing? Or did you find an error in the text or code? Did I omit a topic you would love to know more about. Whatever the reason, good or bad, please send in your feedback.

You can find my contact information on the Real Python website. Or submit an issue on the Real Python official support repository. Thank you!

> **NOTE**: The code found in this course has been tested on Mac OS X v. 10.10.5, Windows 7 and 10, Linux Mint 17, and Ubuntu 14.04.3 LTS.

Getting Started

Let's start with the basics...

Download Python

Before we can do anything, you need to download Python. Even if you already have Python on your computer, make sure that you have the correct version. Python 3.5 is the version used in this course and by most of the rest of the world.

Mac users: You already have a version of Python installed by default, but it's not quite the same as the standard installation. You should still download *Python 3.5.1* as directed below. Otherwise, you might run into problems later when trying to install some additional functionality in Python or running code that involves graphics windows.

Linux users: You might already have at least Python 2.7.6 installed by default. Open your Terminal application and type `python version` or `python3 version` to find out. You should go ahead and update to the latest version.

If you need to, go to http://www.python.org/download/ to download Python 3.5.1 for your operating system and install the program.

> **NOTE**: For further assistance, please refer to **Appendix A** for a basic tutorial on installing Python.

Open IDLE

We'll be using IDLE (Interactive DeveLopment Environment) to write our Python code.

IDLE is a simple editing program that comes automatically installed with Python on Windows and Mac, and it will make our lives much easier while we're coding.

You could write Python scripts in any program from a basic text editor to a very complex development environment (and many professional coders use more advanced setups), but IDLE is simple to use and will easily provide all the functionality we need. I personally use a more advanced text editor called Sublime Text for most projects.

To open IDLE...

OS X

Go to your Applications folder and click on "IDLE" from the "Python 3.5" folder to start running IDLE. Alternatively, you can type "IDLE" (without quotes) into your Terminal window to launch IDLE.

Windows

Go to your start menu and click on "IDLE (Python GUI)" from the "Python 3.5" program folder to open IDLE. You can also type "IDLE" into the search bar.

Linux

I recommend that you install IDLE to follow along with this course. You could use Vim or Emacs, but they will not have the same built-in debugging features.

To install IDLE with admin privileges:

- On Ubuntu/Debian, type: `sudo apt-get install idle`
- On Fedora/Red Hat/RHEL/CentOS, type: `sudo yum install python-tools`
- On SUSE, you can search for IDLE via "install software" through YaST.

Opening IDLE, you will see a brief description of Python, followed by a prompt:

```
>>>
```

We're ready to program!

Write a Python script

The window we have open at the moment is IDLE's interactive window; usually this window will just show us results when we run programs that we've written, but we can also enter Python code into this window directly. Go ahead and try typing some basic math into the interactive window at the prompt - when you hit enter, it should evaluate your calculation, display the result and prompt you for more input:

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

> **NOTE**: A Python prompt makes for a great calculator if you need to quickly crunch some numbers and don't have a calculator handy.

Let's try out some actual code. The standard program to display "Hello, world" on the screen is just that simple in Python. Tell the interactive window to print the phrase by using the print command like so:

```
>>> print("Hello, world")
Hello, world
>>>
```

> **NOTE**: If you want to get to previous lines you've typed into the interactive window without typing them out again or copying and pasting, you can use the pair of shortcut keys ALT+P (or on a Mac, CTRL+P). Each time you hit ALT+P, IDLE will fill in the previous line of code for you. You can then type ALT+N (OS X:CTRL+N) to cycle back to the next most recent line of code.

Normally we will want to run more than one line of code at a time and save our work so that we can return to it later. To do this, we need to create a new script.

From the menu bar, choose "File -> New Window" to generate a blank script. You should rearrange this window and your interactive results window so that you can see them both at the same time.

Type the same line of code as before, but put it in your new script:

```
print("Hello, world")
```

> **WARNING**: If you just copy and paste from the interactive window into the script, make sure you never include the ">>> " part of the line. That's just the window asking for your input; it isn't part of the actual code.

In order to run this script, we need to save it first. Choose "File -> Save As . . .", name the file *hello_world.py* and save it somewhere you'll be able to find it later. The ".py" extension lets IDLE know that it's a Python script.

> **NOTE**: Notice that print and "Hello, world" appear in different colors to let you know that print is a command and "Hello, world" is a string of characters. If you save the script as something other than a ".py" file (or if you don't include the ".py" extension), this coloring will disappear and everything will turn black, letting you know that the file is no longer recognized as a Python script.

Now that the script has been saved, all we have to do in order to run the program is to select "Run -> Run Module" from the script window (or hit F5, or try shift+command+f5), and we'll see the result appear in the interactive results window just like it did before:

```
>>>
Hello, world
>>>
```

To open and edit a program later on, just open up IDLE again and select "File -> Open...", then browse to and select the script to open it in a new script window.

Double-clicking on a ".py" script will run that script, usually closing the window once the script is done running (before you can even see what happened). If you instead want to edit the script in IDLE, you can usually right-click (OS X: control-click) on the file and choose to "Edit with IDLE" to open the script.

Linux users: Read this overview first (especially section 2.2.2) if you want to be able to run Python scripts outside of the editor.

> **NOTE**: You might see something like the following line in the interactive window when you run or re-run a script: >>> ====================== RESTART ======================
>
> This is just IDLE's way of letting you know that everything after this line is the result of the new script that you are just about to run. Otherwise, if you ran one script after another (or one script again after itself), it might not be clear what output belongs to which run of which script.

Screw Things Up

Everybody makes mistakes - especially while programming. In case you haven't made any mistakes yet, let's get a head start on that and mess something up on purpose to see what happens.

Using IDLE, there are two main types of errors you'll experience. The most common is a syntax error, which usually means that you've typed something incorrectly.

Let's try changing the contents of the script to:

```
print("Hello, world)
```

Here we've just removed the ending quotation mark, which is of course a mistake - now Python won't be able to tell where the string of text ends. Save your script and try running it. What happens?

...You can't run it! IDLE is smart enough to realize there's an error in your code, and it stops you from even trying to run the buggy program. In this case, it says: "EOL while scanning string literal." EOL stands for "End Of Line", meaning that Python got all the way to the end of the line and never found the end of your string of text.

IDLE even highlights the place where the error occurred using a different color and moves your cursor to the location of the error. Handy!

The other sort of error that you'll experience is the type that IDLE can't catch for you until your code is already running. Try changing the code in your script to:

```
print(Hello, world)
```

Now that we've entirely removed the quotation marks from the phrase. Notice how the text changes color when we do that? IDLE is letting us know that this is no longer a string of text that we will be printing, but something else. What is our code doing now? Well, save the script and try to run it ...

The interactive window will pop up with ugly red text that looks something like this:

```
>>>

Traceback (most recent call last):

File "[path to your script]\hello world.py", line 1, in <module>

print Hello, world

NameError: name 'Hello' is not defined

>>>
```

So what happened? Python is telling us a few things:

- An error occurred - specifically, Python calls it a `NameError`
- The error happened on line 1 of the script
- The line that generated the error was: print Hello, world
- The specific error was: name 'Hello' is not defined

This is called a run-time error since it only occurs once the programming is already running. Since we didn't put quotes around Hello, world, Python didn't know that this was text we wanted to print. Instead, it thought we were referring to two variables that we wanted to print. The first variable it tried to print was something named "Hello" - but since we hadn't defined a variable named "Hello", our program crashed.

Review exercises:

1. Write a script that IDLE won't let you run because it has a syntax error
2. Write a script that will only crash your program once it is already running because it has a run-time error

Store a Variable

Let's try writing a different version of the previous script. Here we'll use a variable to store our text before printing the text to the screen:

```
phrase = "Hello, world"

print(phrase)
```

Notice the difference in where the quotation marks go from our previous script. We are creating a variable named phrase and assigning it the value of the string of text "Hello, world". We then print the phrase to the screen. Try saving this script and running these two lines; you should see the same output as before:

```
>>>

Hello, world

>>>
```

Notice that in our script we didn't say:

```
print("phrase")
```

Using quotes here would just print the word "phrase" instead of printing the contents of the variable named phrase.

> **NOTE**: Phrases that appear in quotation marks are called strings. We call them strings because they're just that - strings of characters. A string is one of the most basic building blocks of any programming language, and we'll use strings a lot over the next few chapters.

We also didn't type our code like this:

```
Phrase = "Hello, world"

print(phrase)
```

Can you spot the difference? In this example, the first line defines the variable Phrase with a capital "P" at the beginning, but the second line prints out the variable `phrase`.

> **WARNING**: Since Python is case-sensitive, the variables `Phrase` and `phrase` are entirely different things. Likewise, commands start with lowercase letters; we can tell Python to print, but it wouldn't know how to Print. Keep this important distinction in mind!

When you run into trouble with the sample code, be sure to double-check that every character in your code (often including spaces) exactly matches the examples. Computers don't have any common sense to interpret what you meant to say, so being almost correct still won't get a computer to do the right thing!

Review exercises:

1. Using the interactive window, display some text on the screen by using print
2. Using the interactive window, display a string of text by saving the string to a variable, then printing the contents of that variable
3. Do each of the first two exercises again by first saving your code in a script and then running the script

Interlude: Leave yourself helpful notes

As you start to write more complicated scripts, you'll start to find yourself going back to parts of your code after you've written them and thinking, "What the heck was that supposed to do"?

To avoid these moments, you can leave yourself notes in your code; they don't affect the way the script runs at all, but they help to document what's supposed to be happening. These notes are referred to as comments, and in Python you start a comment with a pound (`#`) sign. Our first script could have looked like this:

```
# This is my first script

phrase = "Hello, world."

print(phrase)  # this line displays "Hello, world"
```

The first line doesn't do anything, because it starts with a `#` . This tells Python to ignore the line completely because it's just a note for you.

Likewise, Python ignores the comment on the last line; it will still print phrase, but everything starting with the `#` is simply a comment.

Of course, you can still use a # symbol inside of a string. For instance, Python won't mistake the following for the start of a comment:

```
print("#1")
```

If you have a lot to say, you can also create comments that span over multiple lines by using a series of three single quotes (`'''`) or three double quotes (`"""`) without any spaces between them. Once you do that, everything after the `'''` or `"""` becomes a comment until you close the comment with a matching `'''` or `"""` . For instance, if you were feeling excessively verbose, our first script could have looked like this:

```
"""
This is my first script.

It prints the phrase "Hello, world."

The comments are longer than the script
"""

phrase = "Hello, world."

print(phrase)

"""
The line above displays "Hello, world"
"""
```

The first three lines are now all one comment, since they fall between pairs of `"""`. You can't add a multi-line comment at the end of a line of code like with the # version, which is why the last comment is on its own separate line. (We'll see why in the next chapter.)

Besides leaving yourself notes, another common use of comments is to "comment out code" while you're testing parts of a scripts to temporarily stop that part of the code from running. In other words, adding a `#` at the beginning of a line of code is an easy way to make sure that you don't actually use that line, even though you might want to keep it and use it later.

Fundamentals: Strings and Methods

As we've already seen, you write strings in Python by surrounding them with quotes. You can use single quotes or double quotes, as long as you're consistent for any one string. All of the following lines create string variables (called string literals because we've literally written out exactly how they look):

1. `phrase = 'Hello, world.'`
2. `my_string = "We're #1!"`
3. `string_number = "1234"`
4. `conversation = 'I said, "Put it over by the llama."'`

Strings can include any characters - letters, numbers and symbols. You can see the benefit of using either single or double quotes in the last string example; since we used single quotes, we didn't have any trouble putting double quotes inside the string. (There are other ways to do this, but we'll get to those later in the chapter.)

We can also create really long strings that take up multiple lines by using three single quotes (or three double quotes), like this:

```
long_string = '''This is a
string that spans across multiple lines'''

long_string = """This is a new string
that spans across two lines"""
```

Here we assigned one value to the variable `long_string`, then we overwrote that value with a new string literal. Try putting this in a script and then print the variable `long_string`; you'll see that it displays the string on two separate lines. You can also see now why you can't have multi-line comments appear on the same line as actual code; Python wouldn't be able to tell the difference between these and actual string variables!

It's also worth noting that you can preserve whitespace if you use triple quotes:

```
print("""this is a
    string that spans across multiple lines
        that also preserves whitepace.""")
```

One last thing about strings: If you want to write out a really long string, but you don't want it to appear on multiple lines, you can use a backslash like this when writing it out:

```
my_long_string = "Here's a string that I want to \
write across multiple lines since it is long."
```

Normally Python would get to the end of the first line and get angry with you because you hadn't closed the string with a matching single quote. But because there's a backslash at the end, you can just keep writing the same string on the next line. This is different from the last example since the actual string isn't stored on multiple lines this time, therefore the string gets displayed on a single line without the break:

```
>>> print(my_long_string)

Here's a string that I want to write across multiple lines since it is long.

>>>
```

> **WARNING**: As we've already discussed, Python is case-sensitive. By convention, Python's built-in functions and methods use exclusively lower-case. Since a single variable name can't include any spaces or dashes, when programmers want to give descriptive names to variables, one way of making them easily readable is to use camelCase (i.e., myLongString), so called because of the upper-case "humps" in the middle of terms. Another popular method (especially in Python), and the one we'll be sticking to in this course, is to separate words using underscores (i.e., my_long_string).

Review exercises:

1. Print a string that uses double quotation marks inside the string
2. Print a string that uses an apostrophe (single quote) inside the string
3. Print a string that spans across multiple lines
4. Print a one-line string that you have written out on multiple lines

Mess Around with Your Words

Python has some built-in functionality that we can use to modify our strings or get more information about them. For instance, there's a "length" function (abbreviated as "len" in Python) that can tell you the length of all sorts of things, including strings. Try typing these lines into the interactive window:

```
>>> my_string = "abc"
>>> string_length = len(my_string)
>>> print(string_length)
3
>>>
```

First we created a string named `my_string`. Then we used the `len()` function on `my_string` to calculate its length, which we store in the new variable we named `string_length`. We have to give the `len()` function some input for its calculation, which we do by placing `my_string` after it in the parentheses - you'll see more on exactly how this works later. The length of 'abc' is just the total number of characters in it, 3, which we then print to the screen.

We can combine strings together as well:

```
>>> string1 = "abra"
>>> string2 = "cadabra"
>>> magic_string = string1 + string2
>>> print(magic_string)
abracadabra
>>>
```

Or even like this, without creating any new variables:

```
>>> print("abra"+"ca"+"dabra")
abracadabra
>>>
```

In programming, when we add (or smoosh) strings together like this, we say that we **concatenate** them.

> **NOTE**: You'll see a lot of bold terms throughout the first few chapters of this book. Don't worry about memorizing all of them if they're unfamiliar! You don't need any fancy jargon to program well, but it's good to be aware of the correct terminology. Programmers tend to throw around technical terms a lot; not only does it allow for more precise communication, but it helps make simple concepts sound more impressive.

When we want to combine many strings at once, we can also use commas to separate them. This will automatically add spaces between the strings, like so:

```
>>> print("abra", "ca", "dabra")
abra ca dabra
>>>
```

Of course, the commas have to go outside of the quotation marks, since otherwise the commas would become part of the actual strings themselves.

Since a string is just a sequence of characters, we should be able to access each character individually as well. We can do this by using square brackets after the string, like this:

```
>>> flavor = "birthday cake"
>>> print(flavor[3])
t
>>>
```

Wait, but "t" is the fourth character! Well, not in the programming world. In Python (and most other programming languages), we start counting at 0. So in this case, "b" is the "zeroth" character of the string "birthday cake". This makes "i" the first character, "r" the second, and "t" the third.

If we wanted to display what we would normally tend to think of as the "first" character, we would actually need to print the 0th character:

```
>>> print(flavor[0])
b
>>>
```

Be careful when you're using:

- parentheses: `( )`
- square brackets: `[ ]`

- curly braces: `{ }`

These all mean different things to Python, so you can never switch one for another. We'll see more examples of when each one is used (and we haven't seen `{}` yet), but keep in mind that they're all used differently.

The number that we assigned to each character's position is called the index or subscript number, and Python thinks of the string like this:

Character:	**b**	**i**	**r**	**t**	**h**	**d**	**a**	**y**
Index / Subscript #:	0	1	2	3	4	5	6	7

We can get a particular section out of the string as well, by using square brackets and specifying the range of characters that we want. We do this by putting a colon between the two subscript numbers, like so:

```
>>> flavor = "birthday cake"
>>> print(flavor[0:3])
bir
>>>
```

Here we told Python to show us only the first three characters of our string, starting at the 0th character and going up until (but not including) the 3rd character. The number before the colon tells Python the first character we want to include, while the number after the colon says that we want to stop just before that character.

If we use the colon in the brackets but omit one of the numbers in a range, Python will assume that we meant to go all the way to the end of the string in that direction:

```
>>> flavor = "birthday cake"
>>> print(flavor[:5])
birth
>>> print(flavor[5:])
day cake
>>> print(flavor[:])
birthday cake
>>>
```

The way we're using brackets after the string is referred to as subscripting or indexing since it uses the index numbers of the string's characters.

> **NOTE**: Python strings are immutable, meaning that they can't be changed once you've created them. For instance, see what happens when you try to assign a new letter to one particular character of a string:
>
> `my_string = "goal"`
>
> `my_string[0] = "f" # this won't work!`
>
> Instead, we would have to create an entirely new string (although we can still give `my_string` that new value):
>
> `my_string = "goal"`
>
> `my_string = "f" + my_string[1:]`
>
> In the first example, we were trying to change part of `my_string` and keep the rest of it unchanged, which doesn't work. In the second example, we created a new string by adding two strings together, one of which was a part of `my_string`; then we took that new string and completely reassigned `my_string` to this new value.

Review exercises:

1. Create a string and print its length using the `len()` function
2. Create two strings, concatenate them (add them next to each other) and print the combination of the two strings
3. Create two string variables, then print one of them after the other (with a space added in between) using a comma in your print statement
4. print the string "zing" by using subscripting and index numbers on the string "bazinga" to specify the correct range of characters

Use Objects and Methods

The Python programming language is an example of Object-Oriented Programming (OOP), which means that we store our information in objects. In Python, a string is an example of an object. Strings are very simple objects - they only hold one piece of information (their value) - but a single object can be very complex. Objects can even hold other objects inside of them. This helps to give structure and organization to our programming.

For instance, if we wanted to model a car, we would (hypothetically) create a Car object that holds lots of descriptive information about the car, called its attributes. It would have a color attribute, a model attribute, etc., and each of these attributes would hold one piece of descriptive information about the car. It would also include different objects like Tires, Doors, and an Engine that all have their own attributes as well.

Different objects also have different capabilities, called methods. For instance, our Car object might have a `drive()` method and a `park()` method. Since these methods belong to the car, we use them with "dot notation" by putting them next to the object and after a period, like this:

`car.park()`

Methods are followed by parentheses, because sometimes methods use input. For instance, if we wanted to drive the car object a distance of 50, we would place that input of 50 in the parentheses of the "drive" method:

`car.drive(50)`

There are certain methods that belong to string objects as well. For instance, there is a string method called `upper()` that creates an upper-case version of the string. (Likewise, there is a corresponding method `lower()` that creates a lower-case version of a string.) Let's give it a try in the interactive window:

```
>>> loud_voice = "Can you hear me yet?"

>>> print(loud_voice.upper())

CAN YOU HEAR ME YET?

>>>
```

We created a string `loud_voice`, then we called its `upper()` method to return the upper-case version of the string, which we print to the screen.

> **NOTE**: Methods are just functions that belong to objects. We already saw an example of a general-purpose function, the `len()` function, which can be used to tell us the length of many different types of objects, including strings. This is why we use the length function differently, by only saying: `len(loud_voice)`
>
> Meanwhile, we use dot notation to call methods that belong to an object, like when we call the `upper()` method that belongs to the string `loud_voice`: `loud_voice.upper()`

Let's make things more interactive by introducing one more general function. We're going to get some input from the user of our program by using the function `input()`. The input that we pass to this function is the text that we want it to display as a prompt; what the function actually does is to receive additional input from the user. Try running the following script:

```
user_input = input("Hey, what's up? ")

print("You said: ", user_input)
```

When you run this, instead of the program ending and taking you back to the `>>>` prompt, you'll just see:

```
>>>

Hey, what's up?
```

...with a blinking cursor. It's waiting for you to answer! Enter a response, and it will store that answer in the `user_input` string and display it back:

```
>>>

Hey, what's up? Mind your own business.

You said: Mind your own business.

>>>
```

Now we'll combine the function `input()` with the string method `upper()` in a script to modify the user's input:

```
response = input("What should I shout? ")

response = response.upper()

print("Well, if you insist...", response)
```

Calling `response.upper()` didn't change anything about our response string. The `upper()` method only returned the upper-case version of the string to us, and it was up to us to do something with it. That's why we had to set `response = response.upper()` in order to reassign the value of the string response to its own upper-case equivalent.

In IDLE, if you want to see all the methods can apply to a particular kind of object, you can type that object out followed by a period and then hit CTRL+SPACE. For instance, first define a string object in the interactive window:

```
>>> my_string = "kerfuffle"
```

Now type the name of your string in again, followed by a period (without hitting enter):

```
>>> my_string.
```

When you hit CTRL+SPACE, you'll see a list of method options that you can scroll through with the arrow keys. Strings have lots of methods!

A related shortcut in IDLE is the ability to fill in text automatically without having to type in long names by hitting TAB. For instance, if you only type in "my_string.u" and then hit the TAB key, IDLE will automatically fill in "my_string.upper" because there is only one method belonging to my_string that begins with a "u". In fact, this even works with variable names; try typing in just the first few letters of "my_string" and, assuming you don't have any other names already defined that share those first letters, IDLE will automatically complete the name "my_string" for you when you hit the TAB key.

For more on objects and classes, check out the *Primer on Object-Oriented Programming in Python* chapter.

Review exercises:

1. Write a script that takes input from the user and displays that input back
2. Use CTRL+SPACE to view all the methods of a string object, then write a script that returns the lower-case version of a string

Assignment: Pick apart your user's input

Write a script named *first_letter.py* that first prompts the user for input by using the string: `Tell me your password:`

The script should then determine the first letter of the user's input, convert that letter to upper-case, and display it back. As an example, if the user input was "no" then the program should respond like this: The first letter you entered was: N

For now, it's okay if your program crashes when the user enters nothing as input (just hitting ENTER instead). We'll find out a couple ways you could deal with this situation in an upcoming chapter.

Fundamentals: Working with Strings

We've seen that string objects can hold any characters, including numbers. However, don't confuse string "numbers" with actual numbers. For instance, try this bit of code out in the interactive window:

```
>>> my_number = "2"
>>> print(my_number + my_number)
22
>>>
```

We can add strings together, but we're just concatenating them - we're not actually adding the two quantities together. Python will even let us "multiply" strings as well:

```
>>> my_number = "12"
>>> print(my_number * 3)
121212
>>>
```

If we want to change a string object into a number, there are two general functions we commonly use: `int()` and `float()`.

`int()` stands for "integer" and converts objects into whole numbers, while `float()` stands for "floating-point number" and converts objects into numbers that have decimal points. For instance, we could change the string my_number into an integer or a "float" like so:

```
>>> my_number = "12"
>>> print(int(my_number))
12
>>> print(float(my_number))
12.0
>>>
```

Notice how the second version added a decimal point, because the floating-point number has more precision (more decimal places). For this reason, we couldn't change a string that looks like a floating-point number into an integer because we would have to lose everything after the decimal:

```
>>> my_number = "12.0"

>>> print(int(my_number))
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(int(my_number))
ValueError: invalid literal for int() with base 10: '12.0'
>>>
```

Even though the extra 0 after the decimal place doesn't actually add any value to our number, Python is telling us that we can't just change 12.0 into 12 - because we might lose part of the number.

If you want to turn a number into a string, of course there's a function for that, too - the `str()` function. One place where this becomes important is when we want to add string and numbers together. For instance, we can't just concatenate the two different types of objects like this:

```
>>> print("1" + 1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print("1" + 1)
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Python doesn't know how to add different types of objects together - we could have meant the answer to be "2" or "11" depending on whether we had strings or integers. If we wanted to put the two numbers side-by-side as one string, we would have to convert the integer into a string:

```
>>> print("1" + str(1))
11
>>>
```

"Integer" and "string" are called **types** of objects. Always keep in mind that you might get unexpected results if you mix types incorrectly or if you use one type of object when you really meant to use another.

Review exercises:

1. Create a string object that stores an integer as its value, then convert that string into an actual integer object using `int()` ; test that your new object is really a number

by multiplying it by another number and displaying the result

2. Repeat the previous exercise, but use a floating-point number and `float()`
3. Create a string object and an integer object, then display them side-by-side with a single print statement by using the `str()` function

Streamline Your Print Statements

Suppose we have a string object, `name = "Zaphod"`, and two integer objects, `num_heads = 2` and `num_arms = 3`. We want to display them in the following line: `Zaphod has 2 heads and 3 arms`. This is called **string interpolation**, which is just a fancy way of saying that you want to insert some 'stuff' - i.e., variables - into a string.

We've already seen two ways of doing this. The first would involve using commas to insert spaces between each piece of our statement:

```
print(name, "has", str(num_heads), "heads and", str(num_arms), "arms")
```

Another way we could do this is by concatenating the strings with the + operator:

```
print(name+" has "+str(num_heads)+" heads and "+str(num_arms)+" arms")
```

I didn't use spaces around all the + signs just so that the two expressions would line up, but it's pretty difficult to read either way. Trying to keep track of what goes inside or outside of the quotes can be a huge pain, which is why there's a third way of combining strings together: using the string `format()` method.

The simplest version of the `format()` method would look like this for our example:

```
print("{} has {} heads and {} arms".format(name, num_heads, num_arms))
```

The pairs of empty curly braces (`{}` without any space in between the two) serve as placeholders for the variables that we want to place inside the string. We then pass these variables into our string as inputs of the string's `format()` method, in order. The really great part about this technique is that we didn't even have to change our integers into string types first - the `format()` method did that for us automatically.

Although it's less frequently used, we can also use index numbers inside the curly braces to do the same thing:

```
print("{0} has {1} heads and {2} arms".format(name, num_heads, num_arms))
```

Here we've inserted name into the {0} placeholder because it is the 0th input listed, and so on. Since we numbered our placeholders, we don't even have to provide the inputs in the same order. For instance, this line would also do the exact same thing:

```
print("{2} has {1} heads and {0} arms".format(num_arms, num_heads, name))
```

This style of formatting can be helpful if you want to repeat an input multiple times within a string, i.e.:

```
>>> print("{0} has {0} heads and {0} arms".format(name))
Zaphod has Zaphod heads and Zaphod arms.
>>>
```

If we didn't want to create three separate objects ahead of time, one last way of using `format()` would be to name and assign new objects inside the `format()` method, like so:

```
print("{name} has {num_heads} heads and {num_arms} arms".format(
    name="Zaphod", num_heads=2, num_arms=3
))
```

These input variables don't necessarily have to be listed in the same order since we've called on each of them by name inside the string.

Finally, in Python 3.6 there is a new type of string called a formatted string literal, which can use previously defined variables or other pieces of code placed inside curly braces so long as we put a `f` right before the string, like this:

```
>>> name = "formatted"
>>> print(f"This is a {name} string")
This is a formatted string
```

This syntax combines the benefits of the last couple examples, being both less to type and easier to read when the values themselves appear in place. These formatted strings can use any number of `{}` placeholders to refer to variables or other expressions; take a look at this example, which uses two variables and even includes a calculation inside the string:

```
name = "Zaphod"
num_heads = 2
print(f"{name} has {num_heads} heads and {num_heads+1} arms")
```

> **NOTE**: There is also another way to print formatted strings: using the % operator. You might see this in code that you find elsewhere, and you can read about how it works here if you're curious, but just be aware that this style has been phased out completely in Python 3 (and the "new" `format()` style also works in Python 2.7), so there's no need to use this method in your code. Just be aware of it in case you come across it in legacy code bases.

Review exercises:

1. Create a "float" object (a decimal number) named `weight` that holds the value 0.2, and create a string object named `animal` that holds the value "newt", then use these objects to print the following line without using the `format()` string method: 0.2 kg is the weight of the newt.
2. Display the same line using `format()` and empty `{}` place-holders
3. Display the same line using `{}` place-holders that use the index numbers of the inputs provided to the `format()` method
4. Display the same line by creating new string and float objects inside of the `format()` method
5. Display the same line by creating a formatted string literal that directly uses the `weight` and `animal` variables

Find a String in a String

One of the most useful string methods is `find()`. As its name implies, we can use this method to find the location of one string in another string. We use dot notation because this method belongs to a string, and the input we supply in parentheses is the string we're searching for:

```
>>> phrase = "the surprise is in here somewhere"
>>> print(phrase.find("surprise"))
4
>>>
```

We're searching for the location of the string "surprise" in our phrase string. The value that `find()` returns is the index of the first occurrence of that string. In this case, "surprise" starts at the 4th character into the phrase (remember to start counting at 0), so we displayed 4.

If `find()` doesn't find the string we're looking for, it will return -1 instead:

```
>>> phrase = "the surprise is in here somewhere"
>>> print(phrase.find("ejafjallajökull"))
-1
>>>
```

We can even call string methods on a string literal directly, so in this case we didn't even need to create a new string object:

```
>>> print("the surprise is in here somewhere".find("surprise"))
4
>>>
```

Keep in mind that this matching is done exactly, character by character. If we had tried to find "SURPRISE", we would have gotten a -1.

The part of the string we are searching for (or any part of a string) is called a substring.

If a substring appears more than once in our string, `find()` will just return the first appearance, starting from the beginning of the string. For instance, try out:

```
>>> "I put a string in your string".find("string")
8
>>>
```

Keep in mind that we still can't mix object types; `find()` will only accept a string as its input. If we were looking for an integer inside in a string, we would still have to put that integer value in a string of its own:

```
>>> "My number is 555-555-5555".find("5")
13
>>>
```

A similar string method is `replace()`, which will replace all occurrences of one substring with a different string. For instance, let's replace every instance of "the truth" with the string "lies" in the following:

```
>>> my_story = "I'm telling you the truth; he spoke nothing but the truth!"
>>> print(my_story.replace("the truth", "lies"))
I'm telling you lies; he spoke nothing but lies!
>>>
```

Keep in mind that calling `replace()` did not actually change my_story; in order to affect this string, we would still have to reassign it to a new value, as in:

```
>>> my_story = my_story.replace("the truth", "lies")
>>>
```

Review exercises:

1. In one line, display the result of trying to `find()` the substring "a" in the string "AAA"; the result should be -1
2. Create a string object that contains the value "version 2.0"; `find()` the first occurrence of the number 2.0 inside of this string by first creating a "float" object that stores the value 2.0 as a floating-point number, then converting that object to a string using the `str()` function
3. Write and test a script that accepts user input using `input()`, then displays the result of trying to `find()` a particular letter in that input

Assignment: Turn your user into a l33t h4x0r

Write a script *translate.py* that asks the user for some input with the following prompt: `Enter some text:`

You should then use the `replace()` method to convert the text entered by the user into "leetspeak" by making the following changes to lower-case letters:

- The letter: a becomes: 4
- The letter: b becomes: 8
- The letter: e becomes: 3
- The letter: l becomes: 1
- The letter: o becomes: 0
- The letter: s becomes: 5
- The letter: t becomes: 7

Your program should then display the resulting output. A sample run of the program, with the user input in bold, is shown below:

```
>>> Enter some text: I like to eat eggs and spam.

I 1ik3 70 347 3gg5 4nd 5p4m.

>>>
```

Fundamentals: Functions and Loops

We already did some basic math using IDLE's interactive window. For instance, we saw that we could evaluate simple expressions just by typing them in at the prompt, which would display the answer:

```
>>> 6 * (1 + 6)
42
>>>
```

However, just putting that line into a script-

```
6 * (1 + 6)
```

-would be useless since we haven't actually told the program to do anything. If we want to display the result from a program, we have to rely on the print command again.

Go ahead and open a new script, save it as *arithmetic.py* and try displaying the results of some basic calculations:

```
print("1 + 1 =", 1 + 1)
print("2 * (2 + 3) =", 2 * (2 + 3))
print("1.2 / 0.3 =", 1.2 / 0.3)
print("5 / 2 =", 5 / 2)
```

Here we've used a single print statement on each line to combined two pieces of information by separating the values with a comma. The results of the numerical expressions on the right will automatically be calculated when we display it.

> **NOTE**: All of the spaces we included above were entirely optional, but they help to makes things easier to read.

When you save and run this script, it displays the results of your print commands as follows:

```
>>>
1 + 1 = 2
2 * (2 + 3) = 10
1.2 / 0.3 = 4.0
5 / 2 = 2.5
>>>
```

Assignment: Perform calculations on user input

Write a script called *exponent.py* that receives two numbers from the user and displays the result of taking the first number to the power of the second number. A sample run of the program should look like this (with example input that has been provided by the user included below):

```
>>>
Enter a base: 1.2
Enter an exponent: 3
1.2 to the power of 3 = 1.728
>>>
```

Keep the following in mind:

1. In Python, x^y (x raised to the power y) is calculated by using the expression `x ** y`
2. Before you can do anything with the user's input, you will have to store the results of both calls to `input()` in new objects
3. The `input()` function returns a string object, so you will need to convert the user's input into numbers in order to do arithmetic on them
4. You should use the string `format()` method to print the result
5. You can assume that the user will enter actual numbers as input

Create Your Own Functions

One of the main benefits of programming in Python is the ease with which different parts and pieces of code can be put together in new ways. Think of it like building with Lego bricks instead of having to craft everything by hand each time you start a project.

The Lego brick of programming is called a **function**. A function is basically a miniature program; it accepts input and produces output. We've already seen some examples of functions such as the `find()` string method - when called on a string, it takes some input and returns the location of that input within the string as its output.

> **NOTE**: Functions are like the functions from a math class: You provide the input and the function produces output.

We could create our own function that takes a number as its input and produces the square of that number as its output. In Python, this would look like:

```
def square(number):
    sqr_num = number ** 2
    return sqr_num
```

The `def` is short for "define" and lets Python know that we are about to define a new function. In this case, we called the function square and gave it one input variable (the part in parentheses) named number. A function's input (or, the value passed to a function) is called an **argument** of the function, and a function can take more than one argument.

The first line within our function multiplies number by itself and stores the result in a new variable named sqr_num. Then the last line of our function returns the value of sqr_num, which is the output of our function.

If you just type these three lines into a script, save it and run it, nothing will happen. The function doesn't do anything by itself.

However, now we can use the function later on from the main section of the script. For instance, try running this script:

```
def square(number):
    sqr_num = number ** 2
    return sqr_num

input_num = 5
output_num = square(input_num)

print(output_num)
```

By saying `output_num = square(input_num)`, we are calling up the function square and providing this function with the input variable `input_num`, which in this case has a value of 5. Our function then calculates 25 and returns the value of the variable `sqr_num`, which gets stored in our new variable `output_num`.

> **NOTE**: Notice the colon and the indentation after we defined our function. These aren't optional. This is how Python knows that we are still inside of the function. As soon as Python sees a line that isn't indented, that's the end of the function. Every line inside the function must be indented.

You can define many functions in one script, and functions can even refer to each other. However, it's important that a function has been defined before you try to use it. For instance, try running this code instead:

```
input_num = 5

output_num = square(input_num)

print(output_num)

def square(number):
    sqr_num = number * number
    return sqr_num
```

Here we've just reordered the two parts of our script so that the main section comes before the function. The problem here is that Python runs through our code from the top to the bottom - so when we call the square function on the second line, Python has no idea what we mean yet because we don't actually define the square function until later on in the script, and it hasn't gotten there yet. Instead we see an error:

```
NameError: name 'square' is not defined
```

To create a function that uses more than one input, all we need to do is separate each argument of the function with a comma. For instance, the following function takes two arguments as input and returns the difference, subtracting the second number from the first:

```
def return_difference(num1, num2):
    return num1 - num2
```

To call this function, we need to supply it with two inputs:

```
print(return_difference(3, 5))
```

This line will call our new `return_difference()` function, then display the result of -2 that the function returns.

> **NOTE**: Once a function returns a value with the return command, the function is done running; if any code appears inside the function after the return statement, it will never be run because the function has already returned its final result.

One last helpful thing about functions is that Python allows you to add special comments called **docstrings**. A docstring serves as documentation, helping to explain what a function does and how to use it. They're completely optional, but can be helpful if there's any chance that you'll either share your code with someone else or if you ever come back to your code later, once you've forgotten what it's supposed to do - which is why you should leave comments in the first place. A docstring looks just like a multi-line comment with three quotation marks, but it has to come at the very beginning of a function, right after the first definition line:

```
def return_difference(n1, n2):
    """Return the difference between two numbers.
       Subtracts n2 from n1."""
    return n1 - n2
```

The benefit of this (besides leaving a helpful comment in the code) is that we can now use the `help()` function to get information about this function. Assuming we defined this function by typing it in the interactive window or we already ran a script where it was defined, we can now type `help(return_difference)` and see:

```
>>> help(return_difference)
Help on function return_difference in module __main__:
return_difference(n1, n2)
    Return the difference between two numbers.
    Subtracts n2 from n1.
>>>
```

Of course, you can also call `help()` on the many other Python functions we'll see to get a quick reference on how they are used.

Functions Summary

So, what do we know about functions?

1. Functions require a function signature.
2. They do something useful.
3. They allow us re-use code without having to type each line out.
4. They can take an input and usually produce some output.
5. You call a function by using its name followed by empty parenthesis or its arguments in parenthesis.

Functions require a function signature.

Function signatures tell the user how the function should be called. They start with the `def` keyword, indicating to the Python interpreter that we're defining a function. Next comes a space along with the the name of the function, and then an open and closed parenthesis. If the function takes any inputs, we define these inside the parenthesis and separate each one with a comma, except the last. We call these parameters.

The names of parameters should describe what they are used for, while the names of functions should be descriptive enough for the user to understand what it does. Best practice: a function name should be a verb or an action and arguments names should be nouns.

For example:

```
def add_two_numbers(num1, num2):
```

They do something useful.

Within the function itself something useful should happen, otherwise you have to question, "Why does the function exist in the first place?"

In our `add_two_numbers()` function, we could, as the name describes, add two numbers:

```
def add_two_numbers(num1, num2):
    sum = num1 + num2
```

They allow us re-use code without having to type each line out.

We can use this function as many times as we want, passing in two numbers, which will then get combined.

They take an input and usually produce some output.

In the above function, we pass in two numbers as input parameters, but we don't return anything. Therefore, we don't get any output. Let's fix that by returning the sum:

```
def add_two_numbers(num1, num2):
    sum = num1 + num2
    return sum
```

You call a function by using its name followed by its arguments in parenthesis.

We can call this function as many times as we'd like, providing different arguments each time to produce new output:

```
>>> def add_two_numbers(num1, num2):
...     sum = num1 + num2
...     return sum
...
>>> print(add_two_numbers(1,1))
2
>>> print(add_two_numbers(1,2))
3
>>> print(add_two_numbers(-20,9))
-11
>>>
```

Try creating your own functions that do something useful following these steps.

Review exercise:

1. Write a `cube()` function that takes a number and multiplies that number by itself twice over, returning the new value; test the function by displaying the result of calling your `cube()` function on a few different numbers
2. Write a function `multiply()` that takes two numbers as inputs and multiplies them

together, returning the result; test your function by saving the result of `multiply(2, 5)` in a new variable and printing it

Assignment: Convert temperatures

Write a script *temperature.py* that includes two functions. One function takes a Celsius temperature as its input, converts that number into the equivalent Fahrenheit temperature and returns that value. The second function takes a Fahrenheit temperature and returns the equivalent Celsius temperature. Test your functions by passing input values to them and printing the output results.

For testing your functions, example output should look like:

```
72 degrees F = 22.2222222222 degrees C
37 degrees C = 98.6 degrees F
```

In case you didn't want to spend a minute searching the web or doing algebra (the horror!), the relevant conversion formulas are:

1. F = C * 9/5 + 32
2. C = (F - 32) * 5/9

Run in circles

One major benefit of computers is that we can make them do the same exact thing over and over again, and they rarely complain or get tired. The easiest way to program your computer to repeat itself is with a loop.

There are two kinds of loops in Python: **for loops** and **while loops**. The basic idea behind any kind of loop is to run a section of code repeatedly as long as a specific statement (called the test condition) is true. For instance, try running this script that uses a while loop:

```
n = 1
while (n < 5):
    print("n =", n)
    n = n + 1
print("Loop finished")
```

Here we create a variable `n` and assign it a value of 1. Then we start the `while` loop, which is organized in a similar way to how we defined a function. The statement that we are testing comes in parentheses after the `while` command; in this case we want to know if the statement `n < 5` is `true` or `false`. Since `1 < 5`, the statement is `true` and we enter into the loop after the colon.

> **NOTE**: Notice the indentation on the lines after the colon. Just like when we defined a function, this spacing is important The colon and indenting let Python know that the next lines are inside the while loop. As soon as Python sees a line that isn't indented, that line and the rest of the code after it will be considered outside of the loop.

Once we've entered the loop, we print the value of the variable `n`, then we add 1 to its value. Now `n` is 2, and we go back to test our `while` statement. This is still true, since `2 < 5`, so we run through the next two lines of code again… And we keep on with this pattern while the statement `n < 5` is `true`.

As soon as this statement becomes `false`, we're completely done with the loop; we jump straight to the end and continue on with the rest of the script, in this case printing out "Loop finished "

Go ahead and test out different variations of this code and try to guess what your output will be before you run each script. Just be careful: it's easy to create what's called an **infinite loop**; if you test a statement that's always going to be true, you will never break out of the loop, and your code will just keep running forever.

> **NOTE**: It's important to be consistent with indentation, too. Notice how you can hit tab and backspace to change indentation, and IDLE automatically inserts four space characters. That's because you can't mix tabs and spaces as indentation. Although IDLE won't let you make this mistake, if you were to open your script in a different text editor and replace some of the space indentation with tabs, Python would get confused - even though the spacing looks the same to you.

The second type of loop, a `for` loop, is slightly different in Python. We typically use `for` loops in Python in order to loop over every individual item in a set of similar things - these things could be numbers, variables, lines from an input file, etc.

For instance, the following code does the exact same thing as our previous script by using a `for` loop to repeatedly run code for a range of numbers:

```
for n in range(1, 5):
    print("n =", n)
print("Loop finished")
```

Here we are using the `range()` function, which is a function that is built into Python, to return a list of numbers. The `range()` function takes two input values, a starting number and a stopping number. So in the first line, we create a variable `n` equal to 1, then we run through the code inside of our loop for `n = 1`.

We then run through the loop again for `n = 2`, etc., all the way through our range of numbers. Once we get to the end of the range, we jump out of the loop. Yes, it's slightly counter-intuitive, but in Python a `range(x, y)` starts at x but ends right before y.

> **NOTE**: When we use a `for` loop, we don't need to define the variable we'll be looping over first. This is because the for loop reassigns a new value to the variable each time we run through the loop. In the above example, we assigned the value 1 to the object `n` as soon as we started the loop. This is different from a while loop, where the first thing we do is to test some condition - which is why we need to have given a value to any variable that appears in the while loop before we enter the loop.

Play around with some variations of this script to make sure you understand how the `for` loop is structured. Again, be sure to use the exact same syntax, including the `in` keyword, the colon, and the proper indentation inside the loop.

Don't just copy and paste the sample code into a script - type it out yourself. Not only will that help you learn the concepts, but the proper indentation won't copy over correctly anyway…

As long as we indent the code correctly, we can even put loops inside loops.

Try this out:

```
for n in range(1, 4):
    for j in ["a", "b", "c"]:
        print("n =", n, "and j =", j)
```

Here `j` is looping over a list of strings; we'll see how lists work in a later chapter, but this example is only to show that you don't have to use the `range()` function with a `for` loop. Since the `j` loop is inside the `n` loop, we run through the entire `j` loop (`j="a", j="b", j="c"`) for each value that gets assigned to `n` in the outer loop.

We will use `for` loops in future chapters as an easy way to loop over all the items stored in many different types of objects - for instance, all the lines in a file.

> **NOTE**: Once your code is running in the interactive window, sometimes you might accidentally end up entering a loop that takes much longer than you expected. If that's the case (or if your code seems "frozen" because of anything else that's taking longer than expected), you can usually break out of the code by typing CTRL+C in the interactive window. This should immediately stop the rest of your script from running and take you back to a prompt in the interactive window.

If that doesn't seem to have any effect (because you somehow managed to freeze the IDLE window), you can usually type CTRL+Q to quit out of IDLE entirely, much like "End Task" in Windows or "Force Quit" on a Mac.

Review exercises:

1. Write a `for` loop that prints out the integers 2 through 10, each on a new line, by using the `range()` function
2. Use a `while` loop that prints out the integers 2 through 10 (Hint: you'll need to create a new integer first; there isn't a good reason to use a while loop instead of a `for` loop in this case, but it's good practice...)

3. Write a function called `doubles()` that takes one number as its input and doubles that number three times using a loop, displaying each result on a separate line; test your function by calling `doubles(2)` to display 4, 8, and 16

Assignment: Track your investments

Write a script *invest.py* that will track the growing amount of an investment over time. This script includes an `invest()` function that takes three inputs: the initial investment amount, the annual compounding rate, and the total number of years to invest. So, the first line of the function will look like this:

```
def invest(amount, rate, time):
```

The function then prints out the amount of the investment for every year of the time period.

In the main body of the script (after defining the function), use the following code to test your function:

```
invest(100, .05, 8)
invest(2000, .025, 5)
```

Running this test code should produce the following output exactly:

```
principal amount: $100
annual rate of return: 0.05
year 1: $105.0
year 2: $110.25
year 3: $115.7625
year 4: $121.550625
year 5: $127.62815625
year 6: $134.009564063
year 7: $140.710042266
year 8: $147.745544379

principal amount: $2000
annual rate of return: 0.025
year 1: $2050.0
year 2: $2101.25
year 3: $2153.78125
year 4: $2207.62578125
year 5: $2262.81642578
```

> **NOTE**: Although functions usually return output values, this is entirely optional in Python. Functions themselves can print output directly as well. You can print as much as you like from inside a function, and the function will continue running until you reach its end.*

Some additional pointers if you're stuck:

1. You will need to use a `for` loop over a range that's based on the function's time input.
2. Within your loop, you will need to reassign a new value to the amount every year based on how it grows at 1 + rate.
3. Remember that you need to use either the string `format()` method or `str()` to convert numbers to strings first if you want to print both strings and numbers in a single statement.
4. Using the `print()` function without passing any arguments will print a blank line.

Interlude: Debug your code

You've probably already discovered how easy it is to make mistakes that IDLE can't automatically catch for you. As your code becomes longer and more complicated, it can become a lot more difficult to track down the sources of these errors.

When we learned about syntax and run-time errors, I actually left out a third, most difficult type of error that you've probably already experienced: the logic error. Logic errors occur when you've written a program that, as far as your computer is concerned, is a completely "valid" program that it has no trouble running - but the program doesn't do what you intended it to do because you made a mistake somewhere.

Programmers use debuggers to help get these bugs out of their programs (we're clever with names like that), and there's already a simple debugger built into IDLE that you should learn to use - before you need to use it.

> **NOTE**: Although debugging is the least glamorous and most boring part of programming, learning to make good use of a debugger can save you a lot of time in the end. We all make mistakes; it's a good idea to learn how to find and fix them.

From the file menu of the interactive window of IDLE (not in a script window), click on Debug -> Debugger to open the Debug Control window.

Notice how the interactive window now shows [DEBUG ON] at the prompt to let you know that the debugger is open. We have a few main options available to us, all of which will be explained shortly: Go, Step, Over, Out, and Quit.

Keep both the debugger window and the interactive window open, but let's also start a new script so that we can see how the debugger works:

```
for i in range(1, 4):
    j = i * 2
    print("i is", i, "and j is", j)
```

If you run this script while you have the debugger open, you'll notice that it doesn't get very far. Actually, it pauses before running anything at all, and the "Stack" window at the top of the debugger says:

```
>>> '__main__'.<module>(), line 1: for i in range(1, 4):
```

All this is telling us is that line 1 (which contains the code `for i in range(1, 4):`) is about to be run. The beginning of the "Stack" line in the debugger refers to the fact that we're currently in the "main" section of the script - for instance, as opposed to being in a function definition before the main block of code has been reached.

The debugger allows us to step through the code line by line, keeping track of all our variables as we go. So, let's go ahead and click once on "Step" in the debugger in order to do that. Watch carefully what happens to the debugger window when you do that…

Now the "Stack" window in the debugger says that we're about to run line 2 of the code, which means that line 1 has been executed. Below that, the "Globals and Locals" window includes a new variable i that's been assigned the value 1. This is because the for loop in the first line of our code created this integer i and gave it that starting value. (There are also a few internal system variables listed above it, but we can ignore those.)

Continue hitting the "Step" button to walk through your code line by line, watching what happens in the debugger window. When you arrive at the print statement, a new window will pop open. This is appearing because we called `print()` . It is the internal Python code being used to print output to the interactive window. Don't worry about this for now.

More importantly, you can track the growing values of i and j as you loop through the for statement, and output is displayed in the interactive window as usual when the print statements are run.

Usually, we only want to debug a particular section of the code, so spending all day clicking the "Step" button is less than ideal. This is the idea behind setting a breakpoint.

Breakpoints tell the debugger when to start pausing your code. They don't actually break anything; they're more like "hang on a second, let me take a look at things first"-points.

Let's learn to use breakpoints by working with the following example code, which isn't quite doing what we want it to do yet:

```python
def add_underscores(word):
    new_word = "_"
    for i in range(0, len(word)):
        new_word = word[i] + "_"
    return new_word

phrase = "hello "
print(add_underscores(phrase))
```

What we meant for the function `add_underscores()` to do was to add underscores around every character in the word passed to it, so that we could give it the input "hello " and it would return the output:

```
_h_e_l_l_o_ _
```

Instead, all we see right now is:

```
 _
```

It might already be obvious to you what our error was, but let's use the debugger to work through the problem. We know that the problem is occurring somewhere inside in the function - specifically, within the for loop, since we said that new*word should start with a "*" but it clearly doesn't. So let's put a breakpoint at the start of the for loop so that we can trace out exactly what's happening inside.

To set a breakpoint on a line, right-click (Mac: control-click or two finger click) on that line and select "Set Breakpoint", which should highlight the line to let you know that the breakpoint is active.

Now we can run the script with the debugger open. It will still pause on the very first line it sees (which is defining the function), but we can select "Go" to run through the code normally. This will save the function in Python's memory, save the variable phrase as "hello ", call up our function from the print statement… and then pause at our breakpoint, right before entering the for loop.

At this point, we see that we have two local variables defined (they're "local" because they belong to the function). As expected, we have new_word, which is a string with just the underscore character, and we have word, the variable we passed to the function, with "hello " as its contents.

Click "Step" once and you'll see that we've entered the for loop. The counter we're using, i, has been given its first value of 0.

Click "Step" one more time, and it might become clearer what's happening in our code. The variable new*word has taken on the value "h*"… It got rid of our first underscore character already If you click "Step" a few more times, you'll see that new*word gets set to "e*", then "l_", etc. We've been overwriting the contents of new_word instead of adding to it, and so we correct the line to:

```
new_word = new_word + word[i] + "_"
```

Of course, the debugger couldn't tell us exactly how to fix the problem, but it helped us identify where the problem occurred and what exactly the unexpected behavior was.

You can remove a breakpoint at any time by right-clicking (Mac: control-click or two finger click) and selecting "Clear Breakpoint".

In the debugger, "Quit" does just that - immediately quits out of the program entirely, regardless of where you were in the script. The option "Over" is sort of a combination of "Step" and "Go" - it will step over a function or loop. In other words, if you're about to "Step" into a function with the debugger, you can still run that function's code without having to "Step" all the way through each line of it - "Over" will take you directly to the end result of running that function. Likewise, if you're already inside of a function, "Out" will take you directly to the end result of that function.

> **NOTE**: When trying to reopen the debugger, you might see this error: `You can only toggle the debugger when idle`
>
> It's most likely because you closed out of the debugger while your script was still running. Always be sure to hit "Go" or "Quit" when you're finished with a debugging session instead of just closing the debugger, or you might have trouble reopening it. The IDLE debugger isn't the most carefully crafted piece of software - sometimes you'll just have to exit out of IDLE and reopen your script to make this error go away.

Debugging can be tricky and time-consuming, but sometimes it's the only reliable way to fix errors that you've overlooked. However, before turning to the debugger, in some cases you can just use print statements to your advantage to figure out your mistakes much more easily.

The easiest way to do this is to print out the contents of variables at specific points throughout your script. If your code breaks at some point, you can also print out the contents of variables using the interactive window to compare how they appear to how you thought they should look at that point.

For instance, in the previous example we could have added the statement "print new_word" inside of our for loop. Then when we run the script, we'd be able to see how new_word actually grows (or in this case, how it doesn't grow properly). Just be careful when using print statements this way, especially inside of loops - if you don't plan out the process well, it's easy to end up displaying thousands of lines that aren't informative and only end up slowing down or freezing your program.

Or you could always try rubber ducking.

Fundamentals: Conditional Logic

Thus far we have focused on writing unconditional code - e.g., there are no choices being made, and the code is always ran. Python has another data type (or primitive) called a boolean that is helpful for writing code that makes decisions, which is the focus of this chapter.

Compare Values

Computers understand our world in binary, breaking every problem down into 0's and 1's. In order to make comparisons between values, we have to learn how to communicate in this "1 or 0" language using boolean logic. "Boolean" refers to anything that can only take on one of two values: "true" or "false."

To help make this more intuitive, Python has two special keywords that are conveniently named True and False. The capitalization is important - these keywords are not ordinary variables or strings, but are essentially synonyms for 1 and 0, respectively. Try doing some arithmetic in the interactive window using True and False inside of your expressions, and you'll see that they behave just like 1 and 0:

```
>>> 42 * True + False
42
>>> False * 2 - 3
-3
>>> True + 0.2 / True
1.2
>>>
```

Before we can evaluate expressions to determine whether or not they are True or False, we need a way to compare values to each other. We use a standard set of symbols (called boolean comparators) to do this, and most of them are probably already familiar to you:

1. `a > b` ---> `a` greater than `b`
2. `a < b` ---> `a` less than `b`
3. `a >= b` ---> `a` greater than or equal to `b`
4. `a <= b` ---> `a` less than or equal to `b`
5. `a != b` ---> `a` not equal to `b`
6. `a == b` ---> `a` equal to `b`

The last two symbols might require some explanation. The symbol `!=` is a sort of shortcut notation for saying "not equal to." Try it out on a few expressions, like these:

```
>>> 1 != 2
True

>>> 1 != 1
False
```

In the first case, since 1 does not equal 2, we see the result `True`. Then, in the second example, 1 does equal 1, so our test expression `1 != 1` returns `False`.

When we want to test if a equals b, we can't just use the expression `a = b` because in programming, that would mean we want to assign the value of b to a. We use the symbol `==` as a test expression when we want to see if a is equal to b. For instance, take a look at the two versions of "equals" here:

```
>>> a = 1
>>> b = 2
>>> a == b
False

>>> a = b
>>> a == b
True
```

First we assigned `a` and `b` two different values. When we check whether they are equal by using the `a == b` expression, we get the result `False`. Then, we reassign `a` the value of `b` by saying `a = b`. Now, since `a` and `b` are both equal to 2, we can test this relationship again by saying `a == b` (which is more of a question that we're asking), which returns the result of `True`. Likewise, we could ask:

```
>>> a != b
False
```

It's `True` that `2 == 2`. So the opposite expression, `a != b`, is `False`. In other words, it's not `True` that 2 does not equal 2.

We can compare strings in the same way that we compare numbers. Saying that one word is "less than" another doesn't really mean anything, but we can test whether two strings are the same by using the `==` or `!=` comparators:

```
>>> "dog" == "cat"
False

>>> "dog" == "dog"
True

>>> "dog" != "cat"
True
```

Keep in mind that two strings have to have exactly the same value for them to be equal. For instance, if one string has an extra space character at the end or if the two strings have different capitalization, comparing whether or not they are equal will return `False`.

Review exercises:

1. Figure out what the result will be (`True` or `False`) when evaluating the following expressions, then type them into the interactive window to check your answers:

   ```
   1 <= 1
   1 != 1
   1 != 2
   "good" != "bad"
   "good" != "Good"
   123 == "123"
   ```

Add Some Logic

Python also has special keywords (called **logical operators**) for comparing two expressions, which are: `and` , `or` , and `not` . These keywords work much the same way as we use them in English.

`and` keyword

Let's start with the `and` keyword. Saying "and" means that both statements must be true. For instance, take the following two simple phrases (and assume they're both true):

1. cats have four legs
2. cats have tails

If we use these phrases in combination, the phrase "cats have four legs and cats have tails" is true, of course. If we negate both of these phrases, "cats do not have four legs and cats do not have tails" is false. But even if we make one of these phrases false, the combination also becomes false: "cats have four legs and cats do not have tails" is a false phrase. Likewise, "cats do not have four legs and cats have tails" is still false.

Let's try this same concept out in Python by using some numerical expressions:

```
>>> 1 < 2 and 3 < 4 # both are True
True

>>> 2 < 1 and 4 < 3 # both are False
False

>>> 1 < 2 and 4 < 3 # second statement is False
False

>>> 2 < 1 and 3 < 4 # first statement is False
False
```

1. `1 < 2 and 3 < 4` : both statements are `True` , so the combination is also `True`
2. `< 1 and 4 < 3` : both statements are `False` , so their combination is also `False`
3. `1 < 2 and 4 < 3` : the first statement (`1 < 2`) is `True` , while the second statement (`4 < 3`) is `False` ; since both statements have to be `True` , combining them with the `and` keyword gives us `False`
4. `2 < 1 and 3 < 4` : the first statement (`2 < 1`) is `False` , while the second

statement (`3 < 4`) is `True` ; again, since both statements have to be `True` , combining them with the and keyword gives us `False`

We can summarize these results as follows:

Combination using `and` ---> Result

1. `True and True` ---> `True`
2. `True and False` ---> `False`
3. `False and True` --> `False`
4. `False and False` ---> `False`

```
>>> True and True
True

>>> True and False
False

>>> False and True
False

>>> False and False
False
```

> **NOTE**: It might seem counter-intuitive that "True and False" is False, but think back to how we use this term in English; the following phrase, taken in its entirety, is of course false: "cats have tails and the moon is made of cheese."

`or` keyword

The `or` keyword means that at least one value must be true. When we say the word "or" in everyday conversation, sometimes we mean an "exclusive or" - this means that only the first option or the second option can be true. We use an "exclusive or" when we say a phrase such as, "I can stay or I can go." I can't do both - only one of these options can be true.

However, in programming we use an "inclusive or" since we also want to include the possibility that both values are true. For instance, if we said the phrase, "we can have ice cream or we can have cake," we also want the possibility of ice cream and cake, so we use an "inclusive or" to mean "either ice cream, or cake, or both ice cream and cake."

Again, we can try this concept out in Python by using some numerical expressions:

```
>>> 1 < 2 or 3 < 4 # both are True
True

>>> 2 < 1 or 4 < 3 # both are False
False

>>> 1 < 2 or 4 < 3 # second statement is False
True

>>> 2 < 1 or 3 < 4 # first statement is False
True
```

If any part of our expression is True, even if both parts are True, the result will also be True. We can summarize these results as follows:

Combination using `or` ---> Result

1. `True or True` ---> `True`
2. `True or False` ---> `True`
3. `False or True` --> `True`
4. `False or False` ---> `False`

```
>>> True or True
True

>>> True or False
True

>>> False or True
True

>>> False or False
False
```

We can also combine the "and" and "or" keywords using parentheses to create more complicated statements to evaluate, such as the following:

```
>>> False or (False and True)
False

>>> True and (False or True)
True
```

`not` keyword

Finally, as you would expect, the `not` keyword simply reverses the truth of a single statement:

Effect of using `not` ---> Result

1. `not True` ---> `False`
2. `not False` ---> `True`

Using parentheses for grouping statements together, we can combine these keywords with `True` and `False` to create more complex expressions. For instance:

```
>>> (not False) or True
True

>>> False or (not False)
True
```

We can now combine these keywords with the boolean comparators that we learned in the previous section to compare much more complicated expressions. For instance, here's a somewhat more involved example:

```
True and not (1 != 1)
```

We can break this statement down by starting on the far right side, as follows:

1. We know that `1 == 1` is `True`, therefore…
2. `1 != 1` is `False`, therefore… `not (1 != 1)` can be simplified to `not (` False`)
3. `not False` is `True`
4. Now we can use this partial result to solve the full expression…
5. `True and not (1 != 1)` can be simplified to `True and True`
6. `True and True` evaluates to be `True`

When working through complicated expressions, the best strategy is to start with the most complicated part (or parts) of the expression and build outward from there. For instance, try evaluating this example:

```
("A" != "A") or not (2 >= 3)
```

We can break this expression down into two sections, then combine them:

1. We'll start with the expression on the left side of the `or` keyword
2. We know that "A" == "A" is `True`, therefore…

3. "A" != "A" is `False`, and the left side of our expression is `False`
4. Now on the right side of the `or`, `2 >= 3` is `False`, therefore… `not (2 >= 3)` is `True`
5. So our entire expression simplifies to `False or True`
6. `False or True` evaluates to be `True`

Did you notice that we didn't have to use parentheses to express either of these examples? However, it's usually best to include parentheses as long as they help to make a statement more easily readable.

> **NOTE**: You should feel very comfortable using various combinations of these keywords and boolean comparisons. Play around in the interactive window (shell), creating more complicated expressions and trying to figure out what the answer will be before checking yourself, until you are confident that you can decipher boolean logic.

Review exercises:

1. Figure out what the result will be (`True` or `False`) when evaluating the following expressions, then type them into the interactive window to check your answers:

```
(1 <= 1) and (1 != 1)
not (1 != 2)
("good" != "bad") or False
("good" != "Good") and not (1 == 1)
```

Control the Flow of Your Program

So far, we haven't really let our programs make any decisions on their own - we've been supplying a series of instructions, which our scripts follow in a specific order regardless of the inputs received.

However, now that we have precise ways to compare values to one other, we can start to build logic into our programs; this will let our code "decide" to go in one direction or another based on the results of our comparisons.

We can do this by using `if` statements to test when certain conditions are `True`.

> **NOTE**: The 'if' statement is one of the most commonly used 'things' in Python (or any programming language, for that matter). If you're flying through this course, slow down at this point and really focus on integrating this material.

Here's a simple example of an `if` statement:

```
if 2 + 2 == 4:
    print("2 and 2 is 4")
    print("Arithmetic works.")
```

Just like when we created `for` and `while` loops, when we use an `if` statement, we have a test condition and an indented block of text that will run or not based on the results of that test condition. Here, our test condition (which comes after the `if` keyword) is `2 + 2 == 4`. Since this expression is `True`, our `if` statement evaluates to `True` and all of the code inside of our `if` statement is run, displaying the two lines. We happened to use two print statements, but we could really put any code inside the `if` statement.

If our test condition had been `False` (for instance, `2 + 2 != 4`), nothing would have been displayed because our program would have jumped past all of the code inside this "if block" after finding that the `if` statement evaluated to `False`.

There are two other related keywords that we can use in combination with the `if` keyword: `else` and `elif`. We can add an `else` statement after an `if` statement like so:

```
if 2 + 2 == 4:
    print("2 and 2 is 4")
    print("Arithmetic works.")
else:
    print("2 and 2 is not 4")
    print("Big Brother wins.")
```

The `else` statement doesn't have a test condition of its own because it is a "catch-all" block of code; our else is just a synonym for "otherwise." So if the test condition of the `if` statement had been `False`, the two lines inside of the "else block" would have run instead. However, since our test condition (`2 + 2 == 4`) is `True`, the section of code inside the "else block" is not run.

The `elif` keyword is short for "else if" and can be used to add additional options (or conditions) after an `if` statement. For instance, we could combine an `if`, an `elif`, and an `else` into a single script like this:

```
num = 15

if num < 10:
    print("number is less than 10")
elif num > 10:
    print("number is greater than 10")
else:
    print("number is equal to 10")
```

After creating an integer object named `num` with a value of 15, we first test whether or not our `num` is less than 10; since this condition is `False`, we jump over the first `print` statement without running it. We land at the `elif` statement, which (because the test condition of the `if` statement was `False`) offers us an alternative test; since 15 > 10, this condition is `True`, and we display the second `print` statement's text. Since one of our test conditions was `True`, we skip over the else statement entirely; if both the `if` and the `elif` had been `False`, however, we would have displayed the last line in the `else` statement.

This is called **branching** because we are deciding which "branch" of the code to go down; we have given our code different paths to take based on the results of the test conditions. Try running the script above and changing `num` to be a few different values to make sure you understand how `if`, `elif` and `else` work together.

You can also try getting rid of the "else block" entirely; we don't have to give our code a path to go down, so it's completely acceptable if we skip over the entire set of `if` / `elif` statements without taking any action. For instance, get rid of the last two lines in the script above and try running it again with `num = 10`.

It's important to note that `elif` can only be used after an `if` statement and that `else` can only be used at the end of a set of `if` / `elif` statements (or after a single `if`), since using one of these keywords without an `if` wouldn't make any sense. Likewise, if we want to do two separate tests in a row, we should use two separate `if` statements.

For instance, try out this short script:

```
if 1 < 2:
    print("1 is less than 2")
elif 3 < 4:
    print("3 is less than 4")
else:
    print("Who moved my cheese?")
```

The first test condition (`1 < 2`) is `True`, so we print the first line inside the "if block". Since we already saw one `True` statement, however, the program doesn't even bother to check whether the "elif block" or the "else block" should be run; these are only alternative options. So, even though it's `True` that 3 is less than 4, we only ever run the first print statement.

Just like with `for` and `while` loops, we can nest `if` statements inside of each other to create more complicated paths:

```
want_cake = "yes"
have_cake = "no"

if want_cake == "yes":
    print("We want cake...")
    if have_cake == "no":
        print("But we don't have any cake")
    elif have_cake == "yes":
        print("And it's our lucky day")
else:
    print("The cake is a lie.")
```

In this example, we first check if we want cake - and since we do, we enter the first "if block". After displaying our desire for cake, we enter the inside set of `if` / `elif` statements. We check and display that we don't have any cake. If it had been the case that `have_cake` was "yes" then it would have been our lucky day. On the other hand, if `have_cake` had been any value other than "yes" or "no" then we wouldn't have displayed a second line at all, since we used `elif` rather than `else`. If (for whatever twisted reason) we had set the value of `want_cake` to anything other than "yes", we would have jumped down to the bottom `else` and only displayed the last `print` statement.

Review exercises:

1. Write a script that prompts the user to enter a word using the `input()` function, stores that input in a variable, and then displays whether the length of that string is less than 5 characters, greater than 5 characters, or equal to 5 characters by using a set of `if`, `elif` and `else` statements.

Assignment: Find the factors of a number

1. Write a script *factors.py* that includes a function to find all of the integers that divide evenly into an integer provided by the user. A sample run of the program should look like this (with the user's input highlighted in bold):

```
>>>
Enter a positive integer: 12
1 is a divisor of 12
2 is a divisor of 12
3 is a divisor of 12
4 is a divisor of 12
6 is a divisor of 12
12 is a divisor of 12
>>>
```

You should use the `%` operator to check divisibility. This is called the "**modulus** operator" and is represented by a percent symbol in Python. It returns the remainder of any division. For instance, 3 goes into 16 a total of 5 times with a remainder of 1, therefore 16 % 3 returns 1. Meanwhile, since 15 is divisible by 3, 15 % 3 returns 0.

Also keep in mind that `input()` always returns a string, so you will need to convert this value to an integer before using it in any calculations.

Break Out of the Pattern

There are two main keywords that help to control the flow of programs in Python: `break` and `continue`. These keywords can be used in combination with `for` and `while` loops and with `if` statements to allow us more control over where we are in a loop.

Let's start with `break`, which does just that: it allows you to break out of a loop. If we wanted to break out of a for loop at a particular point, our script might look like this:

```
for i in range(0, 4):
    if i == 2:
        break
    print(i)

print("Finished with i = ", str(i))
```

Without the `if` statement that includes the break command, our code would normally just print out the following output:

```
0
1
2
3
Finished with i = 3
```

Instead, each time we run through the loop, we are checking whether `i == 2`. When this is `True`, we break out of the loop; this means that we quit the for loop entirely as soon as we reach the break keyword. Therefore, this code will only display:

```
0
1
Finished with i = 2
```

Notice that we still displayed the last line since this wasn't part of the loop. Likewise, if we were in a loop inside another loop, `break` would only break out of the inner loop; the outer loop would continue to run (potentially landing us back inside the inner loop again).

Much like `break`, the `continue` keyword jumps to the end of a loop; however, instead of exiting a loop entirely, `continue` says to go back to the top of the loop and continue with the next item of the loop. For instance, if we had used `continue` instead of `break` in our last example-

```
for i in range(0, 4):
    if i == 2:
        continue
    print(i)

print("Finished with i = ", str(i))
```

-we're now skipping over the `print i` command when our `if` statement is `True`, and so our output includes everything from the loop except displaying `i` when `i == 2`:

```
0
1
3
Finished with i = 3
```

> **NOTE**: It's always a good idea to give short but descriptive names to your variables that make it easy to tell what they are supposed to represent. The letters i, j and k are exceptions because they are so common in programming; these letters are almost always used when we need a "throwaway" number solely for the purpose of keeping count while working through a loop.

Loops can have their own `else` statements in Python as well, although this structure isn't used very frequently. Tacking an `else` onto the end of a loop will make sure that your "else block" always runs after the loop unless the loop was exited by using the `break` keyword. For instance, let's use a `for` loop to look through every character in a string, searching for the upper-case letter "X":

```
phrase = "it marks the spot"

for letter in phrase:
    if letter == "X":
        break
else:
    print("There was no 'X' in the phrase")
```

Here, our program attempted to find "X" in the string phrase, and would break out of the `for` loop if it had. Since we never broke out of the loop, we reached the `else` statement and displayed that "X" wasn't found. If you try running the same program on the phrase "it marks X the spot" or some other string that includes an "X", however, there will be no output at all because the block of code in the else will not be run.

Likewise, an `else` placed after a while loop will always be run unless the `while` loop has been exited using a `break` statement. For instance, try out the following script:

```
tries = 0

while tries < 3:
    password = input("Password: ")
    if password == "I<3Bieber":
        break
    else:
        tries = tries + 1
else:
    print("Suspicious activity. The authorities have been alerted.")
```

Here, we've given the user three chances to enter the correct password. If the password matches, we break out of the loop. Otherwise (the first `else`), we add one to the `tries` counter. The second "else block" belongs to the loop (hence why it's less indented than the first `else`), and will only be run if we don't break out of the loop. So when the user enters a correct password, we do not run the last line of code, but if we exit the loop because the test condition was no longer `True` (i.e., the user tried three incorrect passwords), then it's time to alert the authorities.

A commonly used shortcut in Python is the `+=` operator, which is shorthand for saying "increase a number by some amount". For instance, in our last code example above, instead of the line: `tries = tries + 1`, we could have done the exact same thing (adding 1 to `tries`) by saying: `tries += 1`.

In fact, this works with the other basic operators as well; `-=` means "decrease", `*=` means "multiply by", and `/=` means "divide by".

For instance, if we wanted to change the variable `tries` from its original value to that value multiplied by 3, we could shorten the statement to say: `tries *= 3`

Review exercises:

1. Use a `break` statement to write a script that prompts the users for input repeatedly, only ending when the user types "q" or "Q" to quit the program; a common way of creating an infinite loop is to write `while True:`.
2. Combine a `for` loop over a `range()` of numbers with the `continue` keyword to print every number from 1 through 50 except for multiples of 3; you will need to use the `%` operator.

Recover from errors

You've probably already written lots of scripts that generated errors in Python. Run-time errors (so-called because they happen once a program is already running) are called **exceptions**. Congratulations - you've made the code do something exceptional.

There are different types of exceptions that occur when different rules are broken. For instance, try to get the value of "1 / 0" at the interactive window, and you'll see a ZeroDivisionError exception. Here's another example of an exception, a `ValueError`, that occurs when we try (and fail) to turn a string into an integer:

```
>>> int("not a number")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int("not a number")
ValueError: invalid literal for int() with base 10: 'not a number'
>>>
```

The name of the exception is displayed on the last line, followed by a description of the specific problem that occurred.

When we can predict that a certain type of exception might occur, we might not be able to prevent the error, but there are ways that we can recover from the problem more gracefully than having our program break and display lots of angry messages.

In order to stop our code from breaking because of a particular exception, we use a "try/except" pair, as in the following:

```
try:
    number = int(input("Enter an integer: "))
except ValueError:
    print("That was not an integer.")
```

The first thing that happens in a try/except pair is that everything inside of the "try block" is run normally. If no error occurs, the code skips over the "except block" and continues running normally. Since we said, `except ValueError`, however, if the program encounters a `ValueError` (when the user enters something that isn't an integer), we jump down to the "except block" and run everything there. This avoids Python's automatic error display and doesn't break the script since we "caught" the `ValueError`.

If a different kind of exception had occurred, then the program still would have broken; we only handled one type of exception (a `ValueError`) with our "except block".

A single "except block" can handle multiple types of exceptions by separating the exception names with commas and putting the list of names in parentheses:

```
def divide(num1, num2):
    try:
        print(num1 / num2)
    except (TypeError, ZeroDivisionError):
        print("encountered a problem")
```

This isn't used very frequently since we usually want our code to react specifically to each type of exception differently. In this case, we created a `divide()` function that tries to divide two numbers. If one or both of the numbers aren't actually numbers, a `TypeError` exception will occur because we can't use something that isn't a number for division. And if we provide 0 as the second number, a `ZeroDivisionError` will occur since we can't divide by zero. Both of these exceptions will be caught by our "except block", which will just let the user know that we "encountered a problem" and continue on with anything else that might be left in our script outside of the try/except pair.

More `except` error handling blocks can be added after the first `except` to catch different types of exceptions, like so:

```
try:
    number = int(input("Enter an non-zero integer: "))
    print("10 / {} = {}".format(number, 10.0/number))
except ValueError:
    print("You did not enter an integer.")
except ZeroDivisionError:
    print("You cannot enter 0.")
```

Here, we might have encountered two different types of errors. First, the user might not have input an integer; when we try to convert the string input using `int()`, we raise an exception and jump to the "except ValueError: block", displaying the problem. Likewise, we could have tried to divide 10 by the user-supplied number and, if the user gave us an input of 0, we would have ended up with a `ZeroDivisionError` exception; instead, we jump to the second "except block" and display this problem to the user.

A list of Python's built-in exceptions can be found here. **It's usually easiest to figure out the name of an exception by purposefully causing the error to occur yourself**, although you should then read the documentation on that particular type of exception to make sure that your code will actually handle all of the errors that you expect and (just as importantly) that your program will still break if it encounters a different, unexpected type of exception.

We can also use an "except block" by itself without naming specific exceptions to catch:

```
try:
    # do lots of hazardous things that might break
except:
    print("The user must have screwed something up.")
```

However, this is dangerous to do and is usually not a good idea at all. It's easy to hide a poorly written section of code behind a `try/except` and think that everything was working fine, only to discover later that you were silencing all sorts of unexpected problems that should never have occurred.

For more on exception handling, check out this excellent article.

Review exercises:

1. Write a script that repeatedly asks the user to input an integer, displaying a message to "try again" by catching the `ValueError` that is raised if the user did not enter an integer; once the user enters an integer, the program should display the number back to the user and end without crashing

Simulate Events and Calculate Probabilities

You'll probably find the assignments in this section to be fairly difficult, especially if you're not very mathematically inclined. At the very least, I encourage you to read through this section for the information and try the assignments out - if they're too tricky for now, move on and come back to them later.

We will be running a simple simulation known as a Monte Carlo experiment. In order to do this, we need to add a real-world "element of chance" to our code. Python has built-in functionality for just that purpose, and it's suitably called the `random` module.

A **module** is just a collection of related functions. For now, all we will need from the random module is the `randint()` function. Calling `randint(x, y)` on two integers `x` and `y` returns a random (evenly distributed) integer that will have a value between `x` and `y` - including both `x` and `y`, unlike the `range()` function.

We can import this function into our program like so:

```
from random import randint
```

Now we can use the `randint()` function in our code:

```
print(randint(0, 1))
```

If you try this within the interactive window, you should see something like this:

```
>>> from random import randint
>>> print(randint(0, 1))
0
```

Of course, because the output is random, you have a 50% chance of getting a 1 instead.

Okay, let's take everything we've learned so far and put it all together to solve a real problem. We'll start with a basic probability question by simulating the outcome of an event.

Let's say we flip a fair coin (50/50 chance of heads or tails), and we keep flipping it until we get it to land on heads. If we keep doing this, we'll end up with a bunch of individual trials that might include getting heads on the first flip, tails on one flip and heads on the second, or even occasionally tails, tails, tails, tails, tails and finally heads. On average, what's our expected ratio of heads to tails for an individual trial?

To get an accurate idea of the long-term outcome, we'll need to do this lots of times - so let's use a `for` loop:

```
for trials in range(0, 10000):
```

We can use the `randint()` function to simulate a 50/50 coin toss by considering 0 to represent "tails" and 1 to be a "heads" flip. The logic of the problem is that we will continue to toss the coin as long as we get tails, which sounds like a while loop:

```
while randint(0, 1) == 0:
```

Now all we have to do is keep track of our counts of heads and tails. Since we only want the average, we can sum them all up over all our trials, so our full script ends up looking like the following:

```
from random import randint

heads = 0
tails = 0

for trial in range(0, 10000):
    while randint(0, 1) == 0:
        tails = tails + 1
    heads = heads + 1

print("heads / tails = ", heads/tails)
```

Each time we toss tails, we add one to our total tails tally. Then we go back to the top of our while loop and generate a new random number to test. Once it's not true that `randint(0, 1) == 0`, this means that we must have tossed heads, so we exit out of the while loop (because the test condition isn't true that time) and add one to the total heads tally.

Finally, we just print out the result of our heads-to-tails ratio. If you use a large enough sample size and try this out a few times, you should be able to figure out (if you hadn't already) that the ratio approaches 1:1.

Of course, you probably could have calculated this answer faster just by working out the actual probabilities, but this same method can be applied to much more complicated scenarios. For instance, one popular application of this sort of Monte Carlo simulation is to predict the outcome of an election based on current polling percentages.

Review exercises:

1. Write a script that uses the `randint()` function to simulate the toss of a die, returning a random number between 1 and 6.
2. Write a script that simulates 10,000 throws of dice and displays the average number resulting from these tosses.

Assignment: Simulate an election

Write a script *election.py* that will simulate an election between two candidates, A and B. One of the candidates wins the overall election by a majority based on the outcomes of three regional elections. (In other words, a candidate wins the overall election by winning at least two regional elections.) Candidate A has the following odds:

- 87% chance of winning region 1
- 65% chance of winning region 2
- 17% chance of winning region 3

Import and use the `random()` function from the `random` module to simulate events based on probabilities; this function doesn't take any arguments (meaning you don't pass it any input variables) and returns a random value somewhere between 0 and 1.

Simulate 10,000 such elections, then (based on the average results) display the probability that Candidate A will win and the probability that Candidate B will win.

Hint: To do this, you'll probably need to use a `for` loop with a lot of `if` / `else` statements to check the results of each regional election.

Assignment: Simulate a coin toss experiment

Write a script *coin_toss.py* that uses coin toss simulations to determine the answer to this slightly more complex probability puzzle:

I keep flipping a fair coin until I've seen it land on both heads and tails at least once each - in other words, after I flip the coin the first time, I continue to flip it until I get a different result. On average, how many times will I have to flip the coin total? Again, the actual probability could be worked out, but the point here is to simulate the event using `randint()`. To get the expected average number of tosses, you should set a variable `trials = 10000` and a variable `flips = 0`, then add 1 to your `flips` variable every time a coin toss is made. Then you can print `flips / trials` at the end of the code to see what the average number of flips was.

This one is tricky to structure correctly. Try writing out the logic before you start coding. Some additional pointers if you're stuck:

1. You will need to use a `for` loop over a range of trials.
2. For each trial, first you should check the outcome of the first flip.
3. Make sure you add the first flip to the total number of flips.
4. After the first toss, you'll need another loop to keep flipping while you get the same result as the first flip.

If you just want to check whether or not your final answer is correct without looking at the sample code, click here.

Fundamentals: Lists and Dictionaries

Lists are extremely useful - in life and in Python. There are so many things that naturally lend themselves to being put into lists that it's often a very intuitive way to store and order data. In Python, a **list** is a type of object (just like a string or an integer), except a list object is able to hold other objects inside of it. We create a list by simply listing all the items we want in a list, separated by commas, and enclosing everything inside square brackets. These are all examples of simple list objects:

```
colors = ["red", "green", "burnt sienna", "blue"]
scores = [10, 8, 9, -2, 9]
my_list = ["one", 2, 3.0]
```

The first object, `colors`, holds a list of four strings. Our `scores` list holds five integers. And the last object, `my_list`, holds three different objects - a string, an integer and a floating-point number. Since we usually want to track a set of similar items, it's not very common to see a list that holds different types of objects like our last example, but it's possible.

Getting a single item out of a list is as easy as getting a single character out of a string - in fact, it's the exact same process of referring to the item by its index number:

```
>>> colors = ["red", "green", "burnt sienna", "blue"]
>>> print(colors[2])
burnt sienna
>>>
```

Remember, since we start counting at 0 in Python, we asked for `colors[2]` in order to get what we think of as the third object in the list. We can also get a range of objects from a list in the same way that we got substrings out of strings:

```
>>> colors = ["red", "green", "burnt sienna", "blue"]
>>> print(colors)
['red', 'green', 'burnt sienna', 'blue']
>>> print(colors[0:2])
['red', 'green']
>>>
```

First we printed the entire list of colors, which displayed almost exactly the same way as we typed it in originally. Then we printed the objects in the list within the range [0:2] - this includes the objects at positions 0 and 1, which are returned as a smaller list. (We can tell they're still in a list because of the square brackets.)

Unlike strings, lists are **mutable objects**, meaning that we can change individual items within a list. For instance:

```
>>> colors = ["red", "green", "burnt sienna", "blue"]
>>> colors[0] = "burgundy"
>>> colors[3] = "electric indigo"
>>> print(colors)
['burgundy', 'green', 'burnt sienna', 'electric indigo']
>>>
```

Since we can change the items in lists, we can also create new lists and add objects to them, item by item. We create an empty list using an empty pair of square brackets, and we can add an object to the list using the `append()` method of the list:

```
>>> animals = []
>>> animals.append("lion")
>>> animals.append("tiger")
>>> animals.append("frumious Bandersnatch")
>>> print(animals)
['lion', 'tiger', 'frumious Bandersnatch']
>>>
```

Likewise, we can remove objects from the list using the `remove()` method of the list:

```
>>> animals.remove("lion")
>>> animals.remove("tiger")
>>> print(animals)
['frumious Bandersnatch']
>>>
```

The `extend()` method can be used to add more than one object to a list:

```
>>> cities = []
>>> cities.append("New York")
>>> cities
['New York']
>>> cities.extend(["San Francisco", "Boston", "Chicago"])
>>> cities
['New York', 'San Francisco', 'Boston', 'Chicago']
>>>
```

We can also use the list's `index()` method to get the index number of a particular item in a list in order to determine its position. For instance:

```
>>> colors = ["red", "green", "burnt sienna", "blue"]
>>> print(colors.index("burnt sienna"))
2
>>>
```

Copying one list into another list, however, is somewhat unintuitive. You can't just reassign one list object to another list object, because you'll get this (possibly surprising) result:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals
>>> large_cats.append("Tigger")
>>> print(animals)
['lion', 'tiger', 'frumious Bandersnatch', 'Tigger']
>>>
```

We tried to assign the list stored in the `animals` variable to the variable `large_cats` , then we added another string into `large_cats` . But when we display the contents of `animals` , we see that we've also changed our original list - even though we meant to create a new list by giving `large_cats` the values in `animals` .

This is a quirk of object-oriented programming, but it's by design; when we say `large_cats = animals` , we make the lists `large_cats` and `animals` both refer to the same object. When we created our first list, the name animals was only a way to point us to a list object - in other words, the name animals is just a way to reference the actual list object that is somewhere in the computer's memory. Instead of copying all the contents of the list object and creating s new list, saying `large_cats = animals` assigns the same object reference to large_cats; both of our list names now refer to the same object, and any changes made to one will affect the other since both names point to the same object.

If we actually wanted to copy the contents of one list object into a new list object, we have to retrieve all the individual items from the list and copy them over individually. We don't have to use a loop to do this, however; we can simply say:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals[:]
>>> large_cats.append("leopard")
>>> print(large_cats)
['lion', 'tiger', 'frumious Bandersnatch', 'leopard']
>>> print(animals)
["lion", "tiger", "frumious Bandersnatch"]
```

The '[:]' is the same technique that we used to retrieve a subset of the list over some range, but since we didn't specify an index number on either side of the colon, we grabbed everything from the first item through the last item.

You can also achieve the same results by using the `extend()` method:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = []
>>> large_cats.extend(animals)
>>> large_cats
['lion', 'tiger', 'frumious Bandersnatch']
```

Keep in mind that because lists are mutable, there is no need to reassign a list to itself when we use one of its methods. In other words, we only need to say `animals.append("jubjub")` to add the jubjub to the `animals` list. If we had said `animals = animals.append("jubjub")`, we would have saved the result returned by the `append()` method (which is nothing) into animals, wiping out our list entirely.

This is true of all the methods that belong to mutable objects. For instance, lists also have a `sort()` method that sorts all of the items in ascending order (usually alphabetical or numerical, depending on the objects in the list); all we have to say if we want to sort the animals list is `animals.sort()`, which alphabetizes the list:

```
>>> animals.sort()
>>> print(animals)
['frumious Bandersnatch', 'jubjub', 'lion', 'tiger', 'Tigger']
>>>
```

If we had instead assigned the value returned by the `sort()` method to our list, we would have lost the list entirely:

```
>>> animals = animals.sort()
>>> print(animals)
None
>>>
```

Since lists can hold any objects, we can even put lists inside of lists. We sometimes make a "list of lists" in order to create a simple matrix. To do this, we simply nest one set of square brackets inside of another, like so:

```
>>> two_by_two = [[1, 2], [3, 4]]
```

We have a single list with two objects in it, both of which are also lists of two objects each. We now have to stack the index values we want in order to reach a particular item, for instance:

```
>>> two_by_two[1][0]
3
>>>
```

Since saying `two_by_two[1]` by itself would return the list `[3, 4]`, we then had to specify an additional index in order to get a single number out of this sub-list.

A list is just a sequence of objects, so nested lists don't have to be symmetrical:

```
>>> list = ["I heard you like lists", ["so I put a list", "in your list"]]
>>> print(list)
['I heard you like lists', ['so I put a list', 'in your list']]
>>>
```

Finally, if we want to create a list from a single string, we can use the string `split()` method as an easy way of splitting one string up into individual list items by providing the character (or characters) occurring between these items. For instance, if we had a single string of grocery items, each separated by commas and spaces, we could turn them into a list of items like so:

```
>>> groceries = "eggs, spam, pop rocks, cheese"
>>> grocery_list = groceries.split(", ")
>>> print(grocery_list)
['eggs', 'spam', 'pop rocks', 'cheese']
>>>
```

Review exercises:

1. Create a list named `desserts` that holds the two string values "ice cream" and "cookies"
2. Sort `desserts` in alphabetical order, then display the contents of the list
3. Display the index number of "ice cream" in the `desserts` list
4. Copy the contents of the `desserts` list into a new list object named `food`
5. Add the strings "broccoli" and "turnips" to the food `list`
6. Display the contents of both lists to make sure that "broccoli" and "turnips" haven't been added to `desserts`
7. Remove "cookies" from the food `list`
8. Display the first two items in the food `list` by specifying an index range
9. Create a list named `breakfast` that holds three strings, each with the value of "cookies", by splitting up the string "cookies, cookies, cookies"
10. Define a function that takes a list of numbers, `[2, 4, 8, 16, 32, 64]`, as the argument and then outputs only the numbers from the list that fall between 1 and 20 (inclusive)

Assignment: List of lists

Define a function, `enrollment_stats()`, that takes, as an input, a list of lists where each individual list contains three elements: (a) the name of a university, (b) the total number of enrolled students, and (c) the annual tuition fees.

Sample list:

```
universities = [
    ['California Institute of Technology', 2175, 37704],
    ['Harvard', 19627, 39849],
    ['Massachusetts Institute of Technology', 10566, 40732],
    ['Princeton', 7802, 37000],
    ['Rice', 5879, 35551],
    ['Stanford', 19535, 40569],
    ['Yale', 11701, 40500]
]
```

This function should return two lists: the first containing all of the student enrollment values and the second containing all of the tuition fees.

Next, define a `mean()` and a `median()` function. Both functions should take a single list as an argument and return the mean and median from the values in each list. Use the return values from `enrollment_stats()` as arguments for each function. *Challenge yourself by finding a way to sum all the values in a `list` without using the built-in 'sum()' function for calculating the mean.*

At some point you should calculate the total students enrolled and the total tuition paid.

Finally, output all values:

```
*************************
Total students:   77285
Total tuition:  $ 271905

Student mean:     11040
Student median:   10566

Tuition mean:   $ 38843
Tuition median: $ 39849
*************************
```

Assignment: Wax poetic

Write a script *poetry.py* that will generate a poem based on randomly chosen words and a pre-determined structure. When you are done, you will be able to generate poetic masterpieces such as the following in mere milliseconds:

```
A furry horse

A furry horse curdles within the fragrant mango
extravagantly, the horse slurps
the mango meows beneath a balding extrovert
```

All of the poems will have this same general structure, inspired by Clifford Pickover:

```
{A/An} {adjective1} {noun1}

{A/An} {adjective1} {noun1} {verb1} {preposition1} the {adjective2} {noun2}
{adverb1}, the {noun1} {verb2}
the {noun2} {verb3} {preposition2} a {adjective3} {noun3}
```

Your script should include a function `makePoem()` that returns a multi-line string representing a complete poem. The main section of the code should simply print `makePoem()` to display a single poem. In order to get there, use the following steps as a guide:

1. First, you'll need a vocabulary from which to create the poem. Create several lists, each containing words pertaining to one part of speech (more or less); i.e., create separate lists for nouns, verbs, adjectives, adverbs, and prepositions. You will need to include at least three different nouns, three verbs, three adjectives, two prepositions and one adverb. You can use the sample word lists below, but feel free to add your own:

    - Nouns: "fossil", "horse", "aardvark", "judge", "chef", "mango", "extrovert", "gorilla"
    - Verbs: "kicks", "jingles", "bounces", "slurps", "meows", "explodes", "curdles"
    - Adjectives: "furry", "balding", "incredulous", "fragrant", "exuberant", "glistening"
    - Prepositions: "against", "after", "into", "beneath", "upon", "for", "in", "like", "over", "within"
    - Adverbs: "curiously", "extravagantly", "tantalizingly", "furiously", "sensuously"

2. Choose random words from the appropriate list using the `random.choice()` function, storing each choice in a new string. Select three nouns, three verbs, three adjectives, one adverb, and two prepositions. Make sure that none of the words are repeated. (Hint: Use a while loop to repeat the selection process until you get a new word.)
3. Plug the words you selected into the structure above to create a poem string by using the `format()` string method
4. Bonus: Make sure that the "A" in the title and the first line is adjusted to become an "An" automatically if the first adjective begins with a vowel.

Make Permanent Lists

Tuples are very close cousins of the list object. The only real difference between lists and tuples is that tuple objects are **immutable** - they can't be changed at all once they have been created. Tuples can hold any list of objects, and they even look nearly identical to lists, except that we use parentheses instead of square brackets to create them:

```
>>> my_tuple = ("you'll", "never", "change", "me")
>>> print(my_tuple)
("you'll", 'never', 'change', 'me')
>>>
```

Since tuples are immutable, they don't have methods like `append()` and `sort()`. However, we can reference the objects in tuples using index numbers in the same way as we did with lists:

```
>>> my_tuple[2]
'change'
>>> my_tuple.index("me")
3
>>>
```

You probably won't create your own tuples very frequently, although we'll see some instances later on when they become necessary. One place where we tend to see tuples is when a function returns multiple values; in this case, we wouldn't want to accidentally change anything about those values or their ordering, so the function provides them to us as a "permanent" list:

```
>>> def adder_subtractor(num1, num2):
...     add = num1 + num2
...     subtract = num1 - num2
...     return add, subtract
...
>>> adder_subtractor(3, 2)
(5, 1)
>>> test = adder_subtractor(4, 3)
>>> test
(7, 1)
>>> type(test)
<type 'tuple'>
>>>
```

Parentheses are actually optional when we are creating a tuple; we can also just list out a set of objects to assign to a new object, and Python will assume by default that we mean to create a tuple:

```
>>> coordinates = 4.21, 9.29
>>> print coordinates
(4.21, 9.29)
>>>
```

This process is called **tuple packing** because we are "packing" a number of objects into a single immutable tuple object. If we were to receive the above coordinates tuple from a function and we then want to retrieve the individual values from this tuple, we can perform the reverse process, suitably called **tuple unpacking**:

```
>>> x, y = coordinates
>>> print(x)
4.21
>>> print(y)
9.29
>>>
```

We assigned both new variables to the tuple coordinates, separating the names with commas, and Python automatically knew how to hand out the items in the tuple. In fact, we can always make multiple assignments in a single line by separating the names with commas, whether or not we use a tuple:

```
>>> str1, str2, str3 = "a", "b", "c"
>>> print(str1)
a
>>> print(str2)
b
>>> print(str3)
c
>>>
```

This works because Python is basically doing tuple packing and tuple unpacking on its own in the background. *However, we don't use this very frequently because it usually only makes code harder to read and more difficult to update when changes are needed.*

Review exercises:

1. Create a tuple named `cardinal_nums` that holds the strings "first", "second" and "third" in order
2. Display the string at position 2 in `cardinal_nums` by using an index number
3. Copy the tuple values into three new strings named `pos1` , `pos2` and `pos3` in a single line of code by using tuple unpacking, then print those string values

Store Relationships in Dictionaries

One of the most useful structures in Python is a **dictionary**. Like lists and tuples, dictionaries are used as a way to store a collection of objects. However, the order of the objects in a dictionary is unimportant. Instead, dictionaries hold information in pairs of data, called key-value pairs. Every key in a dictionary is associated with a single value. Let's take a look at an example in which we create a dictionary that represents entries in a phonebook:

```
>>> phonebook = {"Jenny": "867-5309", "Mike Jones": "281-330-8004",
"Destiny": "900-783-3369"}
>>> print(phonebook)
{'Mike Jones': '281-330-8004', 'Jenny': '867-5309',
'Destiny': '900-783-3369'}
>>>
```

The keys in our example phonebook are names of people. Keys are joined to their values with colons, where each value is a phone number. The key-value pairs are each separated by commas, just like the items in a list or tuple. While lists use square brackets and tuples use parentheses, dictionaries are enclosed in curly braces, {}. Just as with empty lists, we could have created an empty dictionary by using only a pair of curly braces.

Notice how, when we displayed the contents of the dictionary we had just created, the key-value pairs appeared in a different order. Python sorts the contents of the dictionary in a way that makes it very fast to get information out of the dictionary (by a process called **hashing**), but this ordering changes randomly every time the contents of the dictionary change.

Instead of the order of the items, what we really care about is which value belongs to which key. This is a very natural way to represent many different types of information. For instance, I probably don't care which number I happen to put into my phonebook first; I only want to know which number belongs to which person. We can retrieve this information the same way as we did with lists, using square brackets, except that we specify a key instead of an index number:

```
>>> phonebook["Jenny"]
'867-5309'
>>>
```

We can add entries to a dictionary by specifying the new key in square brackets and assigning it a value:

```
>>> phonebook["Obama"] = "202-456-1414"
>>> print(phonebook)
{'Mike Jones': '281-330-8004', 'Obama': '202-456-1414',
'Jenny': '867-5309', 'Destiny': '900-783-3369'}
>>>
```

We're not allowed to have duplicate keys in a Python dictionary; in other words, each key can only be assigned a single value. This is because having duplicate keys would make it impossible to identify which key we mean when we're trying to find the key's associated value. If a key is given a new value, Python just overwrites the old value. For instance, perhaps Jenny got a new number:

```
>>> phonebook["Jenny"] = "555-0199"
>>> print(phonebook)
{'Mike Jones': '281-330-8004', 'Obama': '202-456-1414',
'Jenny': '555-0199', 'Destiny': '900-783-3369'}
>>>
```

To remove an individual key-value pair from a dictionary, we use the `del()` function (short for delete):

```
>>> del(phonebook["Destiny"])
>>> print(phonebook)
{'Mike Jones': '281-330-8004', 'Obama': '202-456-1414',
'Jenny': '555-0199'}
>>>
```

Often we will want to loop over all of the keys in a dictionary. We can get all of the keys out of a dictionary by using the `keys()` method:

```
>>> print(phonebook.keys())
(['Mike Jones', 'Jenny', 'Obama'])
>>>
```

They `keys()` method returns an object of the type `dict_keys` :

```
names = phonebook.keys()
>>>type(names)
<class 'dict_keys'>
>>>
```

This can be a problem because `dict_keys` object look like `lists`, but they are not, and therefore do not have the same methods available to them. To be able to reference these keys by index (as well as use other list methods such as sort(), sum(), etc.) we need to convert this object to a list like so:

```
>>>names = list(phonebook.keys())
>>>type(names)
<class 'list'>
```

It we wanted to do something specific with each of the dictionary's keys, though, what's usually even easier to do is to use a for loop to get each key individually. Saying `for x in dictionary` automatically gives us each key in the dictionary. We can then use the variable name in our for loop to get each corresponding value out of our dictionary:

```
>>>
for contact_name in phonebook:
    print(contact_name, phonebook[contact_name])

Mike Jones 281-330-8004
Jenny 555-0199
Obama 202-456-1414
>>>
```

We can also use the `in` keyword to check whether or not a particular key exists in a dictionary:

```
>>> "Jenny" in phonebook
True
>>> "Santa" in phonebook
False
>>>
```

This expression (`x in dictionary`) is usually used in an `if` statement, for instance before deciding whether or not to try to get the corresponding value for that key. This is important because it's an error to attempt to get a value for a key that doesn't exist in a dictionary:

```
>>> phonebook["Santa"]
Traceback (most recent call last):
  File "<pyshell#1>", line 1,
in <module>
    phonebook["Santa"]
KeyError: 'Santa'
>>>
```

If we do want to access a dictionary's keys in their sorted order, we can use Python's `sorted()` function to loop over the keys alphabetically:

```
>>>
for contact_name in sorted(phonebook):
    print(contact_name, phonebook[contact_name])

Jenny 555-0199
Mike Jones 281-330-8004
Obama 202-456-1414
>>>
```

Keep in mind that `sorted()` doesn't re-sort the order of the *actual* dictionary:

```
>>> for contact_name in sorted(phonebook):
...     print(contact_name, phonebook[contact_name])
...
Jenny 555-0199
Mike Jones 281-330-8004
Obama 202-456-1414
>>> phonebook
{'Mike Jones': '281-330-8004', 'Jenny': '555-0199',
'Obama': '202-456-1414'}
```

The change is temporary. Python has to keep the apparently haphazard ordering of keys in the dictionary in order to be able to access the dictionary keys quickly using its own complicated hash function.

Dictionaries are very flexible and can hold a wide variety of information beyond the strings that we've experimented with here. Although it's usually the case, dictionaries don't even have to hold keys or values that are all the same types of objects. Dictionary values can be anything, while keys must be immutable objects. For instance, we could have added a key to our phonebook that was an integer object. However, we couldn't have added a list as a key, since that list could be modified while it's in the dictionary, breaking the overall structure of the dictionary.

Dictionary values can even be other dictionaries, which is more common than it probably sounds. For instance, we could imagine a more complicated phonebook in which every key is a unique contact name that is associated with a dictionary of its own; these individual contact dictionaries could then include keys describing the phone number ("home", "work", custom supplied type, etc.) that are each associated with a "phone number" value. This is much like creating a list of lists:

```
>>> contacts = {"Jenny": {"cell": "555-0199", "home": "867-5309"}, "Mike
Jones": {"home": "281-330-8004"}, "Destiny": {"work": "900-783-3369"}}
>>> print(contacts)
{'Mike Jones': {'home': '281-330-8004'},
'Jenny': {'cell': '555-0199', 'home':
'867-5309'}, 'Destiny': {'work': '900-783-3369'}}
>>> print(contacts["Jenny"])
{'cell': '555-0199', 'home': '867-5309'}
>>> print(contacts["Jenny"]["cell"])
555-0199
>>>
```

When we wanted to retrieve a specific value inside the dictionary-value associated with the key for Jenny, we had to say `contacts["Jenny"]["cell"]`. This is because saying `contacts["Jenny"]` now returns the dictionary of *all* numbers for Jenny, from which we have to specify a key for a specific type of phone number.

Finally, there are two alternative ways to create dictionaries that can come in useful in certain specific contexts. You shouldn't worry about learning the details of how to use them right now, but just be aware that they are a possibility.

When you want to use keys that are strings that only include letters and numbers (i.e., strings that could stand for variable names), you can use `dict()` to create a dictionary like so:

```
>>> simple_dict = dict(string1="value1", string2=2, string3=3.0)
>>> print(simple_dict)
{'string2': 2, 'string3': 3.0, 'string1': 'value1'}
>>>
```

Here we created a new dictionary named `simple_dict` that has three string keys, but without putting quotes around the key names this time because Python knows to expect strings when we use `dict()` in this way. We'll see an example of this use of dict() later in the course.

The second way to use `dict()` involves providing a list of key-value pairs represented as tuples, like so:

```
>>> simple_dict = dict([("string1","value1"), ("string2",2),
("string3",3.0)])
>>> print(simple_dict)
{'string2': 2, 'string3': 3.0, 'string1': 'value1'}
>>>
```

Within `dict()`, we include a list (the square brackets), and separated by commas inside that list are tuples (in the pairs of parentheses) that hold each of our key-value pairs. In this case, we aren't limited to simple string keys; we can again assign any sort of keys we want as long as they are all the same type of object.

> **NOTE**: Keep in mind that, even though dictionaries might seem more complicated to use, their main advantage is that they are **very fast**. When you're working with long lists of data, repeatedly cycling through an entire list to find a single piece of information can take a long time; by contrast, looking up the information associated with a dictionary key is almost instantaneous. If you ever find yourself wanting to create multiple lists where items match up across lists based on their ordering, you should probably be using a dictionary instead

Review exercises:

1. Create an empty dictionary named `birthdays`
2. Enter the following data into the dictionary:

   ```
   'Luke Skywalker': '5/24/19'
   'Obi-Wan Kenobi': '3/11/57'
   'Darth Vader': '4/1/41'
   ```

3. Write `if` statements that test to check if 'Yoda' and 'Darth Vader' exist as keys in the dictionary, then enter each of them with birthday value 'unknown' if their name does not exist as a key

4. Display all the key-value pairs in the dictionary, one per line with a space between the name and the birthday, by looping over the dictionary's keys
5. Delete 'Darth Vader' from the dictionary
6. Bonus: Make the same dictionary by using `dict()` and passing in the initial values when you first create the dictionary

Assignment: Capital city loop

Review your state capitals along with dictionaries and while loops!

First, finish filling out the following dictionary with the remaining states and their associated capitals in a file called *capitals.py*. Or you can grab the finished file directly from the exercises folder in the course repository on Github.

```
capitals_dict = {
    'Alabama': 'Montgomery',
    'Alaska': 'Juneau',
    'Arizona': 'Phoenix',
    'Arkansas': 'Little Rock',
    'California': 'Sacramento',
    'Colorado': 'Denver',
    'Connecticut': 'Hartford',
    'Delaware': 'Dover',
    'Florida': 'Tallahassee',
    'Georgia': 'Atlanta',
}
```

Next, write a script that imports the `capitals_dict` variable along with the 'random' package:

```
from capitals import capitals_dict
import random
```

This script should use a while loop to iterate through the dictionary and grab a random state and capital, assigning each to a variable. The user is then asked what the capital of the randomly picked state is. The loop continues forever, asking the user what the capital is, unless the user either answers correctly or types "exit".

If the user answers correctly, "Correct" is displayed after the loop ends. However, if the user exits without guessing correctly, the answer is displayed along with "Goodbye."

> **NOTE**: Make sure the user is not punished for case sensitivity. In other words, a guess of "Denver" is the same as "denver". The same rings true for exiting - "EXIT" and "Exit" are the same as "exit".

Name this file *capital_city_loop.py*.

Assignment: Cats with hats

You have 100 cats.

One day you decide to arrange all your cats in a giant circle. Initially, none of your cats have any hats on. You walk around the circle 100 times, always starting at the same spot, with the first cat (cat # 1). Every time you stop at a cat, you either put a hat on it if it doesn't have one on, or you take its hat off if it has one on.

1. The first round, you stop at every cat, placing a hat on each one.
2. The second round, you only stop at every second cat (#2, #4, #6, #8, etc.).
3. The third round, you only stop at every third cat (#3, #6, #9, #12, etc.).
4. You continue this process until you've made 100 rounds around the cats (e.g., you only visit the 100th cat).

Write a program that simply outputs which cats have hats at the end.

> **NOTE:** This is not an easy problem by any means. Honestly, the code is simple. This problem is often seen on job interviews as it tests your ability to reason your way through a difficult problem. Stay calm. Start with a diagram, and then write pseudo code. Find a pattern. Then code!

Assignment: Reviewing the fundamentals

Let's review functions, loops, lists, dicts, and conditional logic...

1. Copy and paste the code below to IDLE and save it as *fundamental_review.py*.
2. Run the file. All `print` statements in Part 1 return `False` and there is a `TypeError` in Part 2.
3. Modify the variables so that all of the `print` statements return `True`.

```
print "\n# -- part 1 -- #"

# Modify the variable value so that all of the
# `print` statements return `True`.

zero = 1
one = 2
two = [5, 4, 3, 2, 1, 0]
three = "I love Python!"
four = [["P", "y", "t", "h", "o", "n"],["i", "s"],["h", "a", "r", "d"]]
five = {"happy":"birthday"}
six = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
days = ("Fri", "Sat", "Sun")
x, y, seven = days

# DO NOT CHANGE ANYTHING BELOW THIS LINE #
# -------------------------------------- #

print("zero:  {}".format(zero == 0))
print("one:   {}".format(one > 22))
print("two:   {}".format(len(two) == 5))
print("three: {}".format(three[2] == "Python!"))
print("four:  {}".format(
    four[0][5] == 'n' and four[0][0] == "P" and four[2][1] == "u"
)
print("five:  {}".format(five.get("fish") == "chips"))
print("five:  {}".format(len(five) == 3))
print("six:   {}".format(len(six & {2,5,7}) == 2))
print("seven: {}".format(seven == "Wed"))

# -------------------------------------- #

print("\n# -- part 2 -- #")

# Find a value for the `value` variable
# so that all print statements return True.

value = None
```

```
# DO NOT CHANGE ANYTHING BELOW THIS LINE #
# ------------------------------------ #

if type(value) is list:
    print(True)
else:
    print(False)

for x in value:
    if not type(x) is int:
        print(False)
    else:
        print(True)

num = 0
while num < value[2]:
    print(True)
    num += 1

for y in range(value[3]):
    if y in value:
        print(False)

try:
    value[5] = "Cat"
    while True:
        print(False)
except IndexError:
    print(True)

try:
    assert value[3] == -1
except AssertionError:
    print(True)

# -------------------------------------- #
```

Summary

You should use lists when:

1. You have a mixed collection of data,
2. The data needs to be ordered,
3. The data may need to be changed (lists are mutable), and
4. You need a stack or a queue.

You should use tuples when:

1. The data won't need to be changed (tuples are immutable), and
2. Performance is an issue.

You should use dictionaries when:

1. There is a logical association between a key/value pairs,
2. You need to lookup data quickly (via keys), and
3. The data may need to be changed (dictionaries are mutable).

File Input and Output

Read and write simple files

So far, we've allowed the user to type input into our program and displayed output on the screen. But what if we want to work with a lot of data? It's time to learn how to use Python to work with files.

To read or write "raw" text files (i.e., the sort of file that you could use in a basic text editor because it contains no formatting or extra information), we use the general-purpose `open()` function. When we `open()` a file, the first thing we have to determine is if we want to read from it or write to it. Let's start by creating a new text file and writing some data into it:

```
my_output_file = open("hello.txt", "w")
```

We passed two pieces of information (called **arguments**) to the `open()` function. The first argument was a string that represents the actual name of the file we want to create: *hello.txt*. The second argument specifies our purpose for opening the file; in this case, we said `w` because we want to write to the file.

The `open()` function returns a File object that has been saved in the variable `my_output_file`. Let's write a line of text into the file with the `writelines()` method. Our full script should look like this:

```
my_output_file = open("hello.txt", "w")

my_output_file.writelines("This is my first file.")

my_output_file.close()
```

Make sure you know where you are saving this script before you run it; since we didn't specify a directory path for the file, right now *hello.txt* will be created in the same folder as the script.

> **NOTE**: You should always use the `close()` method to close any file that you have `open()` once you're completely done with the file. Python will eventually close any open files when you exit the program, but not closing files yourself can still cause surprising problems. This is because Python often buffers file output, meaning that it might save a bunch of commands you've written (without running them right away), then run them all in a big batch later on to make the process run faster. This could result in something like the following unwanted situation: you write output to a file, then open that file up in a text editor to view the output, but since you didn't `close()` the file in Python (and IDLE is still running), the file is completely blank even though Python is planning to write output to the file before it is closed.

After running this script, you should see a new file named *hello.txt* appear in the same folder as your script; open the output file up yourself to check that it contains the line we wrote.

The `writelines()` method can also take a list of lines to be written all at once. The "lines" will be written one after the other without a new line, so we have to specify the special newline character `\n` if we actually want the lines to appear on separate lines. Let's modify the script to write a couple lines from a list:

```
my_output_file = open("hello.txt", "w")

lines_to_write = ["This is my file.", "\n There are many like it,",
                  "\nbut this one is mine."]

my_output_file.writelines(lines_to_write)
my_output_file.close()
```

Without deleting the previous *hello.txt* file, try running this version of the script, then check the contents of the file. This is an important lesson that's easy to forget: As soon as you `open()` a file in `w` (write) mode, if the file already exists then the file's current contents are completely deleted. It's a common mistake to accidentally overwrite or delete the contents of an important file this way.

If we want to add information to a file instead of overwriting its contents, we can use the `a` mode to append to the end of the file. The rest of the process is identical to writing in `w` mode; the only difference is that we start writing at the end of the file. Again, if we want the new output to appear on a new line, we have to specify the `\n` character to move to a new line. Let's append one additional line onto our current file *hello.txt*:

```
my_output_file = open("hello.txt", "a")
next_line = ["\nNON SEQUITUR"]
my_output_file.writelines(next_line)
my_output_file.close()
```

Now that we have a file written, let's read the data from it. You can probably guess how this goes by now:

```
my_input_file = open("hello.txt", "r")
print(my_input_file.readlines())
my_input_file.close()
```

This time, we used the `r` mode to read input from the file. We then used the `readlines()` method to return every line of the file, which are displayed like so:

```
>>>

['This is my file.\n', 'There are many like it,\n', 'but this one is mine.\n', 'NO
N SEQUITUR']

>>>
```

The output is returned in the form of a list, and all of the line breaks were visible to us as printed newline characters. One common way of working with a file in its entirety is to use a `for` loop:

```
my_input_file = open("hello.txt", "r")

for line in my_input_file.readlines():
    print(line),

my_input_file.close()
```

> **NOTE**: Notice how we ended our `print` statement with a comma; this is because any print statement will usually add a new line to the end of the line of output. The extra comma stops the print statement from adding this automatic `\n` so that the next print statement will continue to display on the same line. Since our file already has new line characters, adding extra new lines would have made the file output display incorrectly, with a blank line appearing in between each actual line of the file.

You can remove the automatic `\n` (or change it to something different) by specifying the "end" parameter of the `print()` function like so:

```
print(line, end="")
```

We supplied empty quotes to get rid of the line break, but we could have put, for instance, an extra line break after every line by passing `end="\n\n"` to the `print()` function.

We can also read lines from the file one at a time using the `readline()` method. Python will keep track of where we are in the file for as long as we have it open, returning the next available line in the file each time `readline()` is called:

```
my_input_file = open("hello.txt", "r")

line = my_input_file.readline()
while line != "":
    print(line),
    line = my_input_file.readline()

my_input_file.close()
```

> **NOTE**: While a file remains open in *read* mode, Python keeps track of the last line that was read. So calling `read()` twice in a row will print the lines the first time and print an empty string the second time. `readline()` and `readlines()` have similar behavior. Once a file is closed, the 'read' position is reset. Play around with these methods in IDLE to get a feel for the differences. As well, there is more on this topic below.

There is an additional shortcut that can be helpful in organizing code when working with files: using Python's `with` keyword. Using `with` to read our file, we could say:

```
with open("hello.txt", "r") as my_input_file:
    for line in my_input_file.readlines():
        print(line),
```

Compare this code carefully to the two previous examples. When we say "with X as Y" we are defining the variable Y to be the result of running X. This begins a block of code where we can use our new variable as usual (in this case, `my_input_file`). The added benefit with using the `with` keyword is that we no longer have to worry about closing the file; once our "with block" is finished, the clean-up work (closing of the file) will be managed for us automatically.

In fact, we can name multiple variables in a `with` statement if we want to open multiple files at once. For instance, if we wanted to read *hello.txt* in and write its contents out into a new file *hi.txt* line-by-line, we could simply say:

```
with open("hello.txt", "r") as my_input, open("hi.txt", "w") as my_output:
    for line in my_input.readlines():
        my_output.write(line)
```

Again, this will take care of all the clean-up work for us, closing both files once we exit the block of code inside the `with` statement. (Of course, practically speaking there's an easier way to accomplish this particular task; the shutil module includes many helpful functions including `copy()`, which can be used to copy an entire file into a new location.)

The rest of the material in this section is conceptually more complicated and usually isn't necessary for most basic file reading/writing tasks. Feel free to skim this remaining material for now and come back to it if you ever find that you need to read or write to a file in a way that involve specific parts of lines rather than taking entire lines from a file one by one.

If we want to visit a specific part of the file, we can use the `seek()` method to jump a particular number of characters into the file. For instance:

```
my_input_file = open("hello.txt", "r")
print "Line 0 (first line):", my_input_file.readline()

my_input_file.seek(0) # jump back to beginning
print("Line 0 again:", my_input_file.readline())
print("Line 1:", my_input_file.readline())

my_input_file.seek(8) # jump to character at index 8
print("Line 0 (starting at 9th character):", my_input_file.readline())

my_input_file.seek(10, 1) # relative jump forward 10 characters
print("Line 1 (starting at 11th character):", my_input_file.readline())

my_input_file.close()
```

Run this script, then follow along with the output as you read the description, since it's not the most intuitive method to use. When we provide a single number to `seek()`, it will go to the character in the file with that index number, regardless of where we currently are in the file. Thus, `seek(0)` always gets us back to the beginning of the file, and `seek(8)` will always place us at the character at index position 8, regardless of what we have done previously. When we provide a second argument of "1" to `seek()`, as in the last example, we are moving forward (or backward, if we use a negative number) relative to where we currently are in the file. In this case, after we displayed line 0 starting at its 9th character, we were currently at the beginning of line 1. Calling `seek(10, 1)` then moved us 10 characters ahead in line 1. Clearly, this sort of seeking behavior is only useful in very specific cases.

Although it's less commonly used, it is possible to open a file for both reading and writing. You may have already guessed why this is usually not a good idea, though; it's typically very difficult to keep track of where you are in a particular file using `seek()` in order to decide which pieces you want to read or write. We can specify the mode `r+` to allow for both reading and writing, or `ra+` to both read and append to an existing file. Since writing or appending will change the characters in the file, however, you will need to perform a new `seek()` whenever switching modes from writing to reading.

Review exercises:

1. Read in the raw text file *poem.txt* from the chapter 10 practice files and display each line by looping over them individually, then close the file; we'll discuss using file paths in the next section, but for now you can save your script in the same folder as the text file
2. Repeat the previous exercise using the `with` keyword so that the file is closed automatically after you're done looping through the lines
3. Write a text file *output.txt* that contains the same lines as *poem.txt* by opening both files at the same time (in different modes) and copying the original file over line-by-line; do this using a loop and closing both files, then repeat this exercise using the `with` keyword
4. Re-open *output.txt* and append an additional line of your choice to the end of the file on a new line

Use More Complicated Folder Structures

Chances are that you don't want to limit yourself to using the same folder as your script for all your files all the time. In order to get access to different directories, we can just add them to the file name to specify a full path. For instance, we could have pointed our script to the following fictitious path:

```
my_input_file = open("C:/My Documents/useless text files/hello.txt", "r")
```

Notice how we used only forward slashes in the path - not backslashes. This method of substituting forward slashes works fine even in Windows, where the operating system's default is to use backslashes to separate directories. We do this to avoid the "**escape character**" problem where Python would have treated a backslash and the character following it as a pair of special characters instead of reading them along with the rest of the string normally. The backslash is called an "escape character" because it lets Python know that the backslash and the character following it should be read as a pair to represent a different character. For instance, `\n` would be interpreted as a newline character and `\t` represents a "tab" character.

Another way to get around this problem is to put a lowercase `r` just before a string, without a space, like so:

```
myName = r"C:\My Documents\useless text files\hello.txt"
```

This creates a "raw" string that is read in exactly as it is typed, meaning that backslashes are only ever read as actual backslash characters and won't be combined with any other characters to create special characters.

In order to do anything more advanced with file structures, we need to rely on a built-in set of Python code called the os module, which gives us access to various functions related to the operating system. So the first thing that we will need to do is `import os` into our code.

If you are used to working in the command line, the os module gives you much of the same basic functionality that will probably already be somewhat familiar - for instance, the `rmdir()` function to delete a directory and the `mkdir()` function to create a new directory.

Soon we'll see how to manipulate and interact with the included example files in various ways. Although there are a number of different ways to set your scripts up correctly for accessing files, for simplicity we will do the following: whenever we need to write a script that makes use of an example file in the course folder, we will start with something like the following code:

```
import os

my_path = "C:/Real Python/Course materials"
```

You should replace the string that gets assigned to the variable path with a string that actually represents the location at which you've saved the main course materials folder (which is named "Real Python/Course materials" by default, but might not be saved directly onto your C: drive). This way, you will only ever have to specify which folders inside of the course folder you want to access instead of typing it out each time you want to access a sample file. We will then join this path to the rest of each file location using the `os.path.join()` function, as we'll see below.

For instance, if you wanted to display the full contents of the example text file named *example.txt* in the chapter 9 practice files folder, the sample code would look like the following:

```
import os

my_path = "C:/book1-exercises/chp09/practice_files"
input_file_name = os.path.join(my_path, "example.txt")

with open(input_file_name, "r") as my_input_file:
    for line in my_input_file.readlines():
        print(line),
```

Again, the string on the second line that represents the path to the Practice files folder might need to be changed if you saved the course files in a different location.

Notice how we used `os.path.join()` as a way of adding the full file path onto the main directory by passing the two parts of the path as arguments to this function. This just combines the two strings together, making sure that the right number of slashes is included in between the two parts. Instead of using `os.path.join()`, we could have simply added (concatenated) the string path to the rest of the file path by using a plus sign and adding an extra forward slash between the two strings like this:

```
input_file_name = my_path + "/example.txt"
```

However, `os.path.join()` comes with the added benefit of Python automatically adding any slashes between the two path strings necessary to create a valid path. This is why it's a good idea to get into the habit of using this function to join path names together; sometimes we will retrieve part of a path names through our code and not know ahead of time if it includes an extra slash or not, and in these cases `os.path.join()` will be a necessity.

> **NOTE**: If you're using Python 3.4+, you also have the option to work with object-oriented paths using pathlib.

Let's start modifying files using a basic practical example: we want to rename every .GIF file in a particular folder to be a .JPG file of the same name. In order to get a list of the files in the folder, we can use the `os.listdir()` function, which returns a list of file names found in the provided directory. We can use the string method `endswith()` to check the file extension of each file name. Finally, we'll use `os.rename()` to rename each file:

```
import os

my_path = "C:/book1-exercises/chp09/practice_files/images"

# get a list of all files and folders
file_names_list = os.listdir(my_path)

# loop over every file in the main folder
for file_name in file_names_list:
    if file_name.lower().endswith(".gif"): # extension matches a GIF file
        print'Converting "{}" to JPG...'.format(file_name)
        # get full path name and change the ".gif" to ".jpg"
        full_file_name = os.path.join(my_path, file_name)
        new_file_name = full_file_name[0:len(full_file_name)-4] +".jpg"
        os.rename(full_file_name, new_file_name)
```

Since `endswith()` is case-sensitive, we had to convert `file_name` to lowercase using the `lower()` method; since this method returns a string as well, we just stacked one method on top of another in the same line. We used subscripting to replace the file extension in the line `new_file_name = full_file_name[0:len(full_file_name)-4]` by trimming the last four characters (the ".gif") off of the full file name, then we added the new ".jpg" extension instead. Our `os.rename()` function took two arguments, the first being the full original file name and the second being the new file name.

A more efficient way of performing this same task would be to import and use the glob module, which serves the purpose of helping to match patterns in file names. The `glob.glob()` function takes a string that uses "wildcard" characters, then returns a list of all possible matches. In this case, if we provide the file name pattern "*.gif" then we will be able to find any file names that match the ".gif" extension at the end:

```
import glob
import os

my_path = "C:/book1-exercises/chp09/practice_files/images"
possible_files = os.path.join(my_path, "*.gif")

for file_name in glob.glob(possible_files):
    print 'Converting "{}" to JPG...'.format(file_name)
    full_file_name = os.path.join(my_path, file_name)
    new_file_name = full_file_name[0:len(full_file_name)-4] +".jpg"
    os.rename(full_file_name, new_file_name)
```

By providing a string with the full file path and an `*`, we were able to return a glob list of all possible GIF images in that particular directory. We can also use glob() to search through subfolders - for instance, if we wanted to search for all of the PNG files that are in folders inside of our images folder, we could search using the string pattern:

```
import glob
import os

my_path = "C:/book1-exercises/chp09/practice_files/images"
possible_files = os.path.join(my_path, "*/*.png")

for file_name in glob.glob(possible_files):
    print(file_name)
```

Adding the string `"*/*.png"` to the path means that we are searching for any files ending in ".png" that are inside of folders that can have any name (the first "*"). Since we used the forward slash to separate the last folder, however, we will only be searching in subfolders of the "images" directory.

> **NOTE**: When you print file paths generated by Python, you may notice that some of them include backslashes. In fact, Python is automatically adding two backslashes everywhere, but only the second backslash of each pair is displayed. This is because the first backslash is still acting as an "escape" character, then the second backslash tells Python that we do in fact want just a backslash character. We could also have specified our own paths this way, for instance: `my_path = "C:\\book1-exercises\\chp09\\practice_files\\images` displays the string correctly (try it out), if we ever use the string in a way that causes Python to reinterpret each of the backslashes in the file path as a double backslash, we'll end up with an invalid path full of doubled double backslashes. This is why it's a good idea to stick with using either forward slashes or "raw" strings for path names.

Another special pattern-matching character that can be included in a glob pattern is a `` ` `` `to stand for any one single character; for instance, searching for anything matching` .gif `would only return GIF files that have a name that is two characters long. We can also include ranges to search over by putting them in square brackets; the pattern` [0-9] `will match any single number from 0 through 9, and the pattern` [a-z] `will match any single letter. For instance, if we wanted to search for any GIF files that have the name "image" followed specifically by two digits, we could pass the pattern` image[0-9][0-9].gif` to glob.

Keep in mind that `listdir()` returns a list of all files and folders in a given folder. Therefore, if we had wanted to affect every file in the image folder, then we would want to be careful not to affect folder names as well. We can check this easily by using either `os.path.isfile()` or `os.path.isdir()`, both of which return `True` or `False`. For instance, if we wanted to add the string "folder" to the end of each folder name inside the images folder but not affect any of the files in images, we could do the following:

```python
import os

my_path = "C:/book1-exercises/chp09/practice_files/images"
files_and_folders = os.listdir(my_path)

for folder_name in files_and_folders:
    full_path = os.path.join(my_path, folder_name)
    if os.path.isdir(full_path):
        os.rename(full_path, full_path + " folder")
```

To rename a folder (or file), we simply passed the original full path string and the new full path string to the `os.rename()` function. Here we used `os.path.isdir()` to decide whether `folder_name` is actually a folder or not; in this case it's either a valid path to a folder or a valid path to a file, but passing any string that isn't a valid folder path to `os.path.isdir()` will return `False`. Another related function that can be especially useful for deciding whether or not a particular file needs to be created for the first time is `os.path.exists()`, which returns `True` or `False` depending on whether the file or folder specified already exists or not.

Sometimes we need to deal with more complicated folder structures - for instance, if we wanted to get all the files in all subfolders of a particular folder. For this, we can use `os.walk()`. This function will return all the possible combinations of (folder, subfolders, file names) as tuples that represent the paths to reach every file anywhere in a named root folder. For instance, if we wanted to display every file within the "images" folder and any of its subfolders, we could do the following:

```
import os

my_path = "C:/book1-exercises/chp09/practice_files/images"

for current_folder, subfolders, file_names in os.walk(my_path):
    for file_name in file_names:
        print(os.path.join(current_folder, file_name))
```

The call to `os.walk()` created an object that Python could loop over, each time returning a different tuple that includes - (1) a particular folder, (2) a list of the subfolders within that folder, and (3) a list of files within that folder. We used **tuple unpacking** in the outer `for` loop in order to get every possible combination of (current_folder, subfolders, file_names) to loop over, where `current_folder` might actually represent the images folder or a subfolder of images. In this case, we don't care about any of the results from subfolders since looping through each of the `file_names` and joining them to `current_folder` will give us the full path to every file.

Although we've covered the most common cases, there are many additional functions belonging to both the os module and the os.path module that can be used in various ways for accessing and modifying files and folders. In the assignment below, we'll practice a couple more of these functions: deleting files and folders by passing them to the `os.remove()` function and getting the size of a file in bytes by passing it to the `os.path.getsize()` function.

Review exercises:

1. Display the full paths of all of the files and folders in the images folder by using `os.listdir()`
2. Display the full paths of any PNG files in the images folder by using `glob.glob()`
3. Rename any PNG files in the images folder and its subfolders to be JPG files by using `os.walk()` ; in case you mess things up beyond repair, there is a copy of the images folder in the backup folder
4. Make sure that your last script worked by using `os.path.exists()` to check that the JPG files now exist (by providing `os.path.exists()` with the full path to each of these files)

Assignment: Use pattern matching to delete files

Write a script *remove_files.py* that will look in the chapter 9 practice files folder named "little pics" as well all of its subfolders. The script should use `os.remove()` to delete any JPG file found in any of these folders if the file is less than 2 Kb (2,000 bytes) in size.

You can supply the `os.path.getsize()` function with a full file path to return the file's size in bytes. Check the contents of the folders before running your script to make sure that you delete the correct files; you should only end up removing the files named "to be deleted.jpg" and "definitely has to go.jpg" - although you should only use the file extensions and file sizes to determine this.

If you mess up and delete the wrong files, there is a folder named "backup" that contains an exact copy of the "little pics" folder and all its contents so that you can copy these contents back and try again.

Read and Write CSV Data

The types of files we have to deal with in our everyday lives are usually more complicated than plain text files. If we want to be able to modify the contents of these files (rather than just copy, rename or delete them), we will need more complex systems for being able to read this information.

One common way of storing text data is in CSV files. "CSV" stands for Comma-Separated Value, because each entry in a row of data is usually separated from other entries by a comma. For example, the contents of the file named *wonka.csv* in the chapter 9 practice materials folder look like this:

```
First name,Last name,Reward
Charlie,Bucket,"golden ticket, chocolate factory"
Veruca,Salt,squirrel revolution
Violet,Beauregarde,fruit chew
```

We have three columns of variables: First name, Last name, and Reward. Each line represents another row of data, including the first row, which is a "header" row that tells us what each entry represents. Our entries have to appear in the same order for each row, with each entry separated from others by commas. Notice how "golden ticket, chocolate factory" is in quotes - this is because it contains a comma, but this comma isn't meant to separate one entry from another. There is no set standard for how to write out CSV files, but this particular file was created with Microsoft Excel, which added the quotation marks around the entry containing a comma.

> **NOTE**: If you open the *wonka.csv* practice file, it will most likely be opened automatically by Excel, OpenOffice Calc, LibreOffice Calc, or a similar program; all of these programs have the ability to read and write CSV data, which is one reason why this format is so useful. As long as you don't need to track characteristics of a data file such as formatting and colors, it's usually easiest to export data to a CSV file before working with the data in Python; the CSV can always be opened in the appropriate program and re-saved as the proper format again later. CSV files can also be useful for importing or exporting data from systems such as SQL databases that we will learn about later.

Python has a built-in csv module that makes it nearly as easy to read and write CSV files as any other sort of text file. Let's start with a basic example and read in our *wonka.csv* file, then display its contents:

```
import csv
import os

my_path = "C:/book1-exercises/chp09/practice_files"

with open(os.path.join(my_path, "wonka.csv"), "r") as my_file:
    my_file_reader = csv.reader(my_file)
    for row in my_file_reader:
        print(row)
```

We opened a file just as we've done before. We then created a CSV file reader using `csv.reader()` and passed it the file. Notice that we had to pass the actual opened file object to `csv.reader()`, not just the file name. From there, we can easily loop over the rows of data in this CSV reader object, which are each displayed as a list of strings:

```
>>>
['First name', 'Last name', 'Reward']
['Charlie', 'Bucket', 'golden ticket, chocolate factory']
['Veruca', 'Salt', 'squirrel revolution']
['Violet', 'Beauregarde', 'fruit chew']
>>>
```

Much like with `readline()` versus `readlines()`, there is also a `next()` method that gets only the next row of data from a CSV reader object. This method is usually used as a simple method of skipping over a row of "header" data; for instance, if we wanted to read in and store all the information except the first line of our CSV file, we could add the line `next(my_file_reader)` after opening the CSV file to skip over the first line, then loop through the remaining rows as usual.

If we know what fields to expect from the CSV ahead of time, we can even unpack them from each row into new variables in a single step:

```
import csv
import os

my_path = "book1-exercises/chp09/practice_files"

with open(os.path.join(my_path, "wonka.csv"), "r") as my_file:
    my_file_reader = csv.reader(my_file)
    next(my_file_reader)
    for first_name, last_name, reward in my_file_reader:
        print("{} {} got: {}".format(first_name, last_name, reward))
```

After skipping the first header row with the `next()` function, we assigned the three values in each row to the three separate strings first_name, last_name and reward, which we then used inside of the `for` loop, generating this output:

```
>>>
['First name', 'Last name', 'Reward']
Charlie Bucket got: golden ticket, chocolate factory
Veruca Salt got: squirrel revolution
Violet Beauregarde got: fruit chew
>>>
```

The first line of this output was generated by the call to the `next()` function rather than a print statement of ours.

The commas in CSV files are called delimiters because they are the character used to separate different pieces of the data. Sometimes a CSV file will use a different character as a delimiter, especially if there are a lot of commas already contained in the data. For instance, let's read in the file *tabbed wonka.csv*, which uses tabs instead of commas to separate entries and looks like this:

```
First name    Last name    Reward
Charlie    Bucket    golden ticket, chocolate factory
Veruca    Salt    squirrel revolution
Violet    Beauregarde    fruit chew
```

We can read files like this using the csv module just as easily as before, but we need to specify what character has been used as the delimiter:

```
import csv
import os

my_path = "book1-exercises/chp09/practice_files"

with open(os.path.join(my_path, "tabbed wonka.csv"), "r") as my_file:
    my_file_reader = csv.reader(my_file, delimiter="\t")
    next(my_file_reader)
    for row in my_file_reader:
        print row
```

Here we used the special character `\t` to mean the "tab" character and assigned it to the argument delimiter when we created `my_file_reader`.

Writing CSV files is accomplished using the `csv.writer()` method in much the same way. Just as rows of data read from CSV files appeared as lists of strings, we first need to structure the rows we want to write as lists of strings:

```
import csv
import os

my_path = "book1-exercises/chp09/practice_files"

with open(os.path.join(my_path, "movies.csv"), "w") as my_file:
    my_file_writer = csv.writer(my_file)
    my_file_writer.writerow(["Movie", "Rating"])
    my_file_writer.writerow(["Rebel Without a Cause", "3"])
    my_file_writer.writerow(["Monty Python's Life of Brian", "5"])
    my_file_writer.writerow(["Santa Claus Conquers the Martians", "0"])
```

We opened a new file in `w` mode this time so that we could write data. We then wrote out individual rows to the CSV file writer object using its `writerow()` method. We also could have used the `writerows()` method, which takes a list of rows, to write all the rows in a single line:

```
import csv
import os

my_path = "book1-exercises/chp09/practice_files"

my_ratings = [ ["Movie", "Rating"],
               ["Rebel Without a Cause", "3"],
               ["Monty Python's Life of Brian", "5"],
               ["Santa Claus Conquers the Martians", "0"] ]

with open(os.path.join(my_path, "movies.csv"), "w") as my_file:
    my_file_writer = csv.writer(my_file)
    my_file_writer.writerows(my_ratings)
```

If we wanted to export data created by a Python script to, for instance, an Excel workbook file, although it's possible to do this directly, it's usually sufficient and much easier to create a CSV file that we can then open later in Excel and, if needed, convert to the desired format. There are a number of special modules that have been designed for interacting with Microsoft Excel documents (although they all have their own limitations), including xlrd and xlwt for reading and writing basic Excel files, openpyxl for manipulating Excel 2010 files, and XlsxWriter for creating .xlsx files from scratch.

Review exercises:

1. Write a script that reads in the data from the CSV file *pastimes.csv* located in the chapter 9 practice files folder, skipping over the header row
2. Display each row of data (except for the header row) as a list of strings
3. Add code to your script to determine whether or not the second entry in each row (the "Favorite Pastime") converted to lower-case includes the word "fighting" using the string methods `find()` and `lower()`
4. Use the list `append()` method to add a third column of data to each row that takes the value "Combat" if the word "fighting" is found and takes the value "Other" if neither word appears
5. Write out a new CSV file *categorized pastimes.csv* to the Output folder with the updated data that includes a new header row with the fields "Name", "Favorite Pastime", and "Type of Pastime"

Assignment: Create a high scores list from CSV data

Write a script *high_scores.py* that will read in a CSV file of users' scores and display the highest score for each person. The file you will read in is named *scores.csv* and is located in the chapter 9 practice files folder. You should store the high scores as values in a dictionary with the associated names as dictionary keys. This way, as you read in each row of data, if the name already has a score associated with it in the dictionary, you can compare these two scores and decide whether or not to replace the "current" high score in the dictionary.

Use the `sorted()` function on the dictionary's keys in order to display an ordered list of high scores, which should match this output:

```
Empiro 23
L33tH4x 42
LLCoolDave 27
MaxxT 25
Misha46 25
O_O 22
johnsmith 30
red 12
tom123 26
```

Assignment: Split a CSV file

Write a script that will take three required command line arguments - `input_file` , `output_file` , and the `row_limit` . From those arguments, split the input CSV into multiple files based on the `row_limit` argument.

Arguments:

1. `-i` : input file name
2. `-o` : output file name
3. `-r` : row limit to split

Default settings:

1. `output_path` is the current directory
2. headers are displayed on each split file
3. the default delimiter is a comma

Example usage:

```
# split csv by every 100 rows
>> python csv_split.py -i input.csv -o output -r 100
```

Before you start coding, stop for a minute and read over the directions again. If you're having trouble following them, take some notes. What makes this assignment so difficult is that it has a number of moving pieces. However, if you can break them down into manageable chunks, then the process will be much easier. Let's look at it together.

1. You first need to grab the command line arguments. I recommend using the argparse library for this. Once obtained, you should validate the arguments to ensure that (a) the input file exists and (b) the number of rows in the input file is greater that the row limit to split. Make sure that each of your functions does only one thing. Think about how many functions you need for this first step.

2. If the validation passes, the program should continue. If not, the program ends, displaying an error message.

3. Next you need to split up the CSV file into separate "chunks" based on the `row_limit` argument. In other words, if your input CSV file has 150 rows (minus the header) and the `row_limit` is set to 50, then there should be three chunks (and

when you create your output CSV files, each chunk will have 50 rows + the header). There's a number of different ways to create each chunk. In this example, it's probably easiest to create a separate `list` for each chunk containing the appropriate # of rows from the input CSV file.

4. You need to have separate output files (one for each chunk) that have some sort of naming convention that makes sense. You could use a timestamp. Or you could add the chunk number to each file name - e.g., *output-file-name_chunk-number.csv*. Each file must have a *.csv* extension as well as the headers. Add each chunk to the appropriate output file.
5. Finally, output information to the user indicating the file name and the # of rows for each chunk. Format this in an appropriate, legible manner.

Try this out on your own before looking at the answer. You should be able to get through the first two steps on your own. The remaining steps are a bit more difficult. If you found this assignment easy, try to add additional functionality to your program, such as the ability to include or exclude the headers from each file, splitting the input CSV by the column # (or name) instead of by row.

Need help? Here are some recommendations for how to break down steps 3, 4, and 5:

In step 3, you should open the CSV file and create a list of lists where each list is a row in the spreadsheet. Remove the header and save it, since you'll need to add it to each chunk. Use a `for` loop to loop through the list of lists and create a chunk that contains the rows from a starting row # to an ending row number. The ending row number is the `row_limit`. If there are not enough rows left - e.g., the rows remaining is less than the `row_limit' - then make sure to add all remaining rows to the final chunk.

Steps 4 and 5 are best kept in the same `for` loop. Create a new, unique output file name, add the headers to each chunk, then add each chunk to the file. Output the information to the user.

Still stuck? Consult Google. Or Stack Overflow.

Interlude: Install Packages

The remaining half of this course relies on functionality found in various toolkits that are not packaged with Python by default. There are over 71,000 of these "extra" packages, as of writing (December 2015), registered with Python and available for download. Although all of the add-on features that we'll cover are widely used and freely available, you will first need to download and install each of these packages in order to be able to import new functionality into your own code.

Python "packages" and "toolkits" are usually just another way of referring to a module or set of modules that can be imported into other Python code, although sometimes these modules rely on other code outside of Python, which is why it can sometimes be tricky to get everything installed correctly.

Some Python packages (especially for Windows) offer automated installers that you can download and run without an extra hassle.

Usually in Linux, installing a Python package is only a matter of searching for the correct package name (e.g., in the Debian package directory), then running the command:

```
$ sudo apt-get python-package-name
```

Installing via pip

However, eventually you will come across a Python package that is not as simple to install for your particular operating system. For this reason, the best way to install most Python packages is through pip, although pip itself can be tricky to install on some configurations. If you're using Python 3.4+, you already have pip installed by default and can use it right away to install new packages with no extra hassle:

```
$ pip3 install python-package-name
```

Installing from Source

Come across a package not found in pip? Most packages will come with a *setup.py* script that will help you to install them into Python. If this is the case, you can follow these steps to install the package:

Windows:

1. Download the .zip file for the package and unzip it into a folder in your user directory (i.e., `C:\Users\yourname`)
2. Double-check that there is a script named *setup.py* in the package folder
3. Open up a command prompt (which should display `C:\Users\yourname>`) and type the command *cd* followed by the package folder name; for instance, if you wanted to install a package in the folder named "beautifulsoup4-4.1.0" then you would enter: `cd beautifulsoup4-4.1.0`
4. To install the package, enter the command: `python setup.py install`

Non-Windows:

1. Download the .tar.gz file for the package and decompress it into a folder in your users directory (i.e., "/home/yourname")
2. Double-check that there is a script named *setup.py* in the package folder
3. In Terminal, type the command *cd* followed by the package folder name; for instance, if you wanted to install a package in the folder named "beautifulsoup4-4.1.0" then you would enter: `cd beautifulsoup4-4.1.0`
4. To install the package, enter the command: `sudo python setup.py install`

If all else fails, you can always download (and unzip) the entire set of files for a given package and copy its folder into the same directory as the script where it is used. Since the package files will be in the same current location as the script, you will be able to find and import them automatically.

Of course, there are a number of problems with this approach - mainly, the package may take up a lot of memory if it's large, and you'll have to copy the entire library over into a new directory every time you write a script in a new folder. If it's a small package (and the license allows you to copy the entire set of source files), however, one benefit is that you can then send this entire folder, complete with your script and the needed library files, to someone else, who would then be able to run your script without having to install the package as well. Despite this minor possible convenience, this approach should usually only be used as a last-ditch effort if all other proper attempts to install a package have failed.

Interact with PDF Files

PDF files have become a sort of necessary evil these days. Despite their frequent use, PDFs are some of the most difficult files to work with in terms of making modifications, combining files, and especially for extracting text information.

Fortunately, there are a few options in Python for working specifically with PDF files. None of these options are perfect solutions, but often you can use Python to completely automate or at least ease some of the pain of performing certain tasks using PDFs.

The most frequently used package for working with PDF files in Python is named PyPDF2 and can be found here. You will need to download and install this package before continuing with the chapter. In you terminal or command line type:

```
$ pip install PyPDF2
```

If that doesn't work, you will need to download and unzip the .tar.gz file and install the module using the *setup.py* script as explained in the previous chapter on installing packages from source.

Debian/Linux: Just type the command: `sudo apt-get install python-PyPDF2`

The PyPDF2 package includes a PdfFileReader and a PdfFileWriter; just like when performing other types of file input/output, reading and writing are two entirely separate processes.

First, let's get started by reading in some basic information from a sample PDF file, the first couple chapters of Jane Austen's Pride and Prejudice via Project Gutenberg:

```
import os
from PyPDF2 import PdfFileReader

path = "C:/book1-exercises/chp11/practice_files"

input_file_name = os.path.join(path, "Pride and Prejudice.pdf")
input_file = PdfFileReader(open(input_file_name, "rb"))

print("Number of pages:", input_file.getNumPages())
print("Title:", input_file.getDocumentInfo().title)
```

> **NOTE**: Be sure to update the `path` variable with the correct path for you system.
>
> **NOTE**: Because we are reading PDF files, we must use the argument 'rb' with the `open()` method.

We created a PdfFileReader object named `input_file` by passing a `file()` object with `rb` (read binary) mode and giving the full path of the file. The additional "binary" part is necessary for reading PDF files because we aren't just reading basic text data. PDFs include much more complicated information, and saying "rb" here instead of just "r" tells Python that we might encounter characters that can't be represented as standard readable text.

We can then return the number of pages included in the PDF input file. We also have access to certain attributes through the `getDocumentInfo()` method; in fact, if we display the result of simply calling this method, we will see a dictionary with all of the available document info:

```
>>> print(input_file.getDocumentInfo())
{'/CreationDate': u'D:20110812174208', '/Author': u'Chuck', '/Producer':
u'MicrosoftÃ,Â® Office Word 2007', '/Creator': u'MicrosoftÃ,Â® Office Word 2007',
'/ModDate': u'D:20110812174208', '/Title': u'Pride and Prejudice, by Jane
Austen'}
>>>
```

We can also retrieve individual pages from the PDF document using the `getPage()` method and specifying the index number of the page (as always, starting at 0). However, since PDF pages include much more than simple text, displaying the text data on a PDF page is more involved. Fortunately, PyPDF2 has made the process of parsing out text somewhat easier, and we can use the `extractText()` method on each page:

```
>>> print(input_file.getPage(0).extractText())
  The Project Gutenberg EBook of Pride and Prejudice, by Jane Austen  This eBook i
s for the use of anyone anywhere at no cost and with almost no
restrictions whatsoever.  You may copy it, give it away or re-use it under the ter
ms of the Project Gutenberg License included with this eBook or online at www.gute
nberg.org
Title: Pride and Prejudice  Author: Jane Austen  Release
Date: August 26, 2008 [EBook #1342] [Last updated: August 11, 2011]  Language:
English  Character set encoding: ASCII  *** START OF THIS PROJECT GUTENBERG
EBOOK PRIDE AND PREJUDICE ***     Produced by Anonymous Volunteers, and David
Widger       PRIDE AND PREJUDICE    By Jane Austen     Contents
>>>
```

Formatting standards in PDFs are inconsistent at best, and it's usually necessary to take a look at the PDF files you want to use on a case-by-case basis. In this instance, notice how we don't actually see newline characters in the output; instead, it appears that new lines are being represented as multiple spaces in the text extracted by PyPDF2. We can use this knowledge to write out a roughly formatted version of the book to a plain text file (for instance, if we only had the PDF available and wanted to make it readable on an untalented (err dumb) mobile device):

```
import os
from PyPDF2 import PdfFileReader

path = "C:/book1-exercises/chp11/practice_files"

input_file_name = os.path.join(path, "Pride and Prejudice.pdf")
input_file = PdfFileReader(file(input_file_name, "rb"))

output_file_name = os.path.join(path, "Output/Pride and Prejudice.txt")
output_file = open(output_file_name, "w")

title = input_file.getDocumentInfo().title # get the file title
total_pages = input_file.getNumPages() # get the total page count

output_file.write(title + "\n")
output_file.write("Number of pages: {}\n\n".format(total_pages))

for page_num in range(0, total_pages):
    text = input_file.getPage(page_num).extractText()
    text = text.replace("  ", "\n")
    output_file.write(text)

output_file.close()
```

Since we're writing out basic text, we chose the plain `w` mode and created a file *book.txt* in the "Output" folder. Meanwhile, we still use `rb` mode to read data from the PDF file since, before we can extract the plain text from each page, we are in fact reading much more complicated data. We loop over every page number in the PDF file, extracting the text from that page. Since we know that new lines will show up as additional spaces, we can approximate better formatting by replacing every instance of double spaces (`"  "`) with a newline character.

Instead of extracting text, we might want to modify the PDF file itself, saving out a new version of the PDF. We'll see more examples of why and how this might occur in the next section, but for now create the simplest "modified" file by saving out only a section of the original file. Here we copy over the first three pages of the PDF (not including the cover page) into a new PDF file:

```
import os
from PyPDF2 import PdfFileReader, PdfFileWriter

path = "C:/book1-exercises/chp11/practice_files"

input_file_name = os.path.join(path, "Pride and Prejudice.pdf")
input_file = PdfFileReader(open(input_file_name, "rb"))
output_PDF = PdfFileWriter()

for page_num in range(1, 4):
    output_PDF.addPage(input_file.getPage(page_num))

output_file_name = os.path.join(path, "Output/portion.pdf")
output_file = open(output_file_name, "wb")
output_PDF.write(output_file)
output_file.close()
```

We imported both `PdfFileReader` and `PdfFileWriter` from `PyPDF2` so that we can write out a PDF file of our own. `PdfFileWriter` doesn't take any arguments, which might be surprising; we can start adding PDF pages to our `output_PDF` before we've specified what file it will become. However, in order to save the output to an actual PDF file, at the end of our code we create an `output_file` as usual and then call `output_PDF.write(output_file)` in order to write the PDF contents into this file.

Review exercises:

1. Write a script that opens the file named *The Whistling Gypsy.pdf* from the Chapter 11 practice files, then displays the title, author, and total number of pages in the file
2. Extract the full contents of *The Whistling Gypsy.pdf* into a .TXT file
3. Save a new version of *The Whistling Gypsy.pdf* that does not include the cover page into the "Output" folder

Manipulate PDF Files

Often the reason we want to modify a PDF file is more complicated than just saving a portion of the file. We might want to rotate some pages, crop pages, or even merge information from different pages together. When manually editing the files, Adobe Acrobat isn't a practical or feasible solution; however, we can automate many of these tasks using PyPDF2.

Let's start with a surprisingly common problem: rotated PDF pages. Go ahead and open up the file *ugly.pdf* in the "book1-exercises/chp11/practice_files/" folder. You'll see that it's a lovely PDF file of Hans Christian Andersen's The Ugly Duckling, except that every odd-numbered page is rotated counterclockwise by ninety degrees. This is simple enough to correct by using the rotateClockwise() method on every other PDF page and specifying the number of degrees to rotate:

```
import os
from PyPDF2 import PdfFileReader, PdfFileWriter

path = "C:/book1-exercises/chp11/practice_files"
input_file_name = os.path.join(path, "ugly.pdf")
input_file = PdfFileReader(open(input_file_name, "rb"))
output_PDF = PdfFileWriter()

for page_num in range(0, input_file.getNumPages()):
    page = input_file.getPage(page_num)
    if page_num % 2 == 0:
        page.rotateClockwise(90)
    output_PDF.addPage(page)

output_file_name = os.path.join(path, "Output/The Conformed Duckling.pdf")
output_file = open(output_file_name, "wb")
output_PDF.write(output_file)
output_file.close()
```

Another useful feature of PyPDF2 is the ability to crop pages, which in turn will allow us to split up PDF pages into multiple parts or save out partial sections of pages. For instance, open up the file *half and half.pdf* from the chapter 11 practice files folder to see an example of where this might be useful. This time, we have a PDF that's presented in two "frames" per page, which again is not an ideal layout in many situations. In order to split these pages up, we will have to refer to the MediaBox belonging to each PDF page, which is a rectangle representing the boundaries of the page. Let's take a look at the MediaBox of a PDF page in the interactive window to get an idea of what it looks like:

```
>>> from PyPDF2 import PdfFileReader
>>> input_file = PdfFileReader(open("C:/book1-exercises/chp11/practice_files/half
and half.pdf", "rb"))
>>> page = input_file.getPage(0)
>>> print(page.mediaBox)
RectangleObject([0, 0, 792, 612])
>>>
```

A mediaBox is a type of object called a `RectangleObject`. Consequently, we can get the coordinates of the rectangle's corners:

```
>>> print(page.mediaBox.lowerLeft)
(0, 0)
>>> print(page.mediaBox.lowerRight)
(792, 0)
>>> print(page.mediaBox.upperRight)
(792, 612)
>>> print(page.mediaBox.upperRight[0])
792.0
>>> print(page.mediaBox.upperRight[1])
612.0
>>>
```

These locations are returned to us as tuples that include the x and y coordinate pairs. Notice how we didn't include parentheses anywhere because `mediaBox` and its corners are unchangeable attributes, not methods of the PDF page.

We will have to do a little math in order to crop each of our PDF pages. Basically, we need to set the corners of each half-page so that we crop out the side of the page that we don't want. To do this, we divide the width of the landscape page into two halves; we set the right corner of the left-side page to be half of the total width, and we set the left corner of the right-side page to start halfway across the width of the page.

Since we have to crop the half-pages in order to write them out to our new PDF file, we will also have to create a copy of each page. This is because the PDF pages are mutable objects; if we change something about a page, we also change the same things about any variable that references that object. This is exactly the same problem that we ran into when having to copy an entire list into a new list before making changes. In this case, we import the built-in `copy` module, which creates and returns a copy of an object by using the `copy.copy()` function. (In fact, this function works just as well for making copies of entire lists instead of the shorthand list2 = list1[:] notation.)

This is tricky code, so take a while to work through it and play with different variations of the copying and cropping to make sure you understand the underlying math:

```
import os
import copy
from PyPDF2 import PdfFileReader, PdfFileWriter

path = "C:/book1-exercises/chp11/practice_files"

input_file_name = os.path.join(path, "half and half.pdf")
input_file = PdfFileReader(open(input_file_name, "rb"))
output_PDF = PdfFileWriter()

for page_num in range(0, input_file.getNumPages()):
    page_left = input_file.getPage(page_num)
    page_right = copy.copy(page_left)
    upper_right = page_left.mediaBox.upperRight # get original page corner
    # crop and add left-side page
    page_left.mediaBox.upperRight = (upper_right[0]/2, upper_right[1])
    output_PDF.addPage(page_left)
    # crop and add right-side page
    page_right.mediaBox.upperLeft = (upper_right[0]/2, upper_right[1])
    output_PDF.addPage(page_right)

output_file_name = os.path.join(path, "Output/The Little Mermaid.pdf")
output_file = open(output_file_name, "wb")
output_PDF.write(output_file)
output_file.close()
```

> **NOTE**: PDF files are a bit unusual in how they save page orientation. Depending on how the PDF was originally created, it might be the case that your axes are switched - for instance, a standard "portrait" document that's been converted into a landscape PDF might have the x-axis represented vertically while the y-axis is horizontal. Likewise, the corners would all be rotated by 90 degrees; the upper left corner would appear on the upper right or the lower left, depending on the file's rotation. Especially if you're working with a landscape PDF file, it's best to do some initial testing to make sure that you are using the correct corners and axes.

Beyond manipulating an already existing PDF, we can also add our own information by merging one PDF page with another. For instance, perhaps we want to automatically add a header or a watermark to every page in a file. I've saved an image with a transparent background into a one-page PDF file for this purpose, which we can use as a watermark, combining this image with every page in a PDF file by using the `mergePage()` method:

```
import os
from PyPDF2 import PdfFileReader, PdfFileWriter

path = "C:/book1-exercises/chp11/practice_files"

input_file_name = os.path.join(path, "The Emperor.pdf")
input_file = PdfFileReader(open(input_file_name, "rb"))
output_PDF = PdfFileWriter()

watermark_file_name = os.path.join(path, "top secret.pdf")
watermark_file = PdfFileReader(file(watermark_file_name, "rb"))

for page_num in range(0, input_file.getNumPages()):
    page = input_file.getPage(page_num)
    page.mergePage(watermark_file.getPage(0)) # add watermark image
    output_PDF.addPage(page)

output_PDF.encrypt("good2Bking") # add a password to the PDF file
output_file_name = os.path.join(path, "Output/New Suit.pdf")
output_file = open(output_file_name, "wb")
output_PDF.write(output_file)
output_file.close()
```

While we were securing the file, notice that we also added basic encryption by supplying the password "good2Bking" through the PdfFileWriter's `encrypt()` method. If you know the password used to protect a PDF file, there is also a matching `decrypt()` method to decrypt an input file that is password protected; this can be incredibly useful as an automation tool if you have many identically encrypted PDFs and don't want to have to type out a password each time you open one of the files.

Review exercises:

1. Write a script that opens the file named *Walrus.pdf* from the Chapter 11 practice files; you will need to decrypt the file using the password "IamtheWalrus"
2. Rotate every page in this input file counter-clockwise by 90 degrees
3. Split each page in half vertically, such that every column appears on its own separate page, and output the results as a new PDF file in the Output folder

Assignment: Add a cover sheet to a PDF file

Write a script *cover_the_emperor.py* that appends the chapter 11 practice file named *The Emperor.pdf* to the end of the chapter 11 practice file named *Emperor cover sheet.pdf* and outputs the full resulting PDF to the file *The Covered Emperor.pdf* in the chapter 11 practice files "Output" folder.

Create PDF Files

Although PyPDF2 is one of the best and most frequently relied-upon packages for interacting with PDFs in Python, it does have some weaknesses. For instance, there is no way to generate your own PDF files from scratch; instead, you must start with at least a template document. For PDF generation in particular, I suggest using the ReportLab toolkit, which has a free, open-source version. They also have a number of examples on their code snippets page that you can use.

That said, we'll be using ReportLab for creating files. So start by downloading it via pip3: `pip3 install reportlab`.

Let's start with creating a basic PDF document. Create a new file called *basic_pdf.py* and add the following code:

```
from reportlab.pdfgen import canvas

c = canvas.Canvas("hello.pdf")
c.drawString(100, 100, "Hello World")
c.save()
```

Take a look at the file. Simple, right? Take note of:

1. By supplying just the PDF name, *hello.pdf*, the file is created in the same directory that the script is run from. You can supply a relative or absolute path, along with the filename, to have the file created in a different directory.
2. `drawString` takes three arguments: points from the left margin, points from the bottom of the page, text to be written. 1 point = 1/72 inch.

So, if we updated that file to-

```
from reportlab.pdfgen import canvas

c = canvas.Canvas("hello.pdf")
# c.drawString(100, 100, "Hello World")
c.drawString(250, 500, "Hello World")
c.save()
```

-the text is close to being centered.

You can also specify the `pagesize` as letter, which defaults to A4, and use inches to define where the text is written/drawn:

```
from reportlab.pdfgen import canvas
from reportlab.lib.pagesizes import letter
from reportlab.lib.units import inch

xmargin = 3.2 * inch
ymargin = 6 * inch

c = canvas.Canvas("hello_again.pdf", pagesize=letter)

c.setLineWidth(1)
c.drawString(xmargin, ymargin, "Hello World from ReportLab!")

c.save()
```

Now let's look at something a bit more complicated:

```
from reportlab.pdfgen import canvas
from reportlab.lib.pagesizes import letter
from reportlab.lib.units import inch
from reportlab.lib import colors
from reportlab.platypus import Table

xmargin = 3.2 * inch
ymargin = 6 * inch

c = canvas.Canvas("tps_report.pdf", pagesize=letter)
c.setFont('Helvetica', 12)

data = [['#1', '#2', '#3', '#4', '#5'],
        ['10', '11', '12', '13', '14'],
        ['20', '21', '22', '23', '24'],
        ['30', '31', '32', '33', '34'],
        ['20', '21', '22', '23', '24'],
        ['20', '21', '22', '23', '24'],
        ['20', '21', '22', '23', '24'],
        ['20', '21', '22', '23', '24']]

t = Table(data)
t.setStyle([('TEXTCOLOR', (0,0), (4,0), colors.red)])
t.wrapOn(c, xmargin, ymargin)
t.drawOn(c, xmargin, ymargin)
c.save()
```

Here we created a basic table (or TPS report). The main difference is that we're using a table, obviously:

```
t = Table(data)
t.setStyle([('TEXTCOLOR', (0,0), (4,0), colors.red)])
t.wrapOn(c, xmargin, ymargin)
t.drawOn(c, xmargin, ymargin)
c.save()
```

Put simply, we're creating the table, styling the header using coordinate ranges and then using the `wrapOn` and `drawOn` functions to add the table to the PDF file. For more on tables, check out Chapter 7 in the ReportLab reference docs.

Experiment around with ReportLab. For example, try creating an invoice or adding a grid to the table we created or importing data from a CSV file. Find examples on Github or StackOverflow and work with them. Add your own data. Figure out ways that you can automate the creation of certain reports or letters. Maybe you need to send out 1,000 letters where the only difference is the name. How can you automate the creation of these using ReportLab?

SQL Database Connections

If you're interested in this chapter, I'm assuming that you have at least a basic knowledge of SQL and the concept of querying a database. If not, you might want to take a moment to read through this article introducing databases and browse through these lessons introducing basic SQL code.

The second Real Python course, *Web Development with Python*, provides a much broader overview of SQL programming as well as numerous examples and assignments.

There are many different variations of SQL, and some are suited to certain purposes better than others. The simplest, most lightweight version of SQL is SQLite, which runs directly on your machine and comes bundled with Python automatically.

SQLite is usually used within applications for small internal storage tasks, but it can also be useful for testing SQL code before setting an application up to use a larger database.

In order to communicate with SQLite, we need to import the module and connect to a database:

```
import sqlite3

connection = sqlite3.connect("test_database.db")
```

Here we've created a new database named *test_database.db*, but connecting to an existing database works exactly the same way. Now we need a way to communicate across the connection:

```
c = connection.cursor()
```

This line creates a `Cursor` object, which will let us execute commands on the SQL database and return the results. We'll be using the cursor a lot, so we can just call it `c` for short. Now we easily execute regular SQL statements on the database through the cursor like so:

```
c.execute("CREATE TABLE People(FirstName TEXT, LastName TEXT, Age INT)")
```

This line creates a new table named `People` and inserts three new columns into the table: text to store each person's `FirstName`, another text field to store the `LastName`, and an integer to store `Age`. We can insert data into this new table like so:

```
c.execute("INSERT INTO People VALUES('Ron', 'Obvious', 42)")
connection.commit()
```

Here we've inserted a new row, with a `FirstName` of Ron, a `LastName` of Obvious, and an `Age` equal to 42. In the second line, we had to commit the change we made to the table to say that we really meant to change the table's contents - otherwise our change wouldn't actually be saved.

> **NOTE**: We used double quotation marks in the string above, with single quotes denoting strings inside of the SQL statement. Although Python doesn't differentiate between using single and double quotes, some versions of SQL (including SQLite) only allow strings to be enclosed in single quotation marks, so it's important not to switch these around.

At this point, you could close and restart IDLE completely, and if you then reconnect to *test_database.db*, your `People` table will still exists there, storing Ron Obvious and his Age; this is why SQLite can be useful for internal storage for those times when it makes sense to structure your data as a database of tables rather than writing output to individual files. The most common example of this is to store information about users of an application.

> **NOTE**: If you just want to create a one-time-use database while you're testing code or playing around with table structures, you can use the special name :memory: to create the database in temporary RAM like so: `connection = sqlite3.connect(':memory:')`

If we want to delete the `People` table, it's as easy as executing a `DROP TABLE` statement:

```
c.execute("DROP TABLE IF EXISTS People")
```

(Here we also checked if the table exists before trying to drop it, which is usually a good idea; it helps to avoid errors if we happened to try deleting a table that's already been deleted or never actually existed in the first place.)

Once we're done with a database connection, we should `close()` the connection; just like closing files, this pushes any changes out to the database and frees up any resources currently in memory that are no longer needed. You close the database connection in the same way as with files:

```
connection.close()
```

When working with a database connection, it's also a good idea to use the `with` keyword to simplify your code (and your life), similar to how we used `with` to open files:

```
with sqlite3.connect("test_database.db") as connection:
    # perform any SQL operations using connection here
```

Besides making your code more compact, this will benefit you in a few important ways. Firstly, you no longer need to `commit()` changes you make; they're automatically saved. Using `with` also helps with handling potential errors and freeing up resources that are no longer needed, much like how we can open (and automatically close) files using the `with` keyword. Keep in mind, however, that you will still need to `commit()` a change if you want to see the result of that change immediately (before closing the connection); we'll see an example of this later in the section.

If you want to run more than one line of SQL code at a time, there are a couple possible options. One simple option is to use the `executescript()` method and give it a string that represents a full script; although lines of SQL code will be separated by semicolons, it's common to pass a multi-line string for readability. Our full code might look like so:

```
import sqlite3

with sqlite3.connect('test_database.db') as connection:
    c = connection.cursor()
    c.executescript("""
        DROP TABLE IF EXISTS People;
        CREATE TABLE People(FirstName TEXT, LastName TEXT, Age INT);
        INSERT INTO People VALUES('Ron', 'Obvious', '42');
""")
```

We can also execute many similar statements by using the `executemany()` method and supplying a tuple of tuples, where each inner tuple supplies the information for a single command. For instance, if we have a lot of people's information to insert into our `People` table, we could save this information in the following tuple of tuples:

```
people_values = (
           ('Ron', 'Obvious', 42),
           ('Luigi', 'Vercotti', 43),
           ('Arthur', 'Belling', 28)
)
```

We could then insert all of these people at once (after preparing our connection and our `People` table) by using the single line:

```
c.executemany("INSERT INTO People VALUES(?, ?, ?)", people_values)
```

Here, the question marks act as place-holders for the tuples in `people_values`; this is called a **parameterized statement**. The difference between parameterized and non-parameterized code is very similar to how we can write out strings by concatenating many parts together versus using the string `format()` method to insert specific pieces into a string after creating it.

For security reasons, especially when you need to interact with a SQL table based on user-supplied input, you should always use parameterized SQL statements. This is because the user could potentially supply a value that looks like SQL code and causes your SQL statement to behave in unexpected ways. This is called a "SQL injection" attack and, even if you aren't dealing with a malicious user, it can happen completely by accident.

For instance, suppose we want to insert a person into our `People` table based on user-supplied information. We might initially try something like the following (assuming we already have our People table set up):

```
import sqlite3

# get person data from user
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
age = int(input("Enter your age: "))

# execute insert statement for supplied person data
with sqlite3.connect('test_database.db') as connection:
    c = connection.cursor()
    line = "INSERT INTO People Values('" + first_name + "','" +
        last_name + "'," + str(age) + ")"
    c.execute(line)
```

Notice how we had to change age into an integer to make sure that it was a valid age, but then we had to change it back into a string in order to concatenate it with the rest of the line; this is because we created the line by adding a bunch of strings together, including using single quotation marks to denote strings within our string. If you're still not clear how this works, try inserting a person into the table and then `print` the line to see how the full line of SQL code looks.

But what if the user's name included an apostrophe? Try adding Flannery O'Connor to the table, and you'll see that she breaks the code; this is because the apostrophe gets mixed up with the single quotes in the line, making it appear that the SQL code ends earlier than expected.

In this case, our code only causes an error (which is bad) instead of corrupting the entire table (which would be very bad), but there are many other hard-to-predict cases that can break SQL tables when not parameterizing your statements. To avoid this, we should have used place-holders in our SQL code and inserted the person data as a tuple:

```
import sqlite3

# get person data from user and insert into a tuple
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
age = int(input("Enter your age: "))
person_data = (first_name, last_name, age)

# execute insert statement for supplied person data
with sqlite3.connect('test_database.db') as connection:
    c = connection.cursor()
    c.execute("INSERT INTO People VALUES(?, ?, ?)", person_data)
```

We can also update the content of a row by using a `SQL UPDATE` statement. For instance, if we wanted to change the `Age` associated with someone already in our `People` table, we could say the following (for a cursor within a connection):

```
c.execute("UPDATE People SET Age=? WHERE FirstName=? AND LastName=?",
    (45, 'Luigi', 'Vercotti'))
```

Of course, inserting and updating information in a database isn't all that helpful if we can't fetch that information back out. Just like with `readline()` and readlines() when reading files, there are two available options; we can either retrieve all the results of a

SQL query, using `fetchall()`, or retrieve a single result at a time, using `fetchone()`. First, let's insert some people into a table and then ask SQL to retrieve information from some of them:

```
import sqlite3

people_values = (
           ('Ron', 'Obvious', 42),
           ('Luigi', 'Vercotti', 43),
           ('Arthur', 'Belling', 28)
)

with sqlite3.connect('test_database.db') as connection:
    c = connection.cursor()
    c.execute("DROP TABLE IF EXISTS People")
    c.execute("CREATE TABLE People(FirstName TEXT, LastName TEXT, Age INT)")
    c.executemany("INSERT INTO People VALUES(?, ?, ?)", people_values)
    # select all first and last names from people over age 30
    c.execute("SELECT FirstName, LastName FROM People WHERE Age > 30")
    for row in c.fetchall():
        print(row)
```

We executed a `SELECT` statement that returned the first and last names of all people over the age of 30, then called `fetchall()` on our cursor to retrieve the results of this query, which are stored as a list of tuples. Looping over the rows in this list to view the individual tuples, we see:

```
>>>
('Ron', 'Obvious')
('Luigi', 'Vercotti')
>>>
```

If we wanted to loop over our result rows one at a time instead of fetching them all at once, we would usually use a loop such as the following:

```
c.execute("SELECT FirstName, LastName FROM People WHERE Age > 30")

while True:
    row = c.fetchone()
    if row is None:
        break
    print(row)
```

This checks each time whether our `fetchone()` returned another row from the cursor, displaying the row if so and breaking out of the loop once we run out of results.

The `None` keyword is the way that Python represents the absence of any value for an object. When we wanted to compare a string to a missing value, we used empty quotes to check that the string object had no information inside: `stringName == ""`

When we want to compare other objects to missing values to see if those objects hold any information, we compare them to `None`, like so: `objectName is None`

This comparison will return `True` if `objectName` exists but is empty and `False` if `objectName` holds any value.

> **NOTE**: `None` is a distinct data type in Python. Test this out. Open the Shell and assign `x = None`, then check `type(x)`. You'll see that `x`'s type is `<type 'NoneType'>`.

Review exercises:

1. Create a database table in RAM named `Roster` that includes the fields `Name`, `Species` and `IQ`
2. Populate your new table with the following values:

```
Jean-Baptiste Zorg,    Human,    122
Korben Dallas,     Meat Popsicle,    100
Ak'not,    Mangalore,    -5
```

3. Update the Species of Korben Dallas to be Human
4. Display the names and IQs of everyone in the table who is classified as Human

Use Other SQL Variants

If you have a particular type of SQL database that you'd like to access through Python, most of the basic syntax is likely to be identical to what you just learned for SQLite. However, you'll need to install an additional package in order to interact with your database since SQLite is the only built-in option. There are many SQL variants and corresponding Python packages available. A few of the most commonly used and reliable open-source alternatives are:

1. The pyodbc module allows connections to most types of databases.
2. Specifically for use with PostgreSQL: Psycopg is one of the most frequently used package to interface between Python and PostgreSQL. A Windows version, win-psycopg, is also available.
3. Specifically for use with MySQL: MySQLdb offers MySQL support for Python. Windows users might want to try the myPySQL extension instead, made specifically for Windows.

One difference from SQLite (besides the actual syntax of the SQL code, which changes slightly with most flavors of SQL) is that when you connect to any other SQL database, you'll need to supply the user and password (as well as your host, if the database is hosted on an external server). Check the documentation for the particular package you want to use to figure out the exact syntax for how to make a database connection.

Again, check out the second Real Python course, *Web Development with Python,* for more SQL examples and lessons in SQLite and other SQL flavors.

Interacting with the Web

Given the hundreds of millions of websites out there, chances are that you might at some point be interested in gathering data from a webpage - or perhaps from thousands of webpages. In this chapter we will explore various options for interacting with and gathering data from the Internet through Python.

The second Real Python course, *Web Development with Python*, provides much more in the way of web scraping, utilizing a more powerful method.

Scrape and Parse Text From Websites

Collecting data from websites using an automated process is known as web scraping. Some websites explicitly forbid users from scraping their data with automated tools like the ones we will create. Websites do this for either of two possible reasons:

1. The site has a good reason to protect its data; for instance, Google Maps will not allow you to request too many results too quickly
2. Making many repeated requests to a website's server may use up bandwidth, slowing down the website for other users and potentially overloading the server such that the website stops responding entirely

> **WARNING:** You should always check a website's acceptable use policy before scraping its data to see if accessing the website by using automated tools is a violation of its terms of use. Legally, web scraping against the wishes of a website is very much a gray area, but I just want to make it clear that the following techniques may be illegal when used on websites that prohibit web scraping.

The primary language of information on the Internet is HTML (HyperText Markup Language), which is how most webpages are displayed in browsers. For instance, if you browse to a particular website and choose to "view page source" in your browser, you will most likely be presented with HTML code underlying that webpage; this is the information that your browser receives and translates into the page you actually see.

If you are not familiar with the basics of HTML tags, you should take a time to read through the first dozen chapters of a brief overview of HTML. None of the HTML used in this chapter will be very complicated, but having a solid understanding of HTML elements is an important prerequisite for developing good techniques for web scraping.

The second Real Python course, *Web Development with Python*, provides an overview of HTML as well.

Let's start by grabbing all of the HTML code from a single webpage. We'll take a very simple page that's been set up just for practice:

```
from urllib.request import urlopen

my_address = "https://realpython.com/practice/aphrodite.html"
html_page = urlopen(my_address)

html_text = html_page.read().decode('utf-8')

print html_text
```

This displays the following result for us, which represents the full HTML of the page just as a web browser would see it:

```
>>>
<html>
  <head>
  <title>Profile: Aphrodite</title>
  </head>
  <body bgcolor="yellow">
    <center>
      <br><br>
      <img src="aphrodite.gif" />
      <h2>Name: Aphrodite</h2>
      <br><br>
      Favorite animal: Dove
      <br><br>
      Favorite color: Red
      <br><br>
      Hometown: Mount Olympus
    </center>
  </body>
</html>
>>>
```

Calling `urlopen()` will cause the following error if Python cannot connect to the Internet: `URLError: <urlopen error [Errno 11001] getaddrinfo failed>`

If you provide an invalid web address that can't be found, you will see the following error, which is equivalent to the "404" page that a browser would load: `HTTPError: HTTP Error 404: Not Found`

Now we can scrape specific information from the webpage using text parsing - i.e., looking through the full string of text and grabbing only the pieces that are relevant to us. For instance, if we wanted to get the title of the webpage (in this case, "Profile: Aphrodite"), we could use the string `find()` method to search through the text of the HTML for the `<title>` tags and parse out the actual title using index numbers:

```
from urllib.request import urlopen

my_address = "https://realpython.com/practice/aphrodite.html"
html_page = urlopen(my_address)

html_text = html_page.read().decode('utf-8')

start_tag = "<title>"
end_tag = "</title>"

start_index = html_text.find(start_tag) + len(start_tag)
end_index = html_text.find(end_tag)

print(html_text[start_index:end_index])
```

Running this script correctly displays the HTML code limited to only the text in the title:

```
>>>
Profile: Aphrodite
>>>
```

Of course, this worked for a very simple example, but HTML in the real world can be much more complicated and *far* less predictable. For a small taste of the "expectations versus reality" of text parsing, visit poseidon.html and view the HTML source code. It looks like the same layout as before, but let's try running the same script as before on `my_address = "https://realpython.com/practice/poseidon.html"`:

```
>>>
<head>
<title >Profile: Poseidon
>>>
```

We didn't manage to find the beginning of the `<title>` tag correctly this time because there was a space before the closing ">", like so: `<title >`. Instead, our `find()` method returned -1 (because the exact string `<title>` wasn't found anywhere) and then added the length of the tag string, making us think that the beginning of the title was six characters into the HTML code.

Because these sorts of problems can occur in countless unpredictable ways, a more reliable alternative than using `find()` is to use regular expressions. Regular expressions (shortened to "regex" in Python) are strings that can be used to determine whether or not text matches a particular pattern.

> **NOTE:** Regular expressions are not particular to Python; they are a general programming concept that can be used with a wide variety of programming languages. Regular expressions use a language all of their own that is notoriously difficult to learn but incredibly useful once mastered.

Python allows us to use regular expressions through the `re` module. Just as Python uses the backslash character as an "escape character" for representing special characters that can't simply be typed into strings, regular expressions use a number of different "special" characters (called **meta-characters**) that are interpreted as ways to signify different types of patterns. For instance, the asterisk character, `*`, stands for "zero or more" of whatever came just before the asterisk. Let's see this in an example, where we use the `re.findall()` function to find any text within a string that matches a given regular expression. The first argument we pass to `re.findall()` is the regular expression that we want to match, and the second argument is the string to test:

```
>>> import re
>>> re.findall("ab*c", "ac")
['ac']
>>> re.findall("ab*c", "abcd")
['abc']
>>> re.findall("ab*c", "acc")
['ac']
>>> re.findall("ab*c", "abcac")
['abc', 'ac']
>>> re.findall("ab*c", "abdc")
>>>
```

Our regular expression, `ab*c`, matches any part of the string that begins with an "a", ends with a "c", and has zero or more of "b" in between the two. This function returns a list of all matches. Note that this is case-sensitive; if we wanted to match this pattern regardless of upper-case or lower-case differences, we could pass a third argument with the value `re.IGNORECASE`, which is a specific variable stored in the `re` module:

```
>>> re.findall("ab*c", "ABC")
>>> re.findall("ab*c", "ABC", re.IGNORECASE)
['ABC']
>>>
```

We can use a period to stand for any single character in a regular expression. For instance, we could find all the strings that contains the letters "a" and "c" separated by a single character as follows:

```
>>> re.findall("a.c", "abc")
['abc']
>>> re.findall("a.c", "abbc")
>>> re.findall("a.c", "ac")
>>> re.findall("a.c", "acc")
['acc']
>>>
```

Therefore, putting the term `.*` inside of a regular expression stands for any character being repeated any number of times. For instance, if we wanted to find every string inside of a particular string that starts with the letter "a" and ends with the letter "c", regardless of what occurs in between these two letters, we could say:

```
>>> re.findall("a.*c", "abc")
['abc']
>>> re.findall("a.*c", "abbc")
['abbc']
>>> re.findall("a.*c", "ac")
['ac']
>>> re.findall("a.*c", "acc")
['acc']
>>>
```

Usually we will want to use the `re.search()` function to search for a particular pattern inside a string. This function is somewhat more complicated because it returns an object called a MatchObject that stores different "groups" of data; this is because there might be matches inside of other matches, and `re.search()` wants to return every possible result. The details of MatchObject are irrelevant here, but for our purposes, calling the `group()` method on a `MatchObject` will return the first and most inclusive result, which most instances is all we care to find. For instance:

```
>>> match_results = re.search("ab*c", "ABC", re.IGNORECASE)
>>> print(match_results.group())
ABC
>>>
```

There is one more `re` function that will come in handy when parsing out text. The `sub()` function, which is short for "substitute," allows us to replace text in a string that matches a regular expression with new text (much like the string `replace()` method). The arguments passed to `re.sub()` are the regular expression, followed by the replacement text, followed by the string. For instance:

```
>>> my_string = "Everything is <replaced> if it's in <tags>."
>>> my_string = re.sub("<.*>", "ELEPHANTS", my_string)
>>> print(my_string)
Everything is ELEPHANTS.
>>>
```

Perhaps that wasn't quite what we expected to happen... We found and replaced everything in between the first `<` and last `>`, which ended up being most of the string. This is because Python's regular expressions are **greedy**, meaning that they try to find the longest possible match when characters like `*` are used. Instead, we should have used the non-greedy matching pattern `*?`, which works the same way as `*` except that it tries to match the shortest possible string of text:

```
>>> my_string = "Everything is <replaced> if it's in <tags>."
>>> my_string = re.sub("<.*?>", "ELEPHANTS", my_string)
>>> print(my_string)
Everything is ELEPHANTS if it's in ELEPHANTS.
>>>
```

Armed with all this knowledge, let's now try to parse out the title from dionysus.html, which includes this rather carelessly written line of HTML:

```
<TITLE >Profile: Dionysus</title  / >
```

Our `find()` method would have a difficult time dealing with the inconsistencies here, but with the clever use of regular expressions, we will be able to handle this code easily:

```
import re
from urllib.request import urlopen

my_address = "https://realpython.com/practice/dionysus.html"

html_page = urlopen(my_address)
html_text = html_page.read()
# Python 3: html_text = html_page.read().decode('utf-8')

match_results = re.search("<title .*?>.*</title .*?>", html_text, re.IGNORECASE)

title = match_results.group()
title = re.sub("<.*?>", "", title) # remove HTML tags

print(title)
```

Let's take the first regular expression we used and break it down into three parts:

1. `<title .*?>` - First we check for the opening tag, where there must be a space after the word "title" and the tag must be closed, but any characters can appear in the rest of the tag; we use the non-greedy `.*?` because we want the first closing ">" to match the tag's end
2. `.*` - Any characters can appear in between the `<title>` tags
3. `</title .*?>` - This expression is the same as the first part, except that we also require the forward slash before "title" because this is a closing HTML tag

Likewise, we then use the non-greedy `.*?` placed inside of an HTML tag to match any HTML tags and remove them from the parsed-out title.

Regular expressions are an incredibly powerful tool when used correctly. We've only scratched the surface of their potential here, so do take some time to study them very thorough Python Regular Expression HOWTO document.

Be sure to also check out the Appendix section on regular expressions for even more practice.

> **NOTE**: Web scraping in practice can be very tedious work. Beyond the fact that no two websites are organized the same way, usually webpages are messy and inconsistent in their formatting. This leads to a lot of time spent handling unexpected exceptions to every rule, which is less than ideal when you want to automate a task. What's more, even after you spend the time to build your scraper, you will probably have to continue to periodically update it as the HTML in a given site changes.

Review exercises:

1. Write a script that grabs the full HTML from the page dionysus.html
2. Use the string `find()` method to display the text following "Name:" and "Favorite Color:" (not including any leading spaces or trailing HTML tags that might appear on the same line)
3. Repeat the previous exercise using regular expressions; the end of each pattern should be a "<" (i.e., the start of an HTML tag) or a newline character, and you should remove any extra spaces or newline characters from the resulting text using the string strip() method

Use an HTML Parser to Scrape Websites

Although regular expressions are great for pattern matching in general, sometimes it's easier to use an HTML parser that is designed specifically for piecing apart HTML pages. There are a number of Python tools written for this purpose, but the most popular (and easiest to learn) is named Beautiful Soup.

To set up Beautiful Soup run `pip3 install beautifulsoup4` .

Otherwise, download the compressed .tar.gz file, unzip it, then install Beautiful Soup using the *setup.py* script from the command line or Terminal as described in the chapter on installing packages.

For Debian/Linux, just type: `sudo apt-get install python-bs4`

Once you have Beautiful Soup installed, you can now import the `bs4` module and pass a string of HTML to `BeautifulSoup` to begin parsing:

```
from bs4 import BeautifulSoup
from urllib.request import urlopen

my_address = "https://realpython.com/practice/dionysus.html"

html_page = urlopen(my_address)
html_text = html_page.read() # Py 3: decode

my_soup = BeautifulSoup(html_text)
```

From here, we can parse data out of `my_soup` in various useful ways depending on what information we want. For instance, BeautifulSoup includes a `get_text()` method for extracting just the text from a document, removing any HTML tags automatically:

```
>>> print(my_soup.get_text())
Profile: Dionysus
Name: Dionysus
Hometown: Mount Olympus
Favorite animal: Leopard
Favorite Color: Wine
>>>
```

There are a lot of extra blank lines left, but these can always be taken out using the string `replace()` method. If we only want to get specific text from an HTML document, using Beautiful Soup to extract the text first and then using `find()` is *sometimes* easier than working with regular expressions.

However, sometimes the HTML tags are actually the elements that point out the data we want to retrieve. For instance, perhaps we want to retrieve links for all the images on the page, which will appear in `<img>` HTML tags. In this case, we can use the `find_all()` method to return a list of all instances of that particular tag:

```
>>> print(my_soup.find_all("img"))
[<img src="dionysus.jpg"/>, <img src="grapes.png"><br><br>
Hometown: Mount Olympus
<br><br>
Favorite animal: Leopard <br>
<br>
Favorite Color: Wine
</br></br></br></br></br></br></img>]
>>>
```

This wasn't exactly what we expected to see, but it happens quite often in the real world; the first element of the list, `<img src="dionysus.jpg"/>`, is a "self-closing" HTML image tag that doesn't require a closing `</img>` tag. Unfortunately, whoever wrote the sloppy HTML for this page never added a closing forward slash to the second HTML image tag, `<img src="grapes.png">`, and didn't include a `</img>` tag either. So Beautiful Soup ended up grabbing a fair amount of HTML after the image tag as well before inserting a `</img>` on its own to correct the HTML.

Fortunately, this still doesn't have much bearing on how we can parse information out of the image tags with Beautiful Soup. This is because these HTML tags are stored as `Tag` objects, and we can easily extract certain information out of each Tag. In our example, assume for simplicity that we know to expect two images in our list so that we can pull two `Tag` objects out of the list:

```
>>> image1, image2 = my_soup.find_all("img")
>>>
```

We now have two `Tag` objects, `image1` and `image2`. These `Tag` objects each have a name, which is just the type of HTML tag that to which they correspond:

```
>>> print(image1.name)
img
>>>
```

These `Tag` objects also have various attributes, which can be accessed in the same way as a dictionary. The HTML tag `<img src="dionysus.jpg"/>` has a single attribute `src` that takes on the value `dionysus.jpg` (much like a key/value pair in a dictionary). Likewise, an HTML tag such as `<a href="https://realpython.com" target="_blank">` would have two attributes, a `href` attribute that is assigned the value `https://realpython.com` and a `target` attribute that has the value `_blank`.

We can therefore pull the image source (the link that we wanted to parse) out of each image tag using standard dictionary notation to get the value that the `src` attribute of the image has been assigned:

```
>>> print(image1["src"])
dionysus.jpg
>>> print(image2["src"])
grapes.png
>>>
```

Even though the second image tag had a lot of extra HTML code associated with it, we could still pull out the value of the image `src` without any trouble because of the way Beautiful Soup organizes HTML tags into `Tag` objects.

In fact, if we only want to grab a particular tag, we can identify it by the corresponding name of the `Tag` object in our soup:

```
>>> print(my_soup.title)
<title>Profile: Dionysus</title>
>>>
```

Notice how the HTML `<title>` tags have automatically been cleaned up by Beautiful Soup. Furthermore, if we want to extract only the string of text out of the `<title>` tags (without including the tags themselves), we can use the string attribute stored by the title:

```
>>> print(my_soup.title.string)
Profile: Dionysus
>>>
```

We can even search for specific kinds of tags whose attributes match certain values. For instance, if we wanted to find all of the `<img>` tags that had a `src` attribute equal to the value `dionysus.jpg`, we could provide the following additional argument to the find_all() method:

```
>>> my_soup.find_all("img", src="dionysus.jpg")
[<img src="dionysus.jpg"/>]
>>>
```

In this case, the example is somewhat arbitrary since we only returned a list that contained a single image tag, but we will use this technique in a later section in order to help us find a specific HTML tag buried in a vast sea of other HTML tags.

Although Beautiful Soup is an excellent tool, it has it's limitations. lxml is somewhat trickier to get started using, but offers all of the same functionality as Beautiful Soup and more. Once you are comfortable with the basics of Beautiful Soup, you should move on to learning how to use lxml for more complicated HTML parsing tasks.

> **NOTE**: HTML parsers like Beautiful Soup can (and often do) save a lot of time and effort when it comes to locating specific data in webpages. However, sometimes HTML is so poorly written and disorganized that even a sophisticated parser like Beautiful Soup doesn't really know how to interpret the HTML tags properly. In this case, you're often left to your own devices (namely, `find()` and regex) to try to piece out the information you need.

Review exercises:

1. Write a script that grabs the full HTML from the page profiles.html
2. Parse out a list of all the links on the page using Beautiful Soup by looking for HTML tags with the name `a` and retrieving the value taken on by the `href` attribute of each tag
3. Get the HTML from each of the pages in the list by adding the full path to the file name, and display the text (without HTML tags) on each page using Beautiful Soup's `get_text()` method

Interact with HTML Forms

We usually retrieve information from the Internet by sending requests for webpages. A module like `urllib` serves us well for this purpose, since it offers a very simple way of returning the HTML from individual webpages. Sometimes, however, we need to send information back to a page - for instance, submitting our information on a login form. For this, we need an actual browser. There are a number of web browsers built for Python, and one of the most popular and easiest to use is in a module called MechanicalSoup.

Essentially, MechanicalSoup is an alternative to `urllib` that can do all of the same things but has more added functionality that will allow us to talk back to webpages without using a standalone browser, perfect for fetching web pages, clicking on buttons and links, and filling out and submitting forms.

To install run this command:

```
$ pip3 install MechanicalSoup
```

Otherwise, download the .tar.gz file, decompress it, and install the package by running the setup.py script from your Terminal as described in the section on installing packages.

You may need to close and restart your IDLE session for MechanicalSoup to load and be recognized after it's been installed.

Getting MechanicalSoup to create a new Browser object and use it to open a webpage is as easy as saying:

```
import mechanicalsoup

my_browser = mechanicalsoup.Browser()
page = my_browser.get("https://realpython.com/practice/aphrodite.html")
```

We now have various information that the website returned to us stored in our `response` variable. If we access the attribute `soup` , we'll see all of the html from the page:

```
>>> print(page.soup)
<html>
  <head>
  <title>Profile: Aphrodite</title>
  </head>
  <body bgcolor="yellow">
  <center>
    <br><br>
      <img src="aphrodite.gif" />
      <h2>Name: Aphrodite</h2>
      <br><br>
      Favorite animal: Dove
      <br><br>
      Favorite color: Red
      <br><br>
      Hometown: Mount Olympus
    </center>
  </body>
</html>
>>>
```

But what if we have to submit information to the website? For instance, what if the information we want is behind a login page such as https://www.realpython.com/practice/login.php? If we are trying to do things automatically, then we will need a way to automate the login process as well.

First, let's take a look at the HTML page provided by login.php:

```
import mechanicalsoup

my_browser = mechanicalsoup.Browser()
page = my_browser.get("https://realpython.com/practice/login.php")

print(page.soup)
```

This returns the following form (which you should take a look at in a regular browser as well to see how it appears):

```
<html>
  <head>
  <title>Log In</title>
  </head>
  <body bgcolor="yellow">
  <center>
      <br><br>
      <h2>Please log in to access Mount Olympus:</h2>
      <br><br>
      <form name="login" action="login.php" method="post">
        Username: <input type="text" name="user"><br>
        Password: <input type="password" name="pwd"><br><br>
        <input type="submit" value="Submit">
      </form>
    </center>
  </body>
</html>
>>>
```

The code we see is HTML, but the page itself is written in another language called PHP. In this case, the PHP code is creating the HTML that we see based on the information we provide. For instance, try logging into the page with an incorrect username and password, and you will see that the same page now includes a line of text to let you know: "Wrong username or password!" However, if you provide the correct login information (username of "zeus" and password of "ThunderDude"), you will be redirected to the profiles page page.

For our purposes, the important section of HTML code is the login form, i.e., everything inside the `<form>` tags. We can see that there is a submission `<form>` named "login" that includes two `<input>` tags, one named `user` and the other named `pwd`. The third `<input>` is the actual "Submit" button. Now that we know the underlying structure of the form, we can return to mechanize to automate the login process.

```
import mechanicalsoup

my_browser = mechanicalsoup.Browser()
login_page = my_browser.get("https://realpython.com/practice/login.php")
login_html = login_page.soup

# select the form and fill in its fields
form = login_html.select("form")[0]
form.select("input")[0]["value"] = "zeus"
form.select("input")[1]["value"] = "ThunderDude"

profiles_page = my_browser.submit(form, login_page.url) # submit form

print(profiles_page.url) # make sure we were redirected
print(profiles_page.soup) # show html
```

We used the browser's `select()` method to grab the form. Then we passed our login values by accessing the `value` attribute of the html `input`'s. Finally, using the the browser object again, we submitted our form with the `submit()` method, which takes a form element and a url, and returns and object containing information about the page we were redirected to.

We displayed the URL of this response to make sure that our login submission worked; if we had provided an incorrect username or password then we would have been sent back to the login page, but we see that we were successfully redirected to profiles as planned.

> **NOTE**: We are always being encouraged to use long passwords with many different types of characters in them, and now you know the main reason: automated scripts like the one we just designed can be used by hackers to "brute force" logins by rapidly trying to log in with many different usernames and passwords until they find a working combination. Besides this being highly illegal, almost all websites these days (including my practice form) will lock you out and report your IP address if they see you making too many failed requests, so don't try it!

We were able to retrieve the webpage form by name because mechanize includes its own HTML parser. We can use this parser through various browser methods as well to easily obtain other types of HTML elements. The `links()` method will return all the links appearing on the browser's current page as Link objects, which we can then loop over to obtain their addresses. For instance, if our browser is still on the profiles page, we could say:

```
for link in profiles_page.soup.select("a"):
    print("Address:", link["href"])
    print("Text:", link.text)
```

This returns:

```
Address: https://realpython.com/practice/aphrodite.html
Text: Aphrodite
Address: https://realpython.com/practice/poseidon.html
Text: Poseidon
Address: https://realpython.com/practice/dionysus.html
Text: Dionysus
>>>
```

Html elements have a number of attributes. Here as we are accessing an `a` tag we van grab the "href" and we also have access to the inner text. Depending on what element you are dealing with you will access the attributes with either dot notation or bracket notation.

With the simple functionality of MechanicalSoup you can get a lot done. By utilizing `get` , `select` , and `submit` methods we can navigate to pages, access the html and submit custom information into forms. Currently the documentation for MechanicalSoup is non-existent, but there are a few more method available that you can find if you look through the source code.

Review exercises:

1. Use MechanicalSoup to provide the correct username "zeus" and password "ThunderDude" to the login page submission form located at: https://realpython.com/practice/login.php
2. Using Beautiful Soup, display the title of the current page to determine that you have been redirected to profiles.html
3. Use mechanize to return to login page by going "back" to the previous page
4. Provide an incorrect username and password to the login form, then search the HTML of the returned webpage for the text "Wrong username or password!" to determine that the login process failed

Interact with Websites in Real-time

Sometimes we want to be able to fetch real-time data from a website that offers continually updated information. In the dark days before you learned Python programming, you would have been forced to sit in front of a browser, clicking the "Refresh" button to reload the page each time you want to check if updated content is available. Instead, we can easily automate this process using the `get()` method of the MechanicalSoup browser.

As an example, let's create a script that periodically checks Yahoo! Finance for a current stock quote for the "YHOO" symbol. The first step with any web scraping task is to figure out exactly what information we're seeking. In this case, the webpage URL is http://finance.yahoo.com/q?s=yhoo. Currently, the stock price (as I see it) is 40.01, and so we view the page's HTML source code and search for this number. Fortunately, it only occurs once in the code:

```
<span class="time_rtq_ticker"><span id="yfs_l84_yhoo">40.01</span></span>
```

Next, we check that the tag `<span id="yfs_l84_yhoo">` also only occurs once in the webpage, since we will need a way to uniquely identify the location of the current price. If this is the only `<span>` tag with an `id` attribute equal to "yfs_l84_yhoo", then we know we'll be able to find it on the webpage later and extract the information we need from this particular pair of `<span>` tags.

We're in luck (this is the only `<span>` tag with this `id`), so we can use MechanicalSoup to find and display the current price, like so:

```
import mechanicalsoup

my_browser = mechanicalsoup.Browser()
page = my_browser.get("http://finance.yahoo.com/q?s=yhoo")
html_text = page.soup

# return a list of all the tags where the id is 'yfs_184_yhoo'
my_tags = html_text.select("#yfs_l84_yhoo")

# take the BeautifulSoup string out of the first (and only) <span> tag
my_price = my_tags[0].text

print("The current price of YHOO is: {}".format(my_price))
```

Now, in order to repeatedly get the newest stock quote available, we'll need to create a loop that loads the page in the browser each time. But first, we should check the Yahoo! Finance terms of use to make sure that this isn't in violation of their acceptable use policy. The terms state that we should not "use the Yahoo! Finance Modules in a manner that exceeds reasonable request volume [or] constitutes excessive or abusive usage," which seems reasonable enough. Of course, reasonable and excessive are entirely subjective terms, but the general rules of Internet etiquette suggest that you don't ask for more data than you need. Sometimes, the amount of data you "need" for a particular use might still be considered excessive, but following this rule is a good place to start.

In our case, an infinite loop that grabs stock quotes as quickly as possible is definitely more than we need, especially since it appears that Yahoo! only updates its stock quotes once per minute. Since we'll only be using this script to make a few webpage requests as a test, let's wait one minute in between each request. We can pause the functioning of a script by passing a number of seconds to the `sleep()` method of Python's `time` module, like so:

```python
from time import sleep

print "I'm about to wait for five seconds..."
sleep(5)
print "Done waiting!"
```

Although we won't explore them here, Python's time module also includes various ways to get the current time in case we wished to add a "time stamp" to each price.

Using the `sleep()` method, we can now repeatedly obtain real-time stock quotes:

```
from time import sleep
import mechanicalsoup

my_browser = mechanicalsoup.Browser()


# obtain 1 stock quote per minute for the next 3 minutes
for i in range(0, 3):
    page = my_browser.get("http://finance.yahoo.com/q?s=yhoo")
    html_text = page.soup
    # return a list of all the tags where the id is 'yfs_184_yhoo'
    my_tags = html_text.select("#yfs_l84_yhoo")
    # take the BeautifulSoup string out of the first tag
    my_price = my_tags[0].text
    print("The current price of YHOO is: {}".format(my_price))
    if i<2: # wait a minute if this isn't the last request
        sleep(60)
```

Review exercises:

1. Repeat the example in this section to scrape YHOO stock quotes, but additionally include the current time of the quote as obtained from the Yahoo! Finance webpage; this time can be taken from part of a string inside another span tag that appears shortly after the actual stock price in the webpage's HTML

Again, check out the second Real Python course, *Web Development with Python*, which provides much more in the way of web scraping, utilizing a more powerful scraping library, Scrapy.

Scientific Computing and Graphing

If you are a scientist, an engineer, or the sort of person who couldn't survive a week without using MATLAB, chances are high that you will want to make use of the NumPy and SciPy packages to increase your Python coding abilities. Even if you don't fall into one of those categories, these tools can still be quite useful. This section will introduce you to a Python package that lets you store and manipulate matrices of data, and the next section will introduce an additional package that makes it possible to visualize data through endless varieties of graphs and charts.

Use NumPy for Matrix Manipulation

The main package for scientific computing in Python is NumPy; there are a number of additional specialized packages, but most of these are based on the functionality offered by NumPy. To install NumPy and SciPy use pip3:

```
pip3 install numpy
pip3 install scipy
```

Otherwise, download the latest version here for 32-bit Windows or here for 32-bit Mac, then run the automated installer. If you have a 64-bit system (including OS X 10.6 or later), you'll need to install a 64-bit release of SciPy (which includes Numpy), available here or here. Debian/Ubuntu users can get NumPy by typing: `sudo apt-get install python-numpy`

Among many other possibilities, NumPy primarily offers an easy way to manipulate data stored in many dimensions. For instance, we usually think of a two-dimensional list as a "matrix" or a "table" that could be created by forming a "list of lists" in Python:

```
>>> matrix = [[1,2,3], [4,5,6], [7,8,9]]
>>> print(matrix)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
```

We could then refer to the numbers at various row/column locations using index numbers. For instance:

```
>>> matrix[0][1]
2
>>>
```

Things get much more complicated, however, if we want to do anything more complicated with this two-dimensional list. For instance, what if we wanted to multiply every entry in our matrix by 2? That would require looping over every entry in every list inside the main list.

Let's create this same list using NumPy:

```
>>> from numpy import array
>>> matrix = array([[1,2,3], [4,5,6], [7,8,9]])
>>> print(matrix)
[[1 2 3]
 [456]
 [789]]
>>> matrix[0][1]
2
>>> matrix[0,1]
2
>>>
```

In this case, our matrix is referred to as a two-dimensional array. An **array** is different from a list because it can only hold similar entries (for instance, all numbers) whereas we could throw any sort of objects together into a list. However, there are many more ways we can create and control NumPy arrays. **See more here on Numpy arrays vs Python lists.**

> **NOTE**: In NumPy, each dimension in an array is called an axis. Our example array has two axes. The total number of dimensions or axes is called the "rank" of a matrix, but these are really all terms for describing the same thing; just keep in mind that NumPy often refers to the term "axis" to mean a dimension of an array. Above, we saw a two-dimensional array. This is an example of a three-dimensional ("three axis") array:
>
> ```
> >>> matrix =  [
>                 [ [1,2,3], [4,5,6] ],
>                 [ [7,8,9], [10,11,12] ],
>                 [ [13,14,15], [16,17,18] ]
>               ]
> >>> matrix[0][1][2]
> 6
> >>>
> ```

Another benefit, as we can already see, is that NumPy automatically knows to display our two-dimensional ("two axis") array in two dimensions so that we can easily read its contents. We also have two options for accessing an entry, either by the usual indexing or by specifying the index number for each axis, separated by commas.

Remember to include the main set of square brackets when creating a NumPy array; even though the `array()` has parentheses, including square brackets is necessary when you want to type out the array entries directly. For instance, this is correct: `matrix = array([[1,2],[3,4]])` Meanwhile, this would be INCORRECT because it is missing outer brackets: `matrix = array([1,2],[3,4])`

We have to type out the array this way because we could also have given a different input to create the `array()` - for instance, we could have supplied a list that is already enclosed in its own set of square brackets:

```
list = [[1,2], [3,4]]
matrix = array(list)
```

In this case, including the square brackets around list would be nonsensical.

Now, multiplying every entry in our matrix is as simple as working with an actual matrix:

```
>>> from numpy import array
>>> matrix = array([[1,2,3], [4,5,6], [7,8,9]])
>>> print(matrix*2)
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
>>>
```

> **NOTE**: Even if you have no interest in using matrices for scientific computing, you still might find it helpful at some point to store information in a NumPy array because of its many helpful properties. For instance, perhaps you are designing a game and need an easy way to store, view and manipulate a grid of values with rows and columns. Rather than creating a list of lists or some other complicated structure, using a NumPy array is a simple way to store your two-dimensional data.

We can just as easily perform arithmetic using multi-dimensional arrays as well:

```
>>> second_matrix = array([[5,4,3], [7,6,5], [9,8,7]])
>>> print(second_matrix - matrix)
[[ 4 2  0]
 [ 3 1 -1]
 [ 2 0 -2]]
>>>
```

If you want to perform matrix multiplication, using the standard * symbol will perform basic multiplication by matching corresponding entries:

```
>>> print(second_matrix * matrix)
[[ 5  8  9]
 [28 30 30]
 [63 64 63]]
>>>
```

To calculate an actual matrix dot product, we need to import and use the `dot()` function:

```
>>> from numpy import dot
>>> print(dot(second_matrix, matrix))
[[ 42  54  66]
 [ 66  84 102]
 [ 90 114 138]]
>>>
```

Two matrices can also be stacked vertically using `vstack()` or horizontally using `hstack()` if their axis sizes match:

```
>>> from numpy import vstack, hstack
>>> print(vstack([matrix, second_matrix]))  # add second_matrix below matrix
[[1 2 3]
 [4 5 6]
 [7 8 9]
 [5 4 3]
 [7 6 5]
 [9 8 7]]
>>> print(hstack([matrix, second_matrix])) # add second_matrix next to matrix
[[1 2 3 5 4 3]
 [4 5 6 7 6 5]
 [7 8 9 9 8 7]]
>>>
```

For basic linear algebra purposes, a few of the most commonly used NumPy array properties are also shown here briefly, as they are all fairly self-explanatory:

```
>>> print(matrix.shape) # a tuple of the axis lengths (3 x 3)
(3, 3)
>>> print(matrix.diagonal()) # array of the main diagonal entries
[1 5 9]
>>> print(matrix.flatten()) # a flat array of all entries
[1 2 3 4 5 6 7 8 9]
>>> print(matrix.transpose()) # mirror-image along the diagonal
[[1 4 7]
 [2 5 8]
 [3 6 9]]
>>> print(matrix.min()) # the minimum entry
1
>>> print(matrix.max()) # the maximum entry
9
>>> print(matrix.mean()) # the average value of all entries
5.0
>>> print(matrix.sum()) # the total of all entries
45
>>>
```

We can also reshape arrays with the `reshape()` function to shift entries around:

```
>>> print matrix.reshape(9,1)
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]
 [9]]
>>>
```

Of course, the total size of the reshaped array must match the original array's size. For instance, we couldn't have said `matrix.reshape(2,5)` because there would have been one extra entry created with nothing left to fill it.

The `reshape()` function can be particularly helpful in combination with `arange()`, which is NumPy's equivalent to `range()` except that a NumPy array is returned. For instance, instead of typing out our sequential list of numbers into a matrix, we could have imported `arange()` and then reshaped the sequence into a two-dimensional matrix:

```
>>> from numpy import arange
>>> matrix = arange(1,10) # an array of numbers 1 through 9
>>> print(matrix)
[1 2 3 4 5 6 7 8 9]
>>> matrix = matrix.reshape(3,3)
>>> print(matrix)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>>
```

Again, just like with `range()`, using `arange(1,10)` will return the numbers 1 through 9 because we stop just before getting to the last number in the range.

We have been working entirely with two-dimensional arrays since they are both the most commonly used and the easiest to grasp. However, just as we can create a "list of lists of lists", NumPy allows us to create arrays of higher dimensions as well. For instance, we could create a simple three dimensional array with the following:

```
>>> array_3d = array([[[1,2],[3,4]], [[5,6],[7,8]], [[9,10],[11,12]]])
>>> print(array_3d)
[[[ 1  2]
  [ 3  4]]

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]]
>>>
```

An easier and safer way to create this particular array would be to `reshape()` and `arange()`:

```
>>> array_3d = arange(1,13)
>>> array_3d = array_3d.reshape(3,2,2)
>>>
```

Even if a multi-dimensional array doesn't use a sequential list of numbers, sometimes it's easier to create the flat, one-dimensional list of entries and then `reshape()` the array into the desired shape instead of struggling with many nested sets of square brackets.

NumPy also has its own set of random functionality, which is particularly useful for creating multi-dimensional arrays of random numbers. For instance, we can easily create a 3x3 matrix of random numbers like so:

```
>>> from numpy import random
>>> print(random.random([3,3]))
[[ 0.738695   0.52153367 0.58619601]
 [ 0.38232677 0.15941573 0.66080916]
 [ 0.61752779 0.60236187 0.5914662 ]]
>>>
```

NumPy offers much more functionality than the basics shown here, although everything covered should be sufficient for most tasks involving basic array storage and manipulation. For an incredibly thorough and mathematical introduction, see the Guide to NumPy. There is also a good Quick Start Tutorial available and a NumPy "work in progress" User Guide.

For scientists and engineers, another indispensable tool is SciPy, which works on top of NumPy to offer a mind-numbingly vast set of tools for data manipulation and visualization. SciPy is a very powerful and very extensive collection of functions and algorithms that are too advanced to cover in an introductory course, but it's worth researching if you have a particular advanced topic already in mind. Some of the tools available include functionality for tasks involving optimization, integration, statistical tests, signal processing, Fourier transforms, image processing and more. For a thorough introduction to additional functionality available in SciPy, there is a reference guide that covers most major topics. SciPy is availbale for download via `pip3`. Or download here for 32-bit and here for 64-bit machines.

Review exercises:

1. Create a 3 x 3 NumPy array named `first_matrix` that includes the numbers 3 through 11 by using `arange()` and `reshape()`
2. Display the minimum, maximum and mean of all entries in `first_matrix`
3. Square every entry in `first_matrix` using the `**` operator, and save the results in an array named `second_matrix`
4. Use `vstack()` to stack `first_matrix` on top of `second_matrix` and save the results in an array named `third_matrix`
5. Use `dot()` to calculate the dot product of `third_matrix` by `first_matrix`
6. Reshape `third_matrix` into an array of dimensions 3 x 3 x 2

Use matplotlib for Plotting Graphs

The matplotlib library works with NumPy to provide tools for creating a wide variety of two-dimensional figures for visualizing data. If you have ever created graphs in MATLAB, matplotlib in many ways directly recreates this experience within Python. Figures can then be exported to various formats in order to save pictures and/or documents.

You will first need to download and install both NumPy (see the previous section) and matplotlib:

```
pip3 install numpy
pip3 install matplotlib
```

Or, Windows and OS X users can download an automated matplotlib installer here.

Debian/Linux users can get matplotlib by typing the command: `sudo apt-get install python-matplotlib`

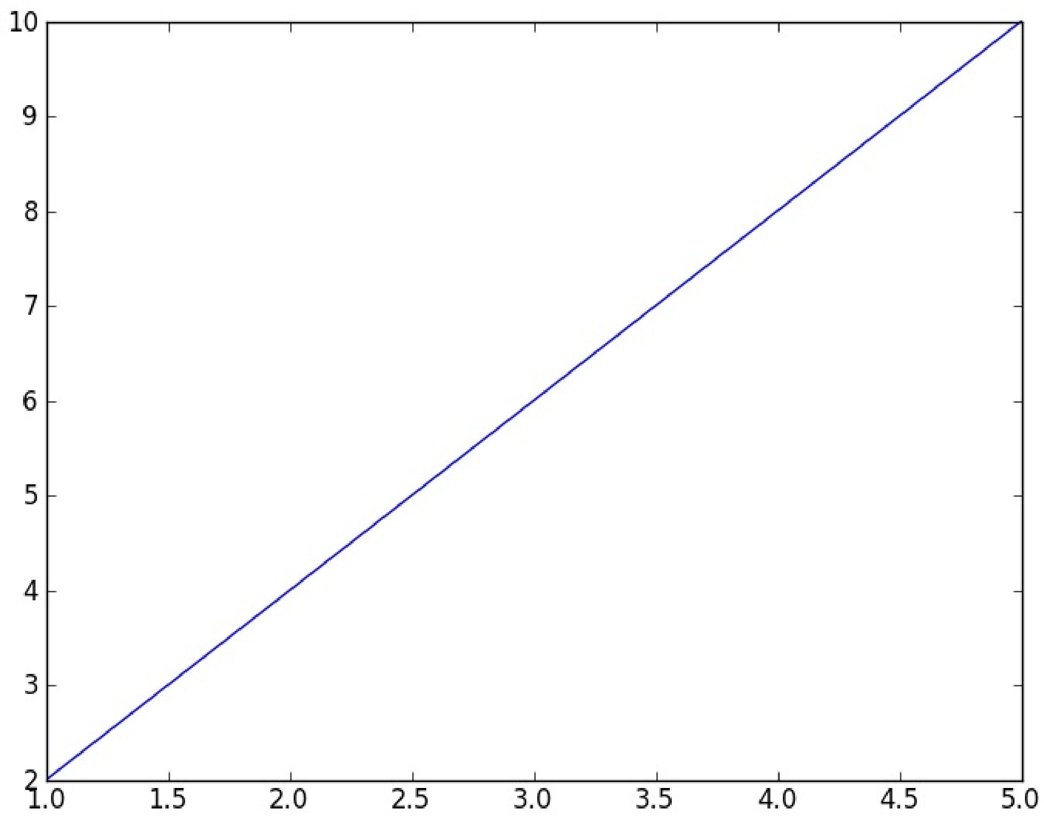
There are a few modules included in the matplotlib package, but we will only work with the basic plotting module, pyplot. We can import this functionality as follows:

```
from matplotlib import pyplot as plt
```

This part of the code might take a little time to run - there is a lot of code being imported with this line! We decided to rename pyplot to the name "plt" to save a little space and effort since we'll be typing it a lot. Plotting a simple graph is in fact quite simple to do.

Try out this short script:

```
from matplotlib import pyplot as plt
plt.plot([1,2,3,4,5], [2,4,6,8,10])
plt.show()
```

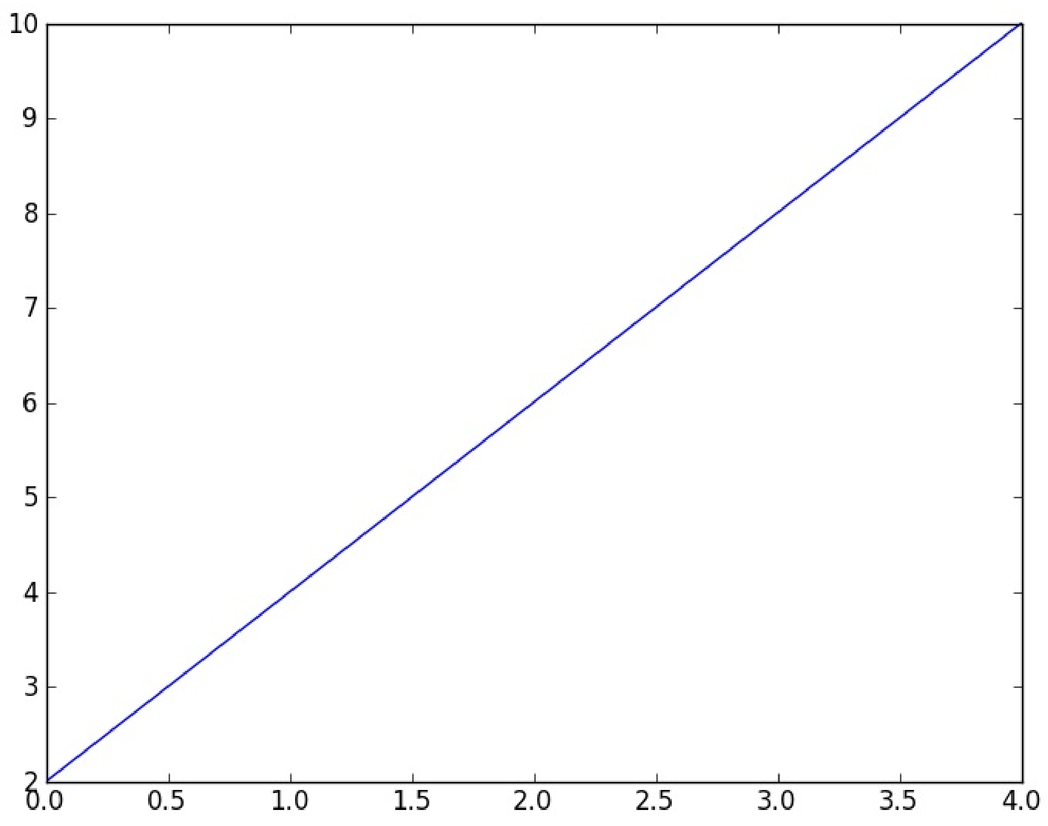


We created a plot, supplying a list of x-coordinate points and a matching list of y-coordinate points. When we call `plt.show()`, a new window appears, displaying a graph of our five (x,y) points. Our interactive window is essentially locked while this plot window is open, and we can only end the script and return to a new prompt in the interactive window once we've closed the graph.

> **NOTE**: We'll stick to using scripts to plot rather than typing commands into the interactive window. Using Windows, you should also have no problems copying this code line-by-line into the interactive window. However, newer versions of OS X have difficulty responding to `show()`. Meanwhile, IDLE in Linux can correctly `show()` a plot from an interactive window, but then IDLE will need to be restarted. We'll discuss other options and alternatives for displaying and interacting with plots at the end of this section.

When plotting, we don't even have to specify the horizontal axis points; if we don't include any, matplotlib will assume that we want to graph our y values against a sequential x axis increasing by one:

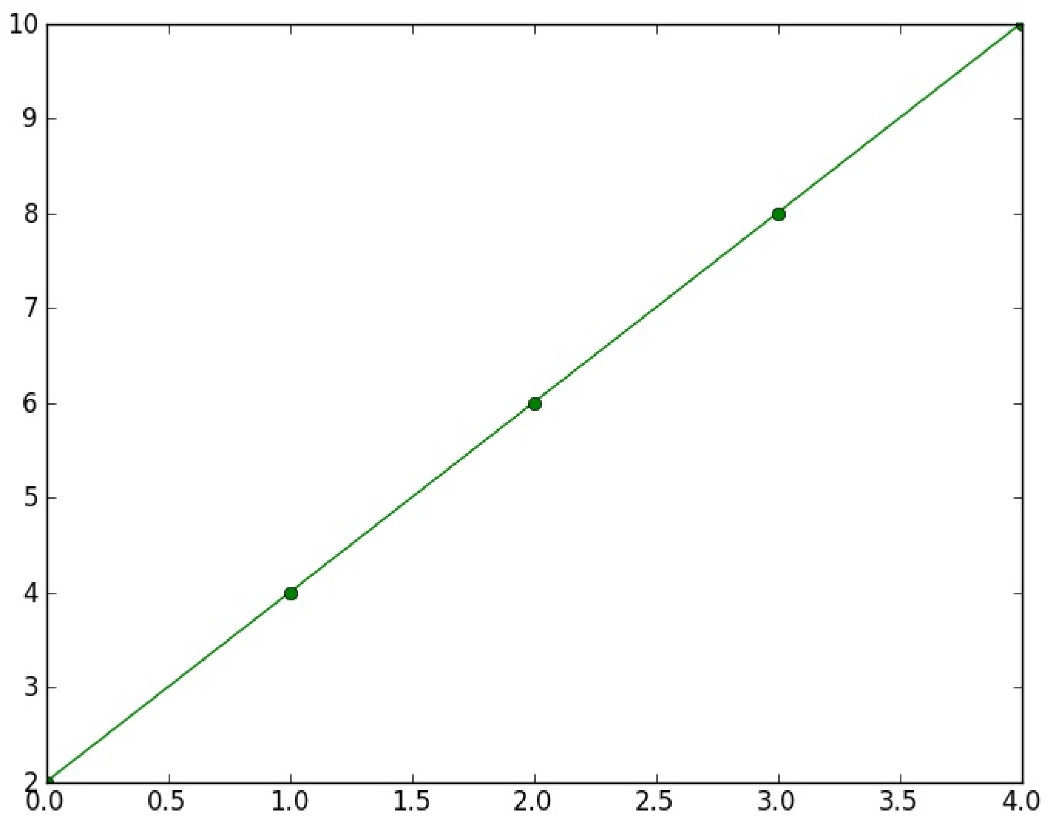
```
from matplotlib import pyplot as plt
plt.plot([2,4,6,8,10])
plt.show()
```



However, this isn't exactly the same graph because, as always, Python begins counting at 0 instead of 1, which is what we now see for the horizontal axis values as well.

There is a optional "formatting" argument that can be inserted into `plot()` after specifying the points to be plotted. This argument specifies the color and style of lines or points to draw. Unfortunately, the standard is borrowed from MATLAB and (compared to most Python) the formatting is not very intuitive to read or remember. The default value is "solid blue line", which would be represented by the format string `b-`. If we wanted to plot green circular dots connected by solid lines instead, we would use the format string `g-o` like so:

```
from matplotlib import pyplot as plt
plt.plot([2,4,6,8,10], "g-o")
plt.show()
```



> **SEE ALSO**: For reference, the full list of possible formatting combinations can be found here.

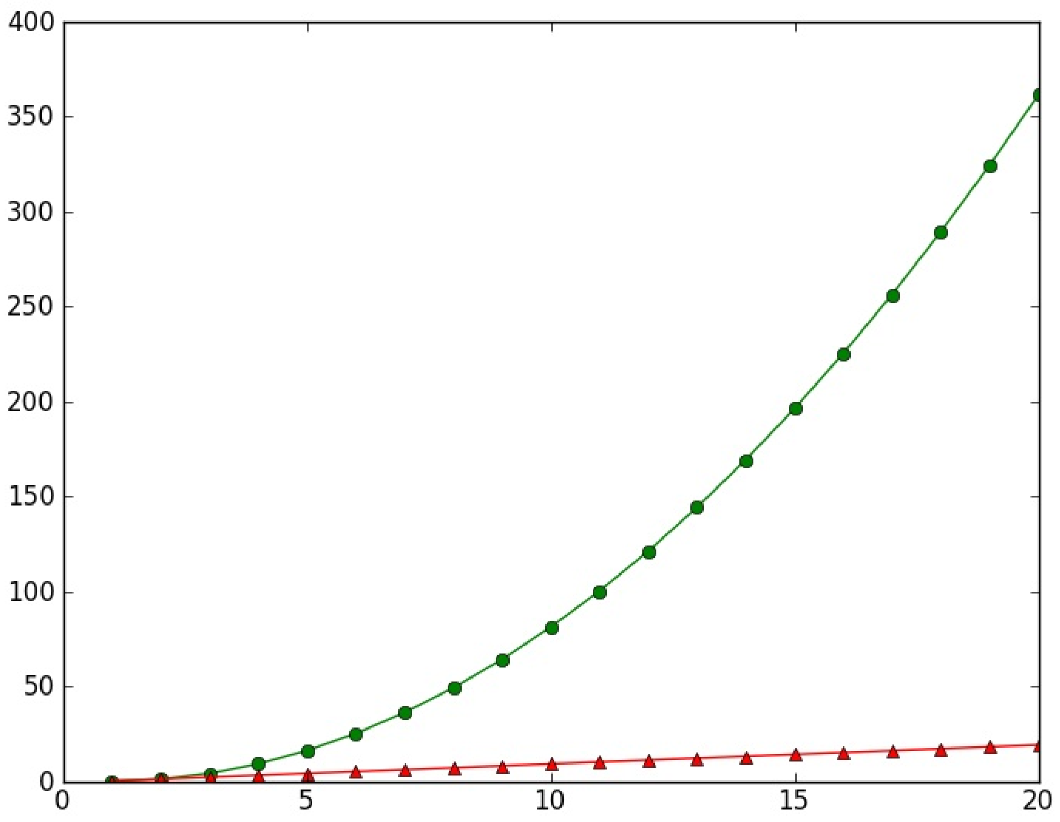
Plotting wouldn't be very convenient if we had to organize everything into lists ourselves first; matplotlib works with NumPy to accept arrays as well. (Even if you can get away with using a basic list, it's a good idea to stick with arrays in case you later discover that you do need to modify the numbers in some way.) For instance, to create our previous graph, we could use `arange()` to return an array of the same numbers:

```
from matplotlib import pyplot as plt
from numpy import arange

plt.plot(arange(2,12,2), "g-o")
plt.show() # displays the same graph as the previous example
```

Here, we used the optional third argument of `arange()`, which specifies the step. The step is the amount by which to increase each subsequent number, so saying `arange(2,12,2)` gives us an array of numbers beginning at 2, increasing by 2 every step, and ending before 12. (In fact, this third step argument works exactly the same way for the built-in `range()` function as well.)

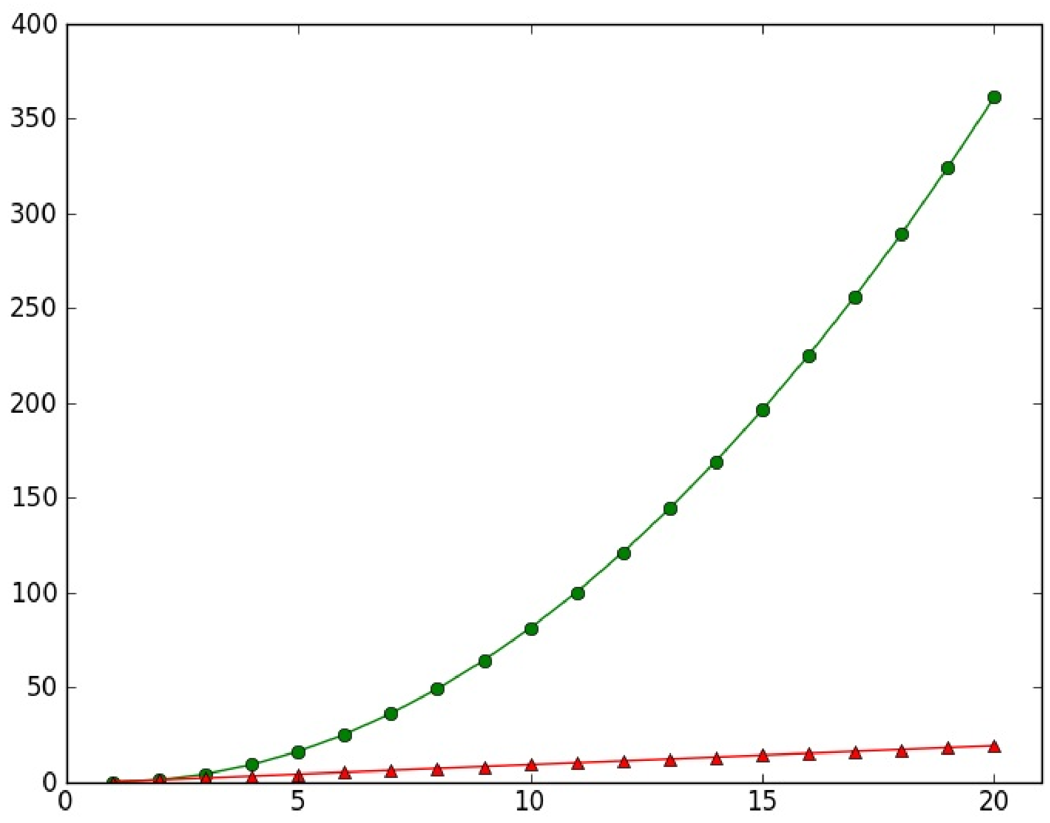
To plot multiple sets of points, we add them to `plot()` as additional arguments:

```
from matplotlib import pyplot as plt
from numpy import arange

x_points = arange(1,21)
baseline = arange(0,20)
plt.plot(x_points, baseline**2, "g-o", x_points, baseline, "r-^")
plt.show()
```



This isn't a very pretty graph, though. Fortunately, there are plenty of things we can do to improve the layout and formatting. First of all, let's change the axes so that our points don't go off the corners of the graph:

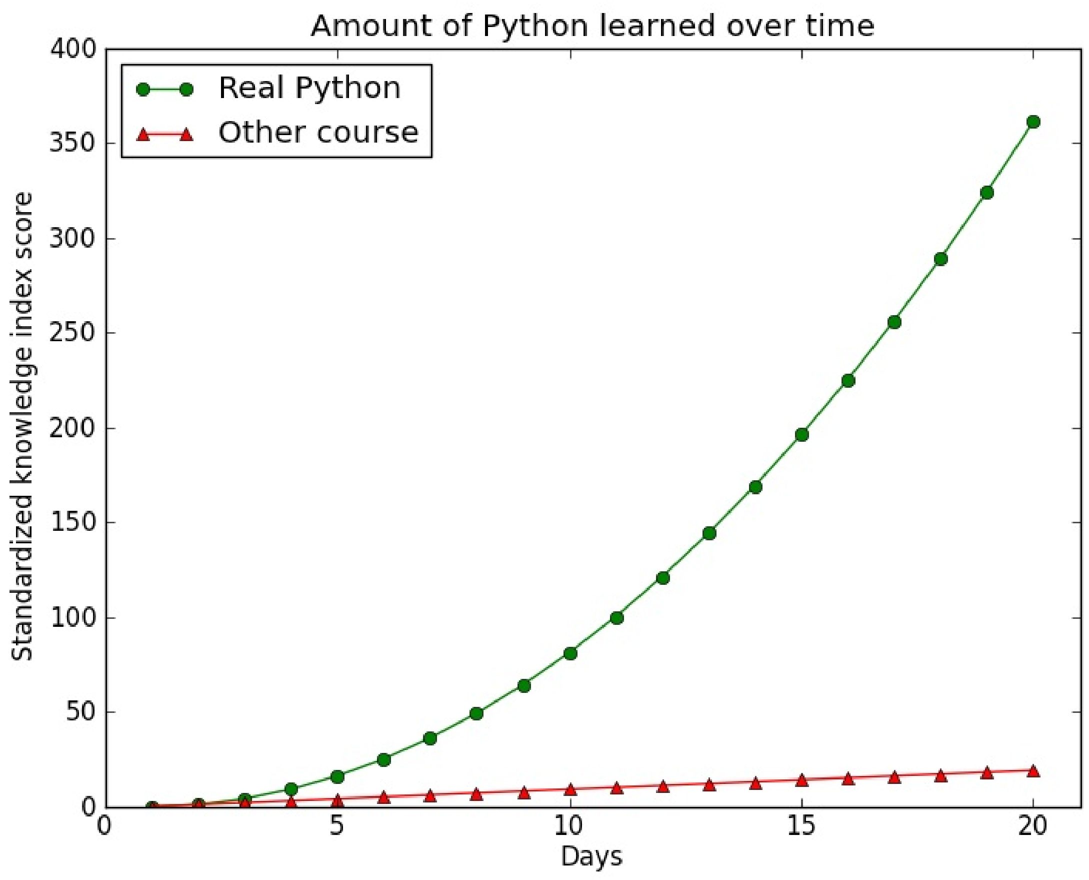
```
from matplotlib import pyplot as plt
from numpy import arange

x_points = arange(1,21)
baseline = arange(0,20)
plt.plot(x_points, baseline**2, "g-o", x_points, baseline, "r-^")
plt.axis([0, 21, 0, 400])
plt.show()
```



We define the boundaries of the displayed axes with a list of the four points in the order [min X, max X, min Y, max Y] - in this case, we increased the maximum value that the x-axis extended from 20 to 21 so that the last two points don't appear halfway off the graph.

Now we're starting to get somewhere useful, but nothing is labeled yet. Let's add a main title, a legend and some labels for the axes:

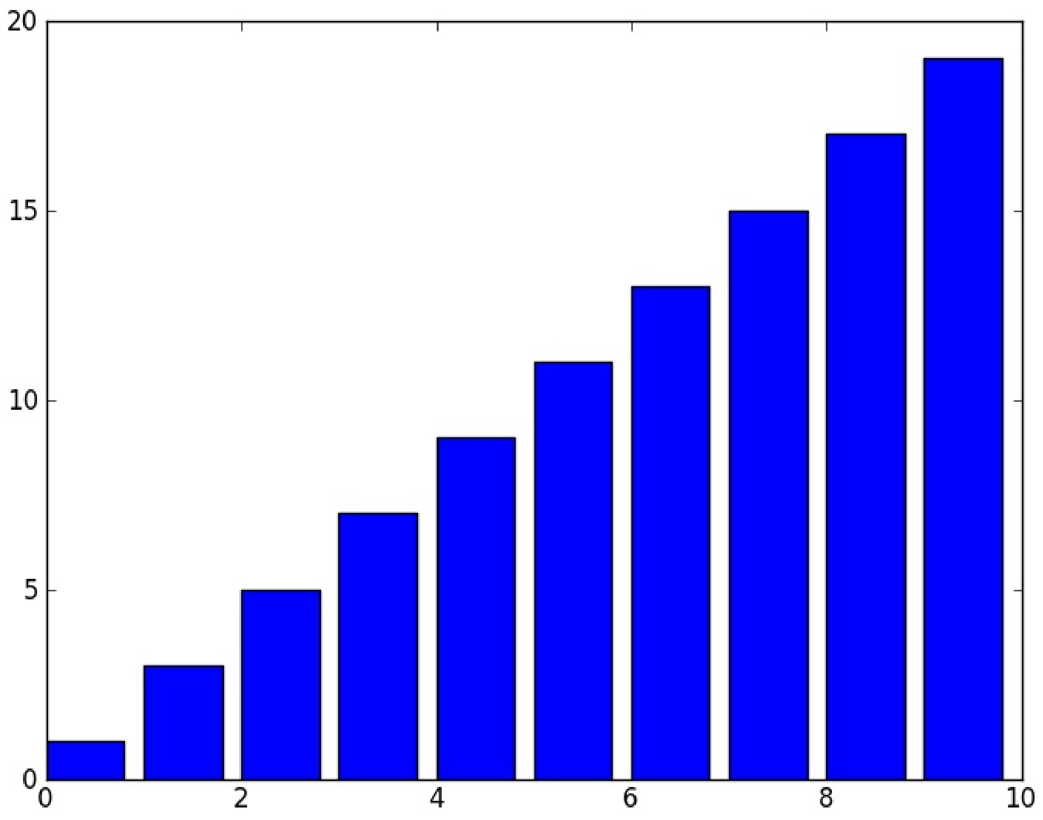
```
from matplotlib import pyplot as plt
from numpy import arange

x_points = arange(1,21)
baseline = arange(0,20)
plt.plot(x_points, baseline**2, "g-o", x_points, baseline, "r-^")
plt.axis([0, 21, 0, 400])
plt.title("Amount of Python learned over time")
plt.xlabel("Days")
plt.ylabel("Standardized knowledge index score")
plt.legend(("Real Python", "Other course"), loc=2)
plt.show()
```



There are many more complicated ways to create a legend, but the simplest is to supply a tuple that lists the labels to be applied to all the plotted points in order. We specified `loc=2` in the legend because we want the legend to appear in the top left corner, while the default is for the legend to appear in the top right corner. Unless you're making graphs very frequently, it's near impossible to remember the particular detail that `loc=2` corresponds to the top left corner along with so many other cryptic formatting details; the best thing to do in this situation is to search the web for a relevant term like "matplotlib legend". In this case, you'll quickly be directed to the matplotlib legend guide that offers, among many other details, a table providing legend location codes.

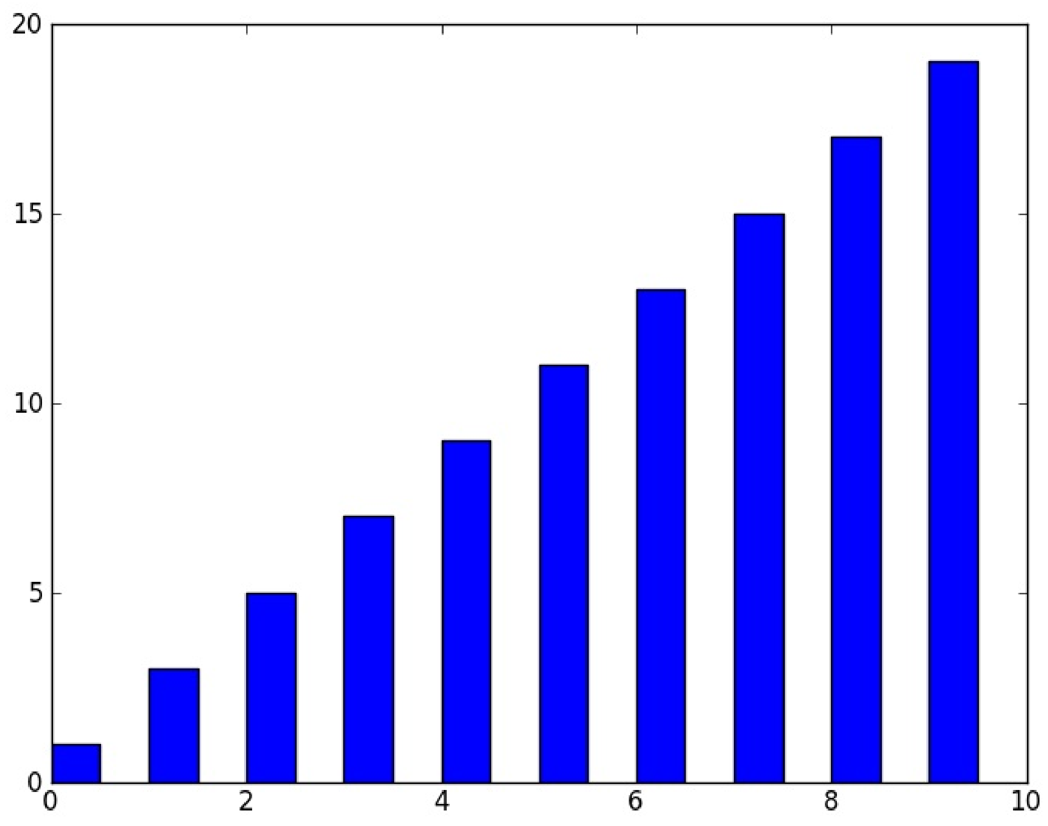
Another frequently used type of plot in basic data visualization is a bar chart. Let's start with a very simple example of a bar chart, which uses the `bar()` plotting function:

```
from matplotlib import pyplot as plt
from numpy import arange

plt.bar(arange(0,10), arange(1,21,2))
plt.show()
```



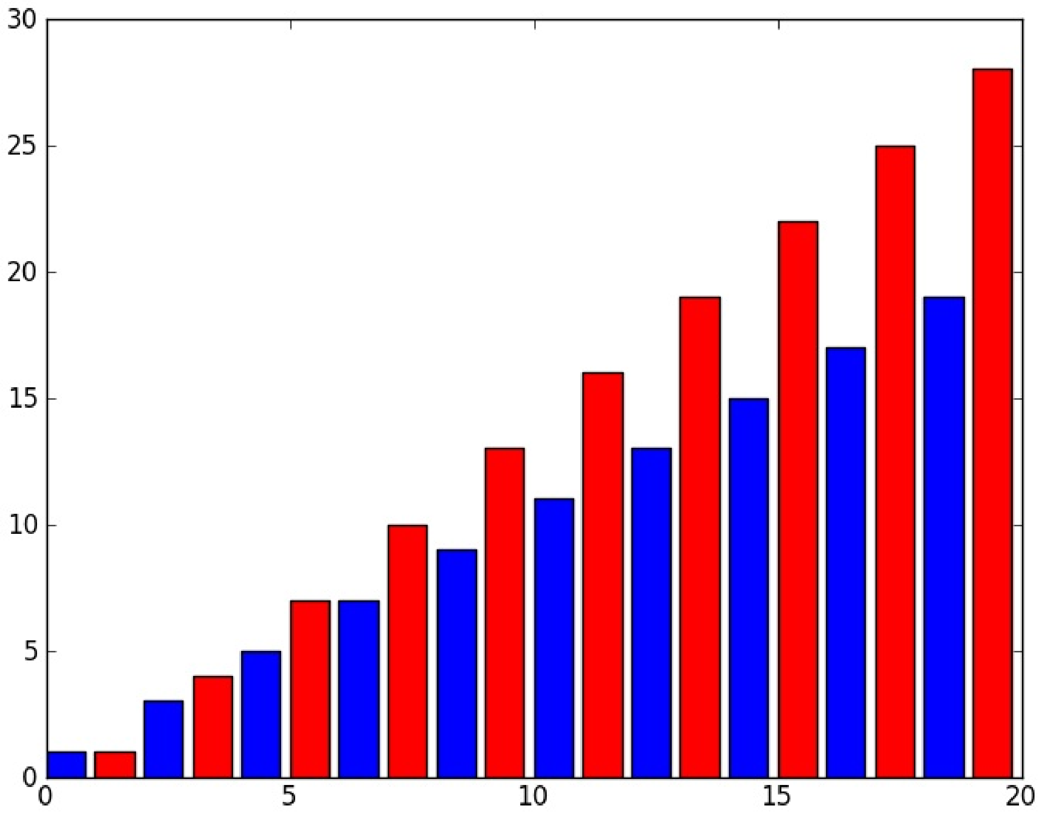
The first argument takes a list or an array of the x-axis locations for each bar's left edge; in this case, we placed the left sides of our bars along the numbers from 0 through 9. The second argument of `bar()` is a list or an array of the ordered bar values; here we supplied the odd numbers 1 through 19 for our bar heights. The bars automatically have widths of 1, although this can also be changed by setting the optional width argument:

```
from matplotlib import pyplot as plt
from numpy import arange

plt.bar(arange(0,10), arange(1,21,2), width=.5)
plt.show()
```



Often we will want to compare two or more sets of bars on a single plot. To do this, we have to space them out along the x-axis so that each set of bars appears next to each other. We can multiply and `arange()` by some factor in order to leave space, since in this case we care more about placing our bars in the correct order rather than specifying where on the x-axis they go:

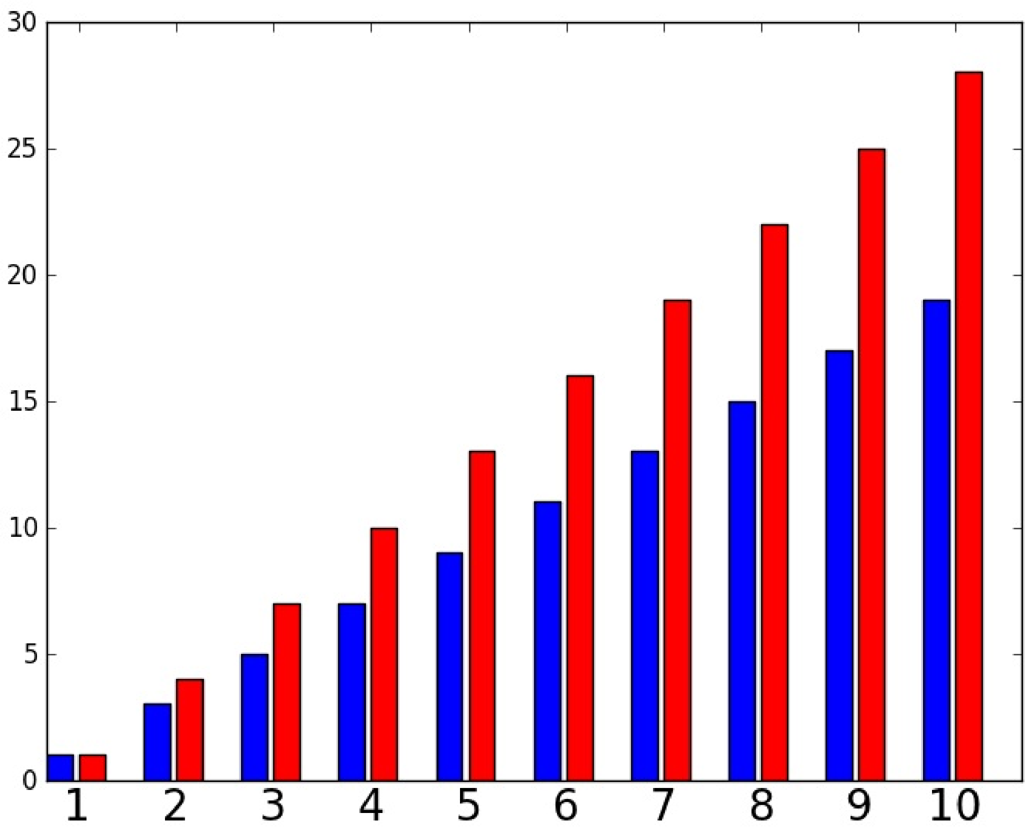
```
from matplotlib import pyplot as plt
from numpy import arange

plt.bar(arange(0,10)*2, arange(1,21,2))
plt.bar(arange(0,10)*2 + 1, arange(1,31,3), color="red")
plt.show()
```



For the x-axis of the first set of bars, we supplied the even numbers 0 through 18 by multiplying every number in our `arange()` by 2. This allows us to leave some space for the second set of bars, which we place along each of the even numbers 1 through 19.

However, the automatic numbering provided along the x-axis is meaningless now. We can change this by giving locations to label along the x-axis with the `xticks()` function. While we're at it, let's also space out each pair of bars, leaving a blank space between each grouping so that the pairs are more apparent:

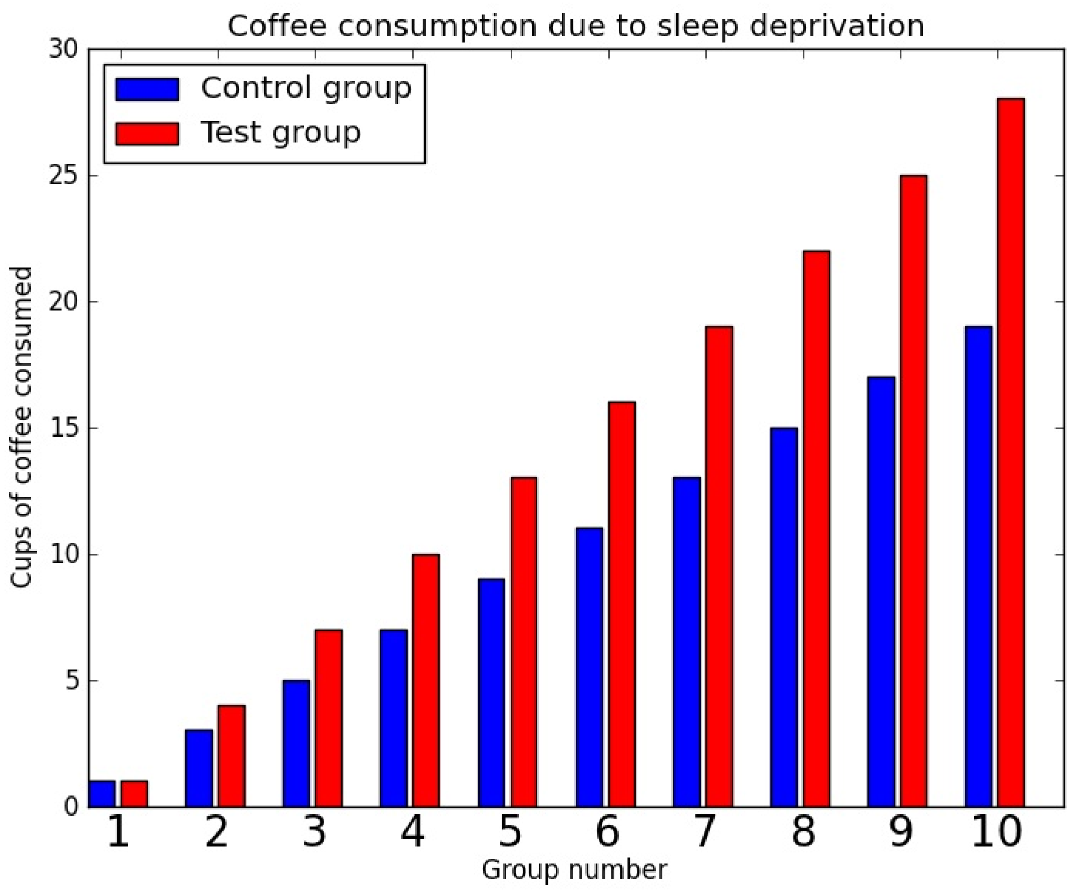
```
from matplotlib import pyplot as plt
from numpy import arange

plt.bar(arange(0,10)*3, arange(1,21,2))
plt.bar(arange(0,10)*3 + 1, arange(1,31,3), color="red")
plt.xticks(arange(0,10)*3 + 1, arange(1,11), fontsize=20)
plt.show()
```



Since we wanted to show the x-axis labels in between each of our pairs of bars, we specified the left side of the second (red) set of bars as the position to show the label. We then gave the numbers 1 through 10 as the actual labels to display; we could just as easily have used a list of strings as labels as well. Finally, we also specified a larger font size for better readability.

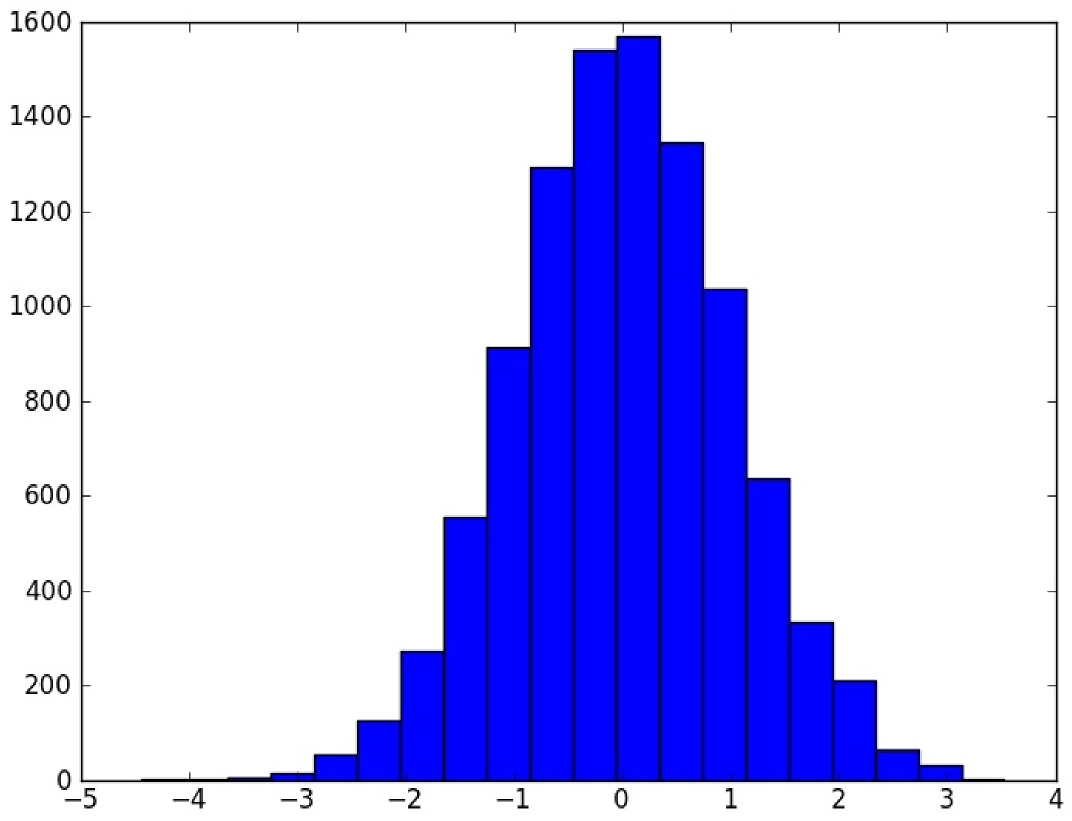
Again, we can also add axis labels, a graph title, a legend, etc. in the same way as with any other plot:

```
from matplotlib import pyplot as plt
from numpy import arange

plt.bar(arange(0,10)*3, arange(1,21,2))
plt.bar(arange(0,10)*3 + 1, arange(1,31,3), color="red")
plt.xticks(arange(0,10)*3 + 1, arange(1,11), fontsize=20)
plt.title("Coffee consumption due to sleep deprivation")
plt.xlabel("Group number")
plt.ylabel("Cups of coffee consumed")
plt.legend(("Control group", "Test group"), loc=2)
plt.show()
```



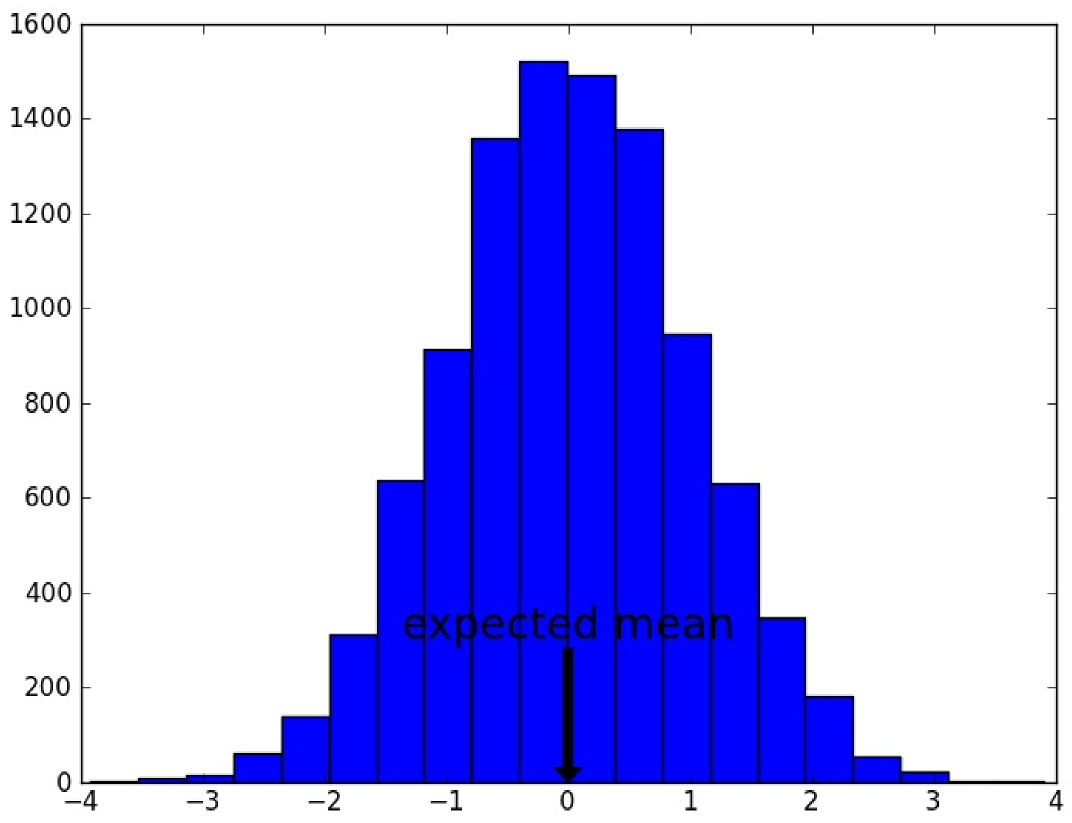
Another commonly used type of graph is the histogram, which is notoriously difficult to create in other programs like Microsoft Excel. We can make simple histograms very easily using matplotlib with the `hist()` function, which we supply with a list (or array) of values and the number of bins to use. For instance, we can create a histogram of 10,000 normally distributed (Gaussian) random numbers binned across 20 possible bars with the following, which uses NumPy's `random.randn()` function to generate an array of normal random numbers:

```
from matplotlib import pyplot as plt
from numpy import random

plt.hist(random.randn(10000), 20)
plt.show()
```



Often we want to add some text to a graph or chart. There are a number of ways to do this, but adding graph annotations is a very detailed topic that can quickly become case-specific. For one short example, let's point out the expected average value on our histogram, complete with an arrow:

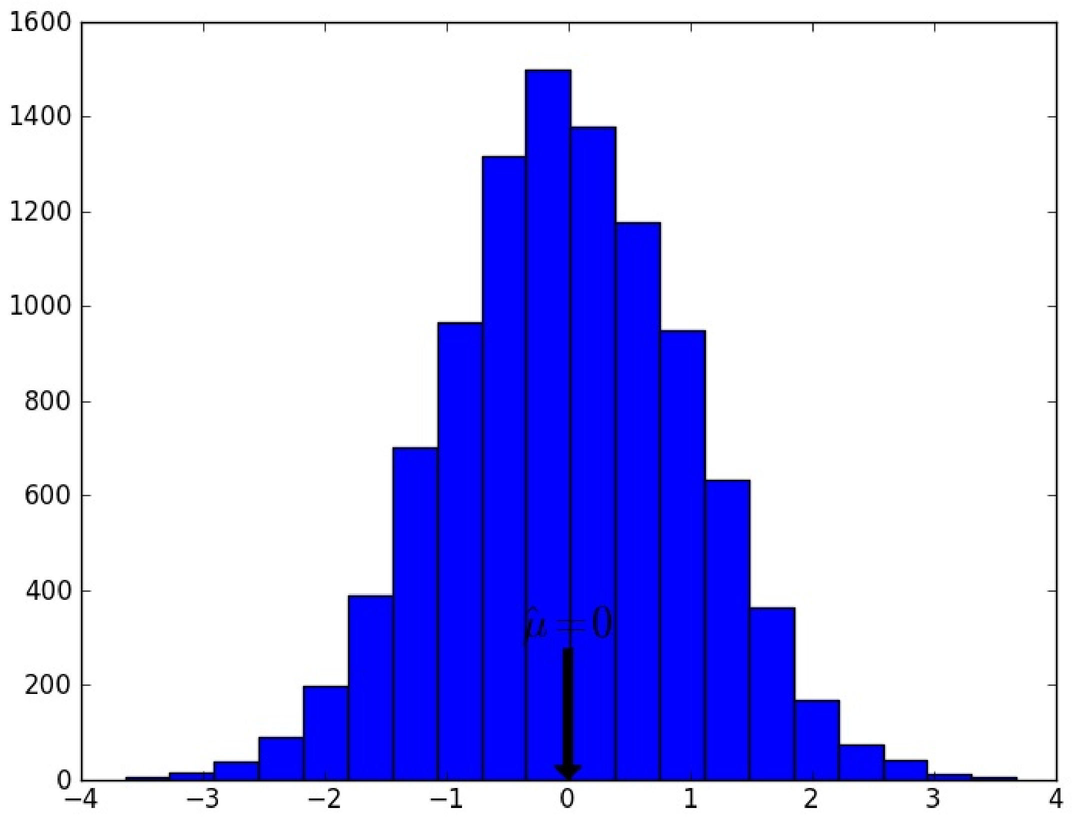
```
from matplotlib import pyplot as plt
from numpy import random

plt.hist(random.randn(10000), 20)
plt.annotate("expected mean", xy=(0, 0), xytext=(0, 300), ha="center",
    arrowprops=dict(facecolor='black'), fontsize=20)
plt.show()
```



When we call `annotate()` , first we provide the string of text that we want to appear. This is followed by the location to be annotated (i.e., where the arrow points) and the location of the text. We optionally say that the text should be centered in terms of horizontal alignment using `ha="center"` , and then we add an "arrow prop" that will point from the text (centered at the point xytext) to the annotation point (centered at the point xy). This arrow takes a dictionary of definitions, although we only provide one - namely that the arrow should be colored black. Finally, we specify a large font size of 20.

We can even include mathematical expressions in our text by using a writing style called TeX markup language. This will be familiar to you if you've ever used LaTeX, although a brief introduction for use in matplotlib can be found here. As a simple example, let's make our annotation a little more scientific by adding the symbol Î¼ with a "hat" over-line to show the predicted mean value:

```
from matplotlib import pyplot as plt
from numpy import random

plt.hist(random.randn(10000), 20)
plt.annotate(r"$\hat \mu = 0$", xy=(0, 0), xytext=(0, 300), ha="center",
    arrowprops=dict(facecolor='black'), fontsize=20)
plt.show()
```



The text expression is prefaced with an `r` to let Python know that it's a "raw" string, meaning that the backslashes should not be interpreted as special characters. Then the full TeX string is enclosed in dollar signs. Again, a full overview of TeX expressions is beyond the scope of this course, but it's usually a fairly simple matter to find a similar example to what you want to create and then modify it to suit your needs.

Once we have a chart created, chances are that we'll want to be able to save it somewhere. It turns out that this process is even easier than writing other kinds of files, because matplotlib allows us to save PNG images, SVG images, PDF documents and PostScript files by simply specifying the type of file to use and then calling the `savefig()` function. For instance, instead of displaying our histogram on the screen, let's save it out as both a PNG image and a PDF file to our chapter 14 exercise folder:

```
from matplotlib import pyplot as plt
from numpy import random

plt.hist(random.randn(10000), 20)

path = "C:/book1-exercises/chp14"
plt.savefig(path + "histogram.png")
plt.savefig(path + "histogram.pdf")
```

> **NOTE**: When using pyplot, if you want to both save a figure and display it on the screen, make sure that you save it first before displaying it! Because `show()` pauses your code and because closing the display window destroys the graph, trying to save the figure after calling `show()` will only result in an empty file.

Finally, when you're initially tweaking the layout and formatting of a particular graph, you might want to change parts of the graph without re-running an entire script to re-display the graph. Unfortunately, on some systems matplotlib doesn't work well with IDLE when it comes to creating an interactive process, but there are a couple options available if you do want this functionality. One simple work-around is simply to save out a script specifically to create the graph, then continually modify and rerun this script. Another possibility is to install the IPython package, which creates an "interactive" version of Python that will allow you to work with a graphics window that's already open. A simpler but less user-friendly solution is to run Python from the Windows command line or Mac/Linux Terminal (see the Interlude: Install Packages section for instructions on how to do this). In either case, you will then be able to turn on matplotlib's "interactive mode" for a given plot using the `ion()` function, like so:

```
>>> from matplotlib import pyplot as plt
>>> plt.ion()
>>>
```

You can then create a plot as usual, without having to call the `show()` function to display it, and then make changes to the plot while it is still being displayed by typing commands into the interactive window.

Although we've covered the most commonly used basics for creating plots, the functionality offered in matplotlib is incredibly extensive. If you have an idea in mind for a particular type of data visualization, no matter how complex, the best way to get started is usually to browse the matplotlib gallery for something that looks similar and then make the necessary modifications to the example code.

Review exercises:

1. Recreate all the graphs shown in this section by writing your own scripts without referring to the provided code
2. It is a well-documented fact that the number of pirates in the world is correlated with a rise in global temperatures. Write a script *pirates.py* that visually examines this relationship:

    - Read in the file *pirates.csv* from the Chapter 14 practice files folder.
    - Create a line graph of the average world temperature in degrees Celsius as a function of the number of pirates in the world - i.e., graph Pirates along the x-axis and Temperature along the y-axis.
    - Add a graph title and label your graph's axes.
    - Save the resulting graph out as a PNG image file.
    - Bonus: Label each point on the graph with the appropriate Year; you should do this "programmatically" by looping through the actual data points rather than specifying the individual position of each annotation.

Graphical User Interface

We've made a few pretty pictures with matplotlib and manipulated some files, but otherwise we have limited ourselves to programs that are generally invisible and occasionally spit out text. While this might be good enough for most purposes, there are some programs that could really benefit from letting the user "point and click" - say, a script you wrote to rename a folder's worth of files that your technologically impaired friend now wants to use. For this, we need to design a graphical user interface (referred to as a GUI and pronounced "gooey" - really).

When we talk about GUIs, we usually mean a full GUI application where everything about the program happens inside a window full of visual elements (as opposed to a text-based program). Designing good GUI applications can be incredibly difficult because there are so many moving parts to manage; all the pieces of an application constantly have to be listening to each other and to the user, and you have to keep updating all the visual elements so that the user only sees the most recent version of everything.

Instead of diving right into the complicated world of making GUI applications, let's first add some individual GUI elements to our code. That way, we can still improve the experience for the person using our program without having to spend endless hours designing and coding it.

Add GUI elements with EasyGUI

We'll start out in GUI programming with a module named EasyGUI. First, you'll need to install this package via pip3:

```
$ pip3 install easygui
```

Or download and install manually:

Windows: Download the compressed .zip file, unzip it, and install the package by running Python on setup.py from the command prompt as described in the installation section.

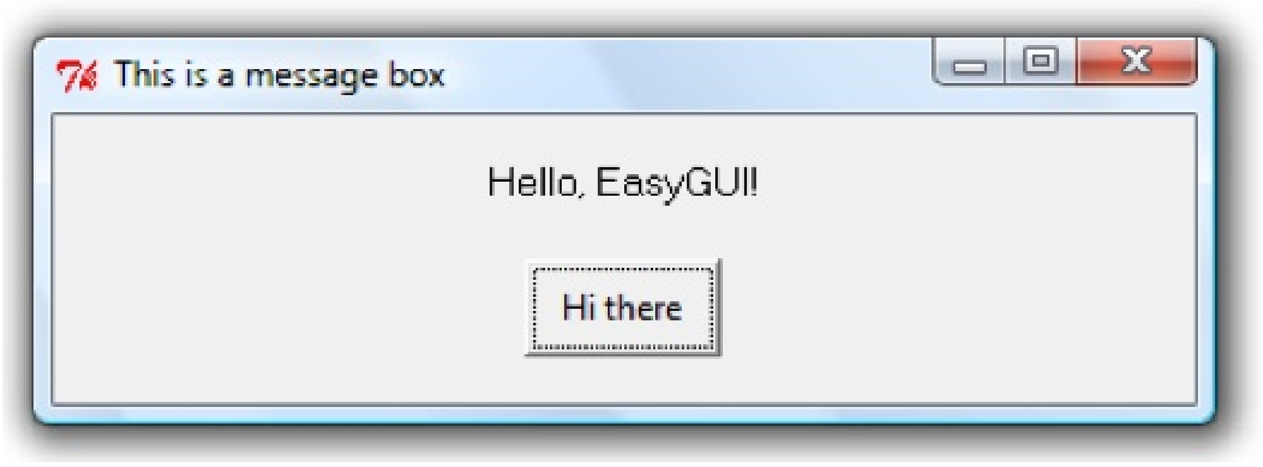
OS X: Download the compressed .tar.gz file, decompress it, then install the package by running Python on setup.py from Terminal as described in the installation section.

Debian/Linux: `sudo apt-get install python-easygui`

EasyGUI is different from other GUI modules because it doesn't rely on events. Most GUI applications are event-driven, meaning that the flow of the program depends on actions taken by the user; this is usually what makes GUI programming so complicated, because any object that might change in response to the user has to "listen" for different "events" to occur. By contrast, EasyGUI is structured linearly like any function; at some point in our code, we display some visual element on the screen, use it to take input from the user, then return that user's input to our code and proceed as usual.

Let's start by importing the functionality from EasyGUI into the interactive window and displaying a simple message box:

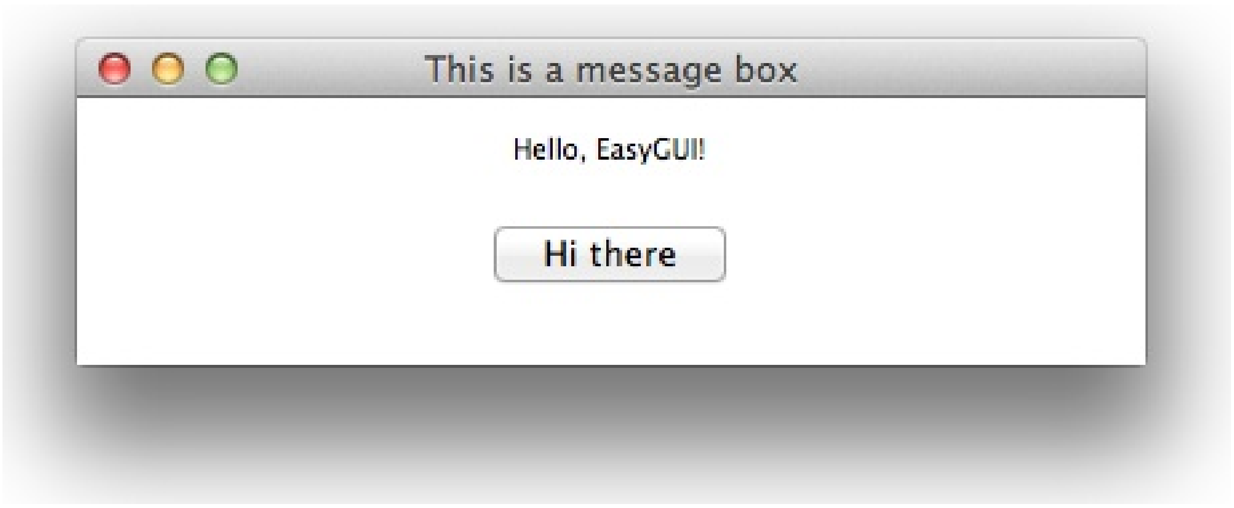
```
>>> from easygui import *
>>> msgbox("Hello, EasyGUI!", "This is a message box", "Hi there")
```



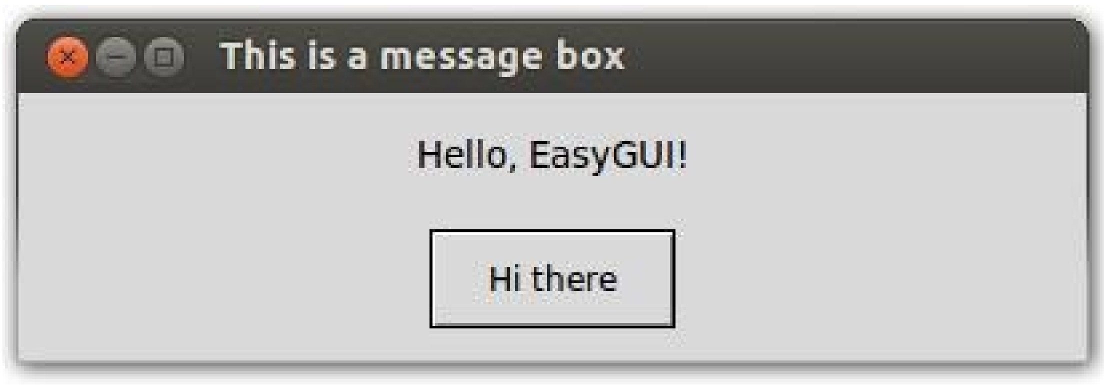
```
'Hi there'
>>>
```

When you run the second line of code, something like the window above should have appeared.

On Mac, it will look more like this:



And in Ubuntu:



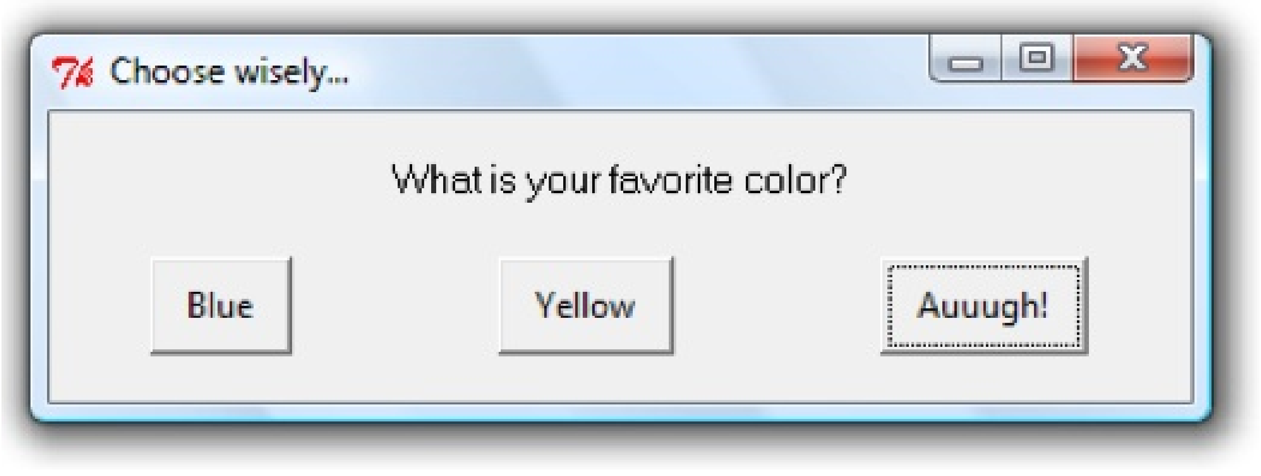
If nothing appears, see the box below. For the sake of the majority, I'll be sticking to screenshots from a Windows GUI perspective only.

We used the `msgbox()` to generate a new box that includes a message, a window title, and button text that we provided, in that order. (The default value for the button text would be "OK" if we hadn't provided a third argument.) When the user clicks the button, EasyGUI returns the value of the button's text, and we're back to the interactive prompt.

> **NOTE**: EasyGUI uses a toolkit called Tkinter, which we will get to use in the next section. IDLE uses Tkinter to run as well. You might run into problems with freezing windows, etc., because of disagreements between the new windows you create and the IDLE window itself. If you think this might be happening, you can always try running your code or script by running Python from the command prompt (Windows) or Terminal (Mac/Linux).

Notice that when we ran the previous code, clicking the button returned the value that was in the button text back to the interactive window. In this case, returning the text of the button clicked by the user wasn't that informative, but we can also provide more than one choice by using a `buttonbox()` and providing a tuple of button values:

```
>>> choices = ("Blue", "Yellow", "Auuugh!")
>>> buttonbox("What is your favorite color?", "Choose wisely...", choices)
```



Choose wisely...

What is your favorite color?

```
'Auuugh!'
>>>
```

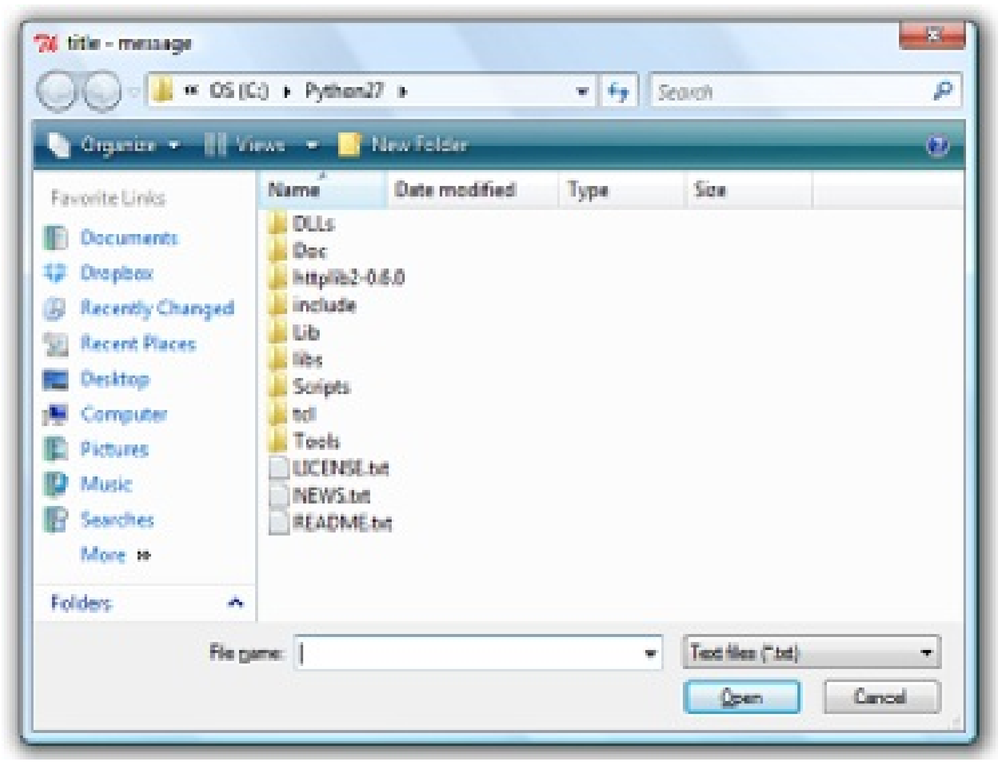
Now we were able to tell that our user chose "Auuugh!" as the favorite color, and we could set that return value equal to a variable to be used later in our code.

There are a number of other ways to receive input from the user through easyGUI to suit your needs. For starters, try out the following lines a few times each and see what values they return based on the different choices you select:

```
>>> choices = ("Blue", "Yellow", "Auuugh!") # tuple of choices
>>> title = "Choose wisely..." # window title
>>> indexbox("What is your favorite color?", title, choices)
>>> choicebox("What is your favorite color?", title, choices)
>>> multchoicebox("What are your favorite colors?", title, choices)
>>> enterbox("What is your favorite color?", title)
>>> passwordbox("What is your favorite color? (I won't tell.)", title)
>>> textbox("Please describe your favorite color:")
```

Another useful feature provided by EasyGUI is a simple way for the user to select a file through a standard file dialog box:

```
>>> fileopenbox("message", "title", "*.txt")
```



The third argument we passed to `fileopenbox()` was the type of file we want the user to open, which we specify in a string by using a "wildcard" *symbol followed by the name of the file extension. This automatically filters the viewable files to those that match the ".txt" extension, although the user still has the option to change this box to "All files (.*)"* and select any type of file.

Notice how we still provided a "message" and a "title" even though they both appeared in the title bar; typically you will just want to pass an empty string to one of these, since a single title is enough. If we look up the easyGUI documentation to find out the names of the variables used by the function `fileopenbox()`, we can also provide only specific arguments by directly assigning values to the variable names, like so:

```
>>> fileopenbox(title="Open a file...", default="*.txt")
```

Within a "file open" dialog box, try typing in the name of a file that doesn't exist and opening it; the dialog box won't let you! This is one less thing that we have to worry about programming into our code. The user can still hit cancel, in which case `None` is returned to represent a lack of any object. Otherwise, `fileopenbox()` gives us a string representing the full path to the selected file. Keep in mind that we aren't actually opening this file in any way; we're simply presenting the user with the ability to select a file for opening, but the rest of the code is still up to us.

There is also a `diropenbox()` function for letting the user choose a folder rather than an individual file; in this case, the optional third argument can tell the dialog box which directory to have open by default.

Finally, there is a `filesavebox()` that works similarly to `fileopenbox()` except that it allows the user to select a file to be "saved" rather than opened. This dialog box also confirms that the user wants to overwrite the file if the chosen name is the same as a file that already exists. Again, no actual saving of files is happening - that's still up to us to program once we receive the file name from easyGUI.

In practice, one of the most difficult problems when it comes to letting the user select files is what to do if the user cancels out of the window when you need to have selected a file. One simple solution is to display the dialog in a loop until the user finally does select a file, but that's not very nice - after all, maybe your user had a change of heart and really doesn't want to run whatever code comes next.

Instead, you should plan to handle rejection gracefully. Depending on what exactly you're asking of the user, most of the time you should use `exit()` to end the program without a fuss when the user cancels. (If you're running the script in IDLE, `exit()` will also close the current interactive window. It's very thorough.)

Let's get some practice with how to handle file dialogs by writing a simple, usable program. We will guide the user (with GUI elements) through the process of opening a PDF file, rotating its pages in some way, and then saving the rotated file as a new PDF:

```
from easygui import *
from pyPDF2 import PdfFileReader, PdfFileWriter

# let the user choose an input file
input_file_name = fileopenbox("", "Select a PDF to rotate...", "*.pdf")
if input_file_name is None: # exit on "Cancel"
    exit()

# let the user choose an amount of rotation
rotate_choices = (90, 180, 270)
message = "Rotate the PDF clockwise by how many degrees?"
degrees = buttonbox(message, "Choose rotation...", rotate_choices)

# let the user choose an output file
output_file_name = filesavebox("", "Save the rotated PDF as...", "*.pdf")
while input_file_name == output_file_name: # cannot use same file as input
    msgbox("Cannot overwrite original file!", "Please choose another file...")
    output_file_name = filesavebox("", "Save the rotated PDF as...", "*.pdf")

if output_file_name is None:
    exit() # exit on "Cancel"

# read in file, perform rotation and save out new file
input_file = PdfFileReader(open(input_file_name, "rb"))
output_PDF = PdfFileWriter()

for page_num in range(0, input_file.getNumPages()):
    page = input_file.getPage(page_num)
    page = page.rotateClockwise(int(degrees))
    output_PDF.addPage(page)

output_file = open(output_file_name, "wb")
output_PDF.write(output_file)
output_file.close()
```

Besides the use of `exit()`, you should already be familiar with all the code in this script. The tricky part is the logic - i.e., how to put all the different pieces together to create a seamless experience for the user. For longer programs, it can be helpful to draw out diagrams by hand using boxes and arrows to represent how we want the user to experience the different parts of our program; it's a good idea to do this even before you start writing any code at all. Let's represent how this script works in an outline form in terms of what we display to the user:

1. Let the user select an input file
2. If the user canceled the "open file" dialog (None was returned), then exit the program
3. Let the user select a rotation amount

   - [No alternative choice here; the user must click a button]
4. Let the user select an output file
5. If the user tries to save a file with the same name as the input file:
   - Alert the user of the problem with a message box
   - Return to step 4
6. If the user canceled the "save file" dialog (None was returned), then exit the program

The final steps are the hardest to plan out. After step 4, since we already know (from step 2) that the input file isn't `None`, we can check whether the output file and input file match before checking for the canceled dialog. Then, based on the return value from the dialog, we can check for whether or not the user canceled the dialog box after the fact.

Review exercises:

1. Recreate the three different GUI elements pictured in this section by writing your own scripts without referring to the provided code
2. Save each of the values returned from these GUI elements into new variables, then print each of them
3. Test out `indexbox()`, `choicebox()`, `multchoicebox()`, `enterbox()`, `passwordbox()` and `textbox()` to see what GUI elements they produce; you can use the `help()` function to read more about each function in the interactive window - for instance, type `import easygui` then `help(easygui.indexbox)`

Assignment: Use GUI elements to help a user modify files

Write a script *partial_PDF.py* that extracts a specific range of pages from a PDF file based on file names and a page range supplied by the user. The program should run as follows:

1. Let the user choose a file using a fileopenbox
2. Let the user choose a beginning page to select using an `enterbox`
3. If the user enters invalid input, use a `msgbox` to let the user know that there was a problem, then ask for a beginning page again using an `enterbox`
4. Let the user choose an ending page using another `enterbox`
5. If the user enters an invalid ending page, use a `msgbox` to let the user know that there was a problem, then ask for a beginning page again using an `enterbox`
6. Let the user choose an output file name using a `savefilebox`
7. If the user chooses the same file as the input file, let the user know the problem using a `msgbox`, then ask again for a file name using a `savefilebox`
8. Output the new file as a section of the input file based on the user- supplied page range. The user should be able to supply "1" to mean the first page of the document. There are a number of potential issues that your script should be able to handle. These include:
    - If the user cancels out of a box, the program should exit without any errors
    - Check that pages supplied are valid numbers; you can use the string method `isdigit()` to check whether or not a string is a valid positive integer. (The `isdigit()` method will return `False` for the string "0" as well.)
    - Check that the page range itself is valid (i.e., the end to the start)

Create GUI Application with Tkinter

For many basic tasks where GUI elements are needed one at a time, EasyGUI can save a lot of effort compared to creating an entire GUI program. If you do want to build a complete GUI application with many interacting elements, it will take significantly more code (and time spent programming), but this section will help get you started.

There are a lot of different tools available for GUI application design in Python. The simplest and most commonly used framework is the Tkinter module, which comes with Python by default. (In fact, easyGUI is really just a simplified way of accessing Tkinter in small, manageable pieces.) There are more advanced toolkits that can be used to produce more complicated (and eventually better-looking) GUIs, and we will touch on these at the end of the chapter. However, Tkinter is still the module of choice for many GUI projects because it is so lightweight and relatively easy to use for simple tasks compared to other toolkits. In fact, IDLE itself uses Tkinter!

> **NOTE**: As mentioned in the last section, because IDLE is also built with Tkinter, you might encounter difficulties when running your own GUI programs within IDLE. If you find that the GUI window you are trying to create is unexpectedly freezing or appears to be making IDLE misbehave in some unexpected way, try running your script in Python via your command prompt (Windows) or Terminal (Mac/Linux) to check if IDLE is the real culprit.

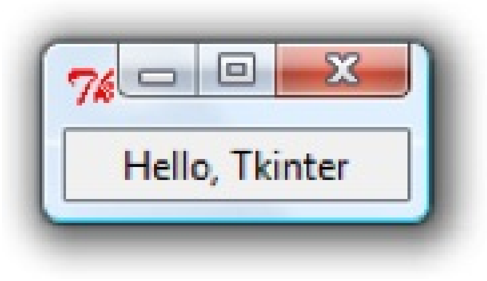
GUI applications exist as windows, which are just the application boxes you're used to using everywhere, each one with its own title, minimize button, close button, and usually the ability to be resized. Within a window we can have one or more frames that contain the actual content; the frames help to separate the window's content into different sections. Frames, and all the different objects inside of them (menus, text labels, buttons, etc.) are all called widgets.

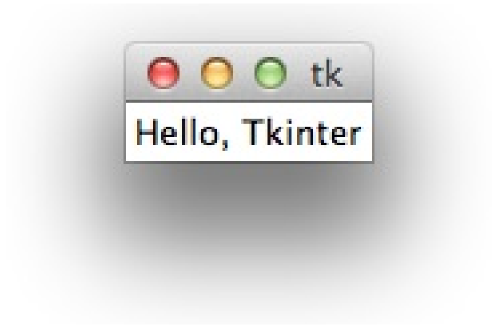
Let's start with a window that only contains a single widget. In this case, we'll use a `Label` to show a single string of text:

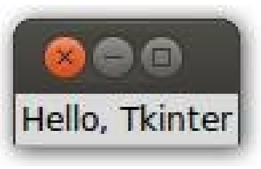
```
from tkinter import *

my_app = Tk() # create a new window

greeting = Label(text="Hello, Tkinter") # create a text label
greeting.pack() # add the label to the window

my_app.mainloop() # run the application
```

This is already significantly more complicated than easyGUI! First, we had to import functionality from the Tkinter module. Then we created a new Tkinter window by calling `Tk()` and made a `Label` object containing some text. To actually add the `Label` to our app window and make it visible, we had to call the `pack()` method on the `Label`, which "packs" it into the window. Finally, we called the `mainloop()` method to make our `my_app` window and its contents visible and begin the processing of running the GUI application. Although it can't do anything interesting yet, it does function; the application can be resized, minimized, and closed.

If you run this script, a window should appear that will look something like one of the following, depending on your operating system:

Windows: 

Mac: 

Ubuntu: 

Again, for the sake of the majority, I'll stick to displaying only the resulting Windows application version for the remainder of this section.

Usually, the layout of even a very basic complete GUI application will be much more complicated than our last example. Although it may look intimidating when starting out, keep in mind that all of the initial work is helpful to make it much easier once we want to add additional features to the application. A more typical setup for a (still blank) GUI application might look something more like this:

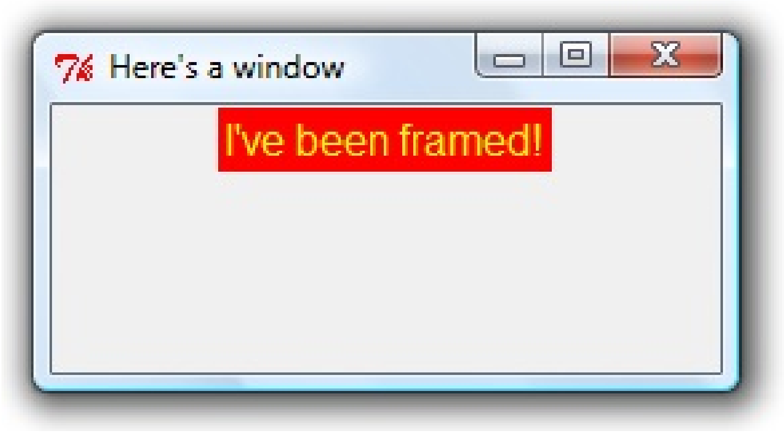
```
from tkinter import *

# define the GUI application
class App(Frame):
    def __init__(self, master):
        Frame.__init__(self, master)

# create the application window
window = Tk()
window.geometry("400x200") # default window size

my_app = App(window)
my_app.master.title("I made a window!")
my_app.mainloop() # start the application
```

Although this type of setup is necessary for expanding GUI applications to more complex functionality, the code involved is beyond the scope of this brief introduction. Instead, we will restrict ourselves to a simpler setup that is still completely functional but less efficient for building advanced user interfaces. For insight into what the above code is doing have a look at *Appendix C: Primer on Object-Oriented Programming in Python*.

Let's start with a simple window again, but this time add a single frame into it. We'll set a default window size, and the one frame will take up the entire window. We can then add widgets to the frame as well, such as a Label with text. This time, we'll specify background (`bg`) and foreground (`fg`) colors for the `Label` text as well as a font:

```
from tkinter import *

window = Tk()
window.title("Here's a window")
window.geometry("250x100") # default window size

my_frame = Frame()
my_frame.pack()

label_text = Label(my_frame, text="I've been framed!",
                   bg="red", fg="yellow", font="Arial")
label_text.pack()

window.mainloop()
```

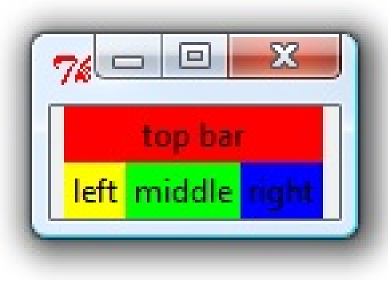
The result is very similar to our first window, since the frame itself isn't a visible object:



When we created the label, we had to assign the label to the frame by passing the name of our frame, `my_frame`, as the first argument of this `Label` widget. This is important to do because we're otherwise packing the label into the window, and anything we do to modify the frame's formatting won't be applied to widgets that don't specifically name the frame to which they belong. The frame is called the parent widget of the label since the label is placed inside of it. This becomes especially important once we have multiple frames and have to tell Tkinter which widgets will appear in which frames.

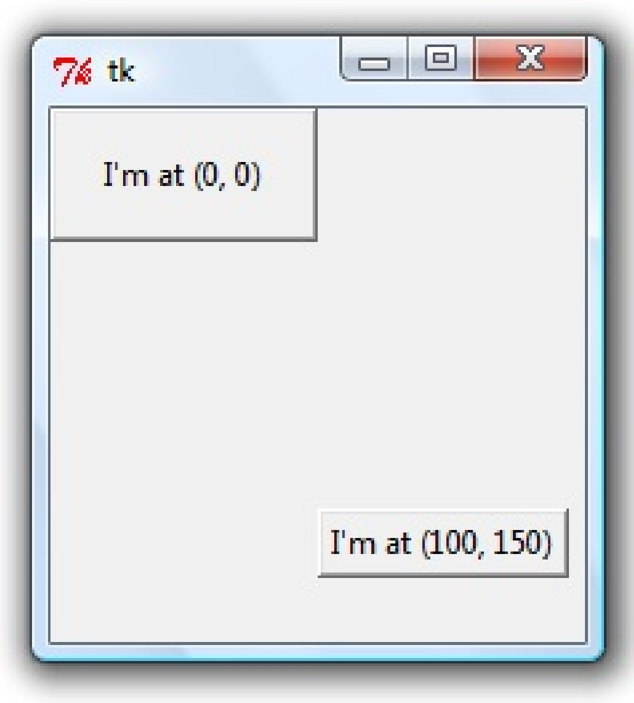
There are a few different ways we can organize the packing of widgets into a frame. For instance, running the same script as above but packing our `Label` with label_text.pack(fill=X) will make the `Label` span across the entire width of the frame. Widgets stack vertically on top of each other by default, but we can pack them side-by-side as well:

```python
from tkinter import *

window = Tk()

my_frame = Frame()
my_frame.pack()

# a bar to span across the entire top

label_text1 = Label(my_frame, text="top bar", bg="red")
label_text1.pack(fill=X)

# three side-by-side labels
label_left = Label(my_frame, text="left", bg="yellow")
label_left.pack(side=LEFT) # place label to the left of the next widget

label_mid = Label(my_frame, text="middle", bg="green")
label_mid.pack(side=LEFT) # place label to the left of the next widget

label_right = Label(my_frame, text="right", bg="blue")
label_right.pack()

window.mainloop()
```



It can quickly get difficult to figure out how to arrange things using `pack()`, but there are two other options available as well for getting your widgets organized. The `place()` method can be used to place a widget in an exact location within a frame based on specific x and y coordinates, where the point (0, 0) is the upper left corner of the frame. At the same time, we can also specify each widget's width and height in pixels. For instance, let's `place()` a couple buttons in a frame:

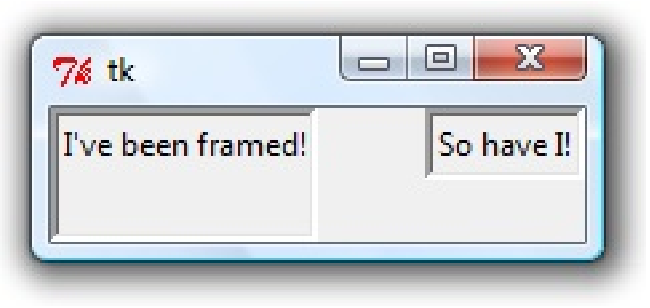
```
from tkinter import *

window = Tk()
window.geometry("300x200")

button1 = Button(window, text="I'm at offset (50, 60)")
button2 = Button(window, text="I'm at offset (0, 0)")

button1.pack()
button2.pack()

button1.place(height=200, width=200, x=50, y=65)
button2.place(height=150, width=150, x=0, y=0)

window.mainloop()
```



First we set a specific window size, which is 300 pixels wide and 200 pixels tall. We created `button1` and `button2` using `Button()`, made them children of `window` and gave them some text. Next, we used `pack()` to put them into our `window`. Using `place()`, we gave each button a size and positioned them at specific (x, y) coordinates within the window. Finally, we started the program with `mainloop()`.

Other than specifying absolute placement and size (meaning that we give exact amounts of pixels to use), we can instead provide relative placement and size of a widget. For instance, since frames are also widgets, let's add two frames to a window and place them based on relative positions and sizes:

```
from tkinter import *

window = Tk()
window.geometry("200x50") # window is 200 pixels wide by 50 pixels tall

# create side-by-side frames
frame_left = Frame(bd=3, relief=SUNKEN) # give the frame an outline
frame_left.place(relx=0, relwidth=0.5, relheight=1)

frame_right = Frame(bd=3, relief=SUNKEN) # give the frame an outline
frame_right.place(relx=0.7, relwidth=0.3)

# add a label to each frame
left_label = Label(frame_left, text="I've been framed!")
left_label.pack()

right_label = Label(frame_right, text="So have I!")
right_label.pack()

window.mainloop()
```



The extra parameters we passed to each frame provided a border (`bd`) and set that border in "sunken relief" so that we could actually see the edges of the frames, since by default the frames wouldn't have been visible by themselves.

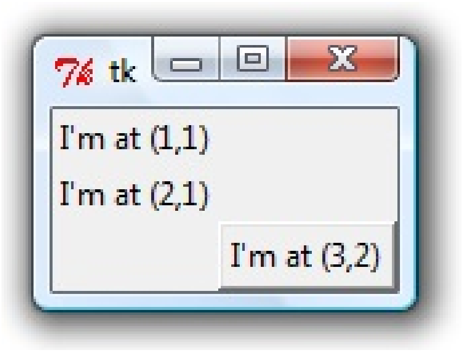
This time we used `relx` to provide a relative x placement, which takes a value from 0 to 1 to represent a fraction of the window's width. The first label has `relx=0` , so it appears all the way at the left of the window. Since we gave the second label a `relx=.7` and `relwidth=.3` , it appears 70% of the way across the window and takes up the remaining 30% of the space without running off the end of the window. We specified a `relheight=1` for the first label so that it took up the entire height of the window. Since we didn't name a `relheight` value (and didn't even provide a `rely` value) for the second label, it defaulted to appear at the top of the window with only the height necessary to display the text inside of it. Try modifying all these values one by one to see how they change the appearance of the frames.

> **NOTE**: Notice how we had to pass the name of each frame as the first argument of each label. This is how Tkinter tracks which widget belongs in which frame. Otherwise, we wouldn't know which frame our label belongs to and couldn't tell where to pack and display it. Even if you only have a single frame, however, since relative placement/sizing is done inside of the parent widget, you should always be sure to name the frame to which each widget belongs.

Although it offers more detailed control than `pack()` does, using `place()` to arrange widgets usually isn't an ideal strategy, since it can be difficult to update the specific placements of everything if one widget gets added or deleted. Beyond this, different screen resolutions can make a window appear somewhat differently, making your careful placements less effective if you want to share the program to be run on different computers.

The last and usually the easiest way to make clean, simple GUI layouts without too much hassle is by using `grid()` to place widgets in a two-dimensional grid. To do this, we can imagine a grid with numbered rows and columns, where we can then specify which cell (or cells) of our grid we want to be taken up by each particular widget.

```python
from tkinter import *

window = Tk()

my_frame = Frame()
my_frame.grid() # add frame to take up the whole window using grid()

label_top_left = Label(my_frame, text="I'm at (1,1)")
label_top_left.grid(row=1, column=1)

label_bottom_left = Label(my_frame, text="I'm at (2,1)")
label_bottom_left.grid(row=2, column=1)

button_bottom_right = Button(my_frame, text="I'm at (3,2)")
button_bottom_right.grid(row=3, column=2)

window.mainloop() # start the application
```

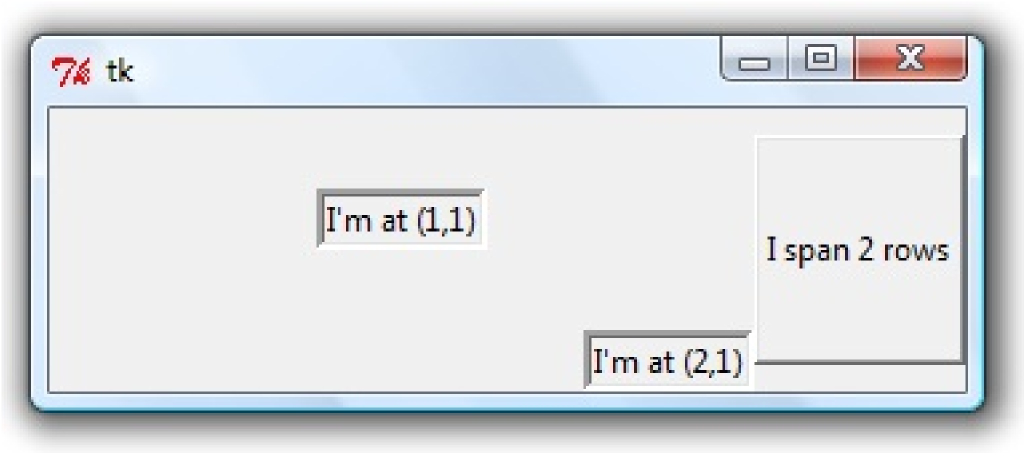


Instead of calling `pack()` on our frame widget, we put the frame into the window by using `grid()`, which will let Tkinter know that we plan to place widgets inside this frame by also using `grid()` instead of another method. We could then call `grid()` on each widget instead of `pack()` or `place()` by providing a row and column numbers, which start at `row=1` and `column=1` in the upper-left corner of the window.

> **WARNING**: Don't try to combine different ways of adding widgets into a single frame - for instance, don't use `grid()` on some widgets and `pack()` on others. Tkinter has no way to prioritize one method over the other and usually ends up freezing your application entirely when you've given it conflicting packing instructions.

We can assign values to the arguments `padx` and `pady` for any given widget, which will include extra space around the widget horizontally and/or vertically. We can also assign values to the argument sticky for each widget, which takes cardinal directions (like a compass) such as `E` or `N+W`; this will tell the widget which side (or sides) it should "stick" to if there is extra room in that cell of the grid. Let's look at these arguments in an example:

```
from tkinter import *

window = Tk()

my_frame = Frame()
my_frame.grid() # add frame to window using grid()

label_top_left = Label(my_frame, text="I'm at (1,1)", bd="3", relief=SUNKEN)
label_top_left.grid(row=1, column=1, padx=100, pady=30)

label_bottom_left = Label(my_frame, text="I'm at (2,1)", bd="3", relief=SUNKEN)
label_bottom_left.grid(row=2, column=1, sticky=E)

button_right = Button(my_frame, text="I span 2 rows", height=5)
button_right.grid(row=1, column=2, rowspan=2)

window.mainloop() # start the application
```



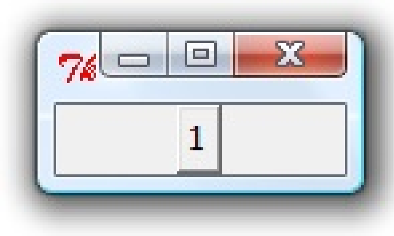
We included a border and `sunken` relief to the two labels (just like we did previously for the frame) so that we could see their outlines. Because we gave large `padx` and `pady` values to the first label, it appears with lots of extra space around it, centered in the middle of the first grid cell. The second label appears in the second row and first column, right underneath the first label, but this time we specified `sticky=E` to say that the label should stick to the east side of the grid cell if there is extra horizontal space.

We added a button in the second column, specifying a height of 5 (text lines). This allows us to see that, even though we placed the button at `row=1`, since we also specified `rowspan=2`, the button is centered vertically across both the first and second rows.

> **NOTE**: Since it often takes a fair amount of effort to get widgets arranged in a window exactly as you'd like, it's usually a good idea to draw out a mock-up of what you want your GUI application to look like on paper before doing any coding at all. With a physical reference, it will be much easier to translate what's in your head into GUI widgets organized in frames.

The buttons we've created so far aren't very useful, since they don't yet do anything when we click on them. However, we can specify a command argument to a button that will run a function of our choice. For instance, let's create a simple function `increment_button()` that takes the number displayed on our button and increments it by one:

```
from tkinter import *

window = Tk()

def increment_button():
    new_number = 1 + my_button.cget("text")
    my_button.config(text=new_number)

my_button = Button(text=1, command=increment_button)
my_button.pack()

window.mainloop() # start the application
```

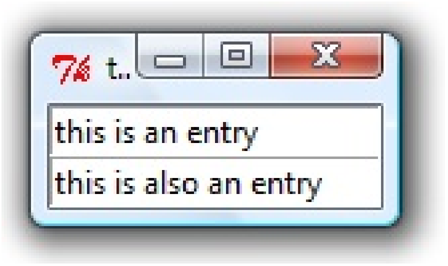


Now each time you click on the button, it will add one to the value by configuring the button with a new text attribute. We used the `cget()` method on the button to retrieve the text currently displayed on the button; we can use `cget()` in this same way to get the values taken by other attributes of other types of widgets as well.

There are a number of other types of widgets available in Tkinter as well; a good quick reference can be found here. Besides labels and buttons, one very commonly used type of widget is an `Entry`, which allows a place for the user to enter a line of text.

An entry works a little differently from labels and buttons in that the text in the entry box is blank at first. If you want to add "default" text to display, it must be inserted after creating the entry using the `insert()` method, which requires as its first argument a position in the text for inserting the new string. For instance, after creating a new `Entry` with `myEntry = Entry()` , we would then say `myEntry.insert(0, "default text")` to add the string "default text" to the entry box. In order to return text currently in the entry box, we use the `get()` method instead of the usual `cget()` method. Both these concepts are easier seen in an example:

```
from tkinter import *

window = Tk()

entry1 = Entry()
entry1.pack()
entry1.insert(0, "this is an entry")

entry2 = Entry()
entry2.pack()

my_text = entry1.get() # get the text from entry1
entry2.insert(0, my_text)
entry2.insert(8, "also ") # add "also" to the middle of my_text

window.mainloop() # start the application
```

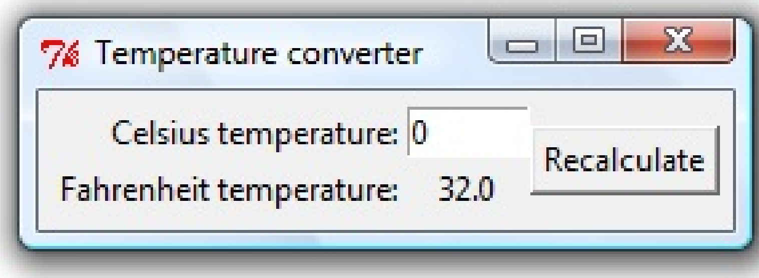


```
this is an entry
this is also an entry
```

Here we packed two entry boxes into a window, adding new text at index position 0 to `entry1` first. We added the text seen in `entry2` by first using `get()` to return the text in `entry1` , inserting this same text into `entry2` , then inserting additional text into this string starting at index location 8. We can also specify a width (in characters) that we want an entry box to take up by passing a value to width when we create the Entry.

Let's put all of these GUI pieces together into a single usable application. We'll modify the first function that we wrote way back in the **Functions and Loops** chapter to create a GUI-based temperature conversion application:

```
from tkinter import *

def recalc():
    cel_temp = entry_cel.get() # get temp from text entry
    try: # calculate converted temperature and place it in label
        far_temp = float(cel_temp) * 9/5 + 32
        far_temp = round(far_temp, 3) # round to three decimal places
        result_far.config(text=far_temp)
    except ValueError: # user entered non-numeric temperature
        result_far.config(text="invalid")

# create the application window and add a Frame
window = Tk()
window.title("Temperature converter")

frame = Frame()
frame.grid(padx=5, pady=5) # pad top and left of frame 5 pixels before grid

# create and add text labels
label_cel = Label(frame, text="Celsius temperature:")
label_cel.grid(row=1, column=1, sticky=S+E)

label_far = Label(frame, text="Fahrenheit temperature:")
label_far.grid(row=2, column=1, sticky=S+E)

# create and add space for user entry of text
entry_cel = Entry(frame, width=7)
entry_cel.grid(row=1, column=2)
entry_cel.insert(0, 0)

# create and add label for text calculation result
result_far = Label(frame)
result_far.grid(row=2, column=2)

# create and add 'recalculate' button
btn_recalc = Button(frame, text="Recalculate", command=recalc)
btn_recalc.grid(row=1, column=3, rowspan=2)

recalc() # fill in first default temperature conversion

window.mainloop() # start the application
```



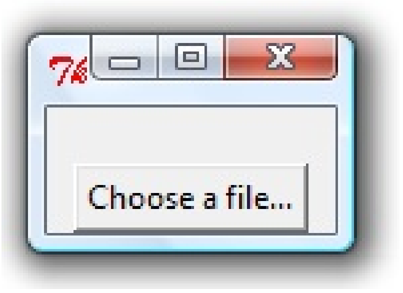
First we created the function `recalc()`, which gets the Celsius temperature from the entry box, converts it into Fahrenheit, and sets that (rounded) number as the text of the Fahrenheit temperature label. Notice how we used the `try/except` structure to check if the user entered something that isn't a number so that we can display "invalid" as the converted temperature instead of raising an error.

The uses of GUI widgets should all look familiar to you now. We used a `grid()` layout with the value `S+E` to the sticky argument of our labels so that they would be right-aligned. The button, which is centered across two rows, calls our `recalc()` function so that the current Celsius temperature is converted and redisplayed in the Fahrenheit label each time the button is clicked; the actual button click by the user is considered an event, and by passing a function name to the command argument we ensure that the button is "listening" for this event so that it can take an action. We can also call the function ourselves, which we do before running the program in order to fill in a default converted value to our Fahrenheit label.

Finally, we can also use `filedialog` to allow the user to open and save files, much like we did before with EasyGUI. Although the `filedialog` module is part of the `tkinter` module, we must import it separately.

Since we aren't presenting GUI elements one at a time, however, file dialogs in Tkinter will usually be linked to other specific GUI widgets - for instance, having the user click on a button in order to pop up a file dialog box. Let's write a quick script that uses an "Open" file dialog to let the user select either a ".PY" file or a ".TXT" file and then displays the full name of that file back in a label:

```
from tkinter import *
from tkinter import filedialog


window = Tk()

frame = Frame()
frame.pack()

def open_file(): # ask user to choose a file and update label
    type_list = [("Python scripts", "*.py"), ("Text files", "*.txt")]
    file_name = filedialog.askopenfilename(filetypes=type_list)
    label_file.config(text=file_name)

# blank label to hold name of chosen file
label_file = Label(frame)
label_file.pack()

# button to click on for 'Open' file dialog
button_open = Button(frame, text="Choose a file...", command=open_file)
button_open.pack()

window.mainloop() # start the application
```



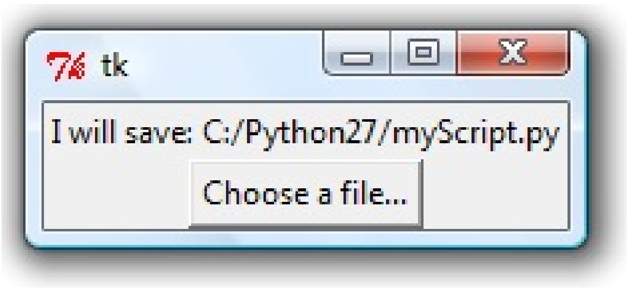
Clicking the button will cause our function `open_file()` to run, which opens a file dialog using `filedialog.askopenfilename()` .

We passed `type_list` into the argument `filetypes` of the `filedialog.askopenfilename()` function in order to provide a list of the different types of files that we want the user to be able to choose; these are provided as a list of tuples, where the first item in each tuple is a description of the file type and the second item in each tuple is the actual file extension.

Just like with EasyGUI, the `filedialog.askopenfilename()` function returns the full name of the file, which we can then set equal to our string `file_name` and pass into our label. If the user hit the "Cancel" button instead, the function returns `None` and our label becomes blank.

Likewise, there is a `filedialog.asksaveasfilename()` function that takes the same arguments as `filedialog.askopenfilename()` and works analogously to the `filesavebox()` of EasyGUI. We can also pass a default extension type to `asksaveasfilename()` in order to append a file extension onto the name of the provided file if the user didn't happen to provide one (although some operating systems might ignore this argument):

```
from tkinter import *
from tkinter import filedialog

window = Tk()

frame = Frame()
frame.pack()

def save_file(): # ask user to choose a file and update label
    type_list = [("Python scripts", "*.py"), ("Text files", "*.txt")]
    file_name = filedialog.asksaveasfilename(filetypes=type_list,
defaultextension=".py")
    my_text = "I will save: " + file_name
    label_file.config(text=my_text)

# blank label to hold name of chosen file
label_file = Label(frame)
label_file.pack()

# button to click on for 'Open' file dialog
button_open = Button(frame, text="Choose a file...", command=save_file)
button_open.pack()

window.mainloop() # start the application
```



I will save: C:/Python27/myScript.py

Above, I navigated to the folder "C:/Python27/" and told the application to save the file *myScript* without typing out any file extension, but because we provided `defaultextension=".py"` as an argument to `asksaveasfilename()`, this ending was automatically added to the name of the file.

As mentioned at the start of this section, Tkinter is a popular choice for GUI application development, and this introduction has only covered the most basic use cases. There are a number of other choices available, some of which offer additional flexibility and more complicated functionality - although usually at the cost of the extra time it takes to develop such detailed and complex applications. Once you feel that you need something more customizable than Tkinter, wxPython is a good choice for a next step. PyQt and PyGtk are two of the other most popular and widely used GUI toolkits for Python.

Review exercises:

1. Recreate the various windows pictured and described in this section, complete with all the same GUI widgets and any necessary interaction, by writing your own scripts without referring to the provided code
2. Using `grid()` to organize your widgets horizontally, create a button that, when clicked, takes on the value entered into an entry box to its right

Final Thoughts

Congratulations! You've made it to the beginning. You already know enough to do a lot of amazing things with Python, but now the real fun starts: it's time to explore on your own!

The best way to learn is by solving real problems of your own. Sure, your code might not be very pretty or efficient when you're just starting out, but it will be useful. If you don't think you have any problems of the variety that Python could solve, pick a popular module that interests you and create your own project around it.

Part of what makes Python so great is the community. Know someone learning Python? Help them out! The only way to know you've really mastered a concept is when you can explain it to someone else.

If you're interested in web development, consider diving into the more advanced Real Python courses, *Web Development with Python* and *Advanced Web Development with Django*.

When you feel ready, consider helping out with an open-source project on GitHub. If puzzles are more your style, try working through some of the mathematical challenges on Project Euler or the series of riddles at Python Challenge. You can also sign up for Udacity's free CS101 course to learn how to build a basic search engine using Python - although you know most of the Python concepts covered there already!

If you get stuck somewhere along the way, I guarantee that someone else has encountered (and potentially solved) the exact same problem before; search around for answers, particularly at Stack Overflow, or find a community of Pythonistas willing to help you out.

If all else fails, `import this` and take a moment to meditate on that which is Python.

Appendix A: Installing Python

Let's get Python 3.6 installed!

Check Current Version

Mac and Linux

All Mac OS X versions since 10.4 and most Linux distributions come pre-installed with the latest version of Python 2.7.x. You can view the version available by opening the terminal and typing `python` to enter the Python interpreter. If you have already installed multiple versions of Python, you may need to type `python3` instead of `python` at the terminal. The output will look something like this:

```
$ python
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

> **NOTE**: If you have a version older than 3.6, please download the latest version below.

Install Python

Choose your Operating system.

Mac

You need a Python version 3.6+. So, if you need to download a new version, download the latest installer for version 3.6.0.

Once downloaded, double-click the file to install.

Linux

If you are using Ubuntu, Linux Mint, or another Debian-based system, enter the following command in your terminal to install Python:

```
$ sudo apt-get install python3.6
```

Or you can download the tarball directly from the official Python website. Once downloaded, run the following commands:

```
$ tar -zxvf [mytarball.tar.gz]
$ ./configure
$ make
$ sudo make install
```

> **NOTE**: If you have problems or have a different Linux distribution, you can always use your package manager or just do a Google search for how to install Python on your particular Linux distribution.

Windows

Download

Start by downloading Python 3.6.0 from the official Python website. The Windows version is distributed as a MSI package. Once downloaded, double-click to start the installer. Follow the installer instructions to completion. By default this will install Python to `C:\Python36` .

> **NOTE**: You may need to be logged in as the administrator to run the install.

Test

To test this install open your command prompt, which should open to the C:prompt, `C:/>`, then type:

```
\Python36\python.exe
```

And press enter. You should see something like:

```
Python 3.6.0 (v3.6.0:...) on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Yay! You just started Python!

> **NOTE**: The `>>>` indicates that you are at the Python interpreter (or prompt) where you can run Python code interactively.

To exit the Python prompt, type:

```
exit()
```

Then press Enter. This will take you back to the C: prompt.

Path

You also need to add Python to your PATH environmental variables, so when you want to run a Python script, you do not have to type the full path each and every time, as this is quite tedious. In other words, after adding Python to the PATH, we will be able to simply type `python` in the command prompt rather than `\Python36\python.exe`.

Since you downloaded Python version 3.6.0, you need to add the add the following directories to your PATH:

- `C:\Python36\`
- `C:\Python36\Scripts\`
- `C:\PYTHON36\DLLs\`
- `C:\PYTHON36\LIB\`

Open your power shell and run the following statement:

```
[Environment]::SetEnvironmentVariable("Path",
    "$env:Path;C:\Python36\;C:\Python36\Scripts\;
    C:\PYTHON36\DLLs\;C:\PYTHON36\LIB\;", "User")
```

That's it.

Video

Watch the video here for assistance. Note: Even though this is an older version of Python the steps are the same.

Verify Install

Test this new install by opening a *new* terminal, then type `python`. You should see the same output as before except the version number should now be 3.6.0 (or whatever the latest version of Python is):

```
$ python
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

> **NOTE**: You may need to run `python3` instead of `python` if you have multiple versions of Python installed.

Congrats! You have Python installed and configured.

Appendix B: Regular Expressions

A regular expression (also known as regex, regexp, or RE) is a powerful tool used for quickly finding and extracting patterns in text. Regular expressions are not particular to Python; they are a general programming concept that can be used with a wide variety of programming languages. They use a language all of their own that is notoriously difficult to learn but incredibly useful. Once mastered, you can find complex patterns within text in very compact syntax.

Basic Syntax

The basic idea behind regular expressions is to combine ordinary text with a number of different "special" characters that are interpreted as ways to signify different types of patterns. For instance, the asterisk character, `*`, stands for "zero or more" of whatever came just before the asterisk. Let's look at some basic examples of using such patterns to define regular expressions (left column) and then the subsequent strings that those expressions match (right column):

Expression	Description	Example Match(es)
a	matches the character "a"	a
happy	matches the entire word "happy"	happy
h.ppy	"." matches any single character	happy, hippy
ha*ppy	"*" matches zero or more of "a"	hppy, happy, haaappy
us|them	"|" matches "us" or "them" using OR	us, them

> **SEE ALSO**: Check out the official Python docs for all the special characters or PyRegex for a compact version.

The `*` special character works similarly to using a wildcard in file searching. The `|` special character works the same way as saying `or` in Python. You can also use parentheses to give precedence to certain patterns. What do you think `the(ir|re|y're)` will match? Answer

When Should You Use Regular Expressions?

Rule of thumb: Unless you *absolutely* need to use regular expressions, stick with Python's basic string-matching methods - like `find()` and `replace()`. They are *much* easier to understand. In other words, take a lazy approach and start with built-in functions; if the code starts becoming overly complex, switch to regular expressions. But what does "overly complex" mean?

Well, let's start at the opposite extreme: simple. When you can implement all of your search logic in a single predicate, you should use the basic Python functions.

For example:

```
>>> str = 'Explicit is always better than implicit.'
>>> print(str.replace('always', 'much'))
Explicit is much better than implicit.
```

In the above case we know exactly what we're looking for. But what happens if we don't? What about a 10-digit phone number? In that case, we only know that the pattern is a series of integers with 10 digits (that may or may not be broken up by dashes or some other characters).

```
>>> str = "Jack's phone number is 415-690-4993"
>>> print(str.replace('-', ''))
Jack's phone number is 4156904993
```

Here we can still use `replace()` since we are working with just a single phone number. Let's say you have a `list` of phone numbers that need to be standardized into the format `xxx-xxx-xxxx`:

```
phone_list = [
    "555-555-5555","555 555 5555","555.555.5555",
    "555.555.5555","555.555-5555","555/555/5555"
]
```

It's safe to say that if you used `find()` and/or `replace()`, you'd have a number of conditional statements and a loop, making the solution significantly more complex, spanning beyond a single predicate.

However, if we use regex, the solution is much simpler:

```
>>> import re
>>>
>>> phone_list = [
...     "555-555-5555","555 555 5555","555.555.5555",
...     "555.555.5555","555.555-5555","555/555/5555"
... ]
>>> pattern = r'\D'
>>> for phone in phone_list:
...     phone_num = re.sub(pattern, "-", phone)
...     print "Phone Num: ", phone_num
...
Phone Num:  555-555-5555
Phone Num:  555-555-5555
Phone Num:  555-555-5555
Phone Num:  555-555-5555
Phone Num:  555-555-5555
Phone Num:  555-555-5555
>>>
```

Here, we used the `re.sub()` function to match all non-digits (`r'\D'`) - e.g., periods, forward-slashes, and spaces - and then replace (or substitute) them with `"-"`. This is a commonly used function, so make sure you understand it.

Although these are still just basic examples, you should understand what a regular expression is and how to use them. Used correctly, in the right situation, you can save much time when having to find and parse a string.

Before moving on, make sure you understand the examples, as well the the meaning of the special characters along with how they are used in your regular expressions from this lesson. Especially take note of when you should and shouldn't use regular expressions, and remember that regular expressions can become complex as well, especially when used incorrectly. If you find yourself writing complex expressions, you may need to test the waters with `find()` and `replace()`.

Another good rule of thumb for deciding when to use regular expressions rather than `find()` and `replace()` would be when regex:

1. is easier to understand;
2. expresses a clear, concise intent; and

3. is much shorter add easier to change/adapt.

> **SEE ALSO**: If you still have questions over whether to use regex or the regular string methods, follow the *Zen of Python*: `import this`.

Functions

`re.match()` and `re.search()`

Used for *finding* text, the `re.match()` and `re.search()` functions are two of the most widely used methods in the `re` module.

1. `re.match(pattern, string)` searches for a match only at the beginning of a string.
2. `re.search(pattern, string)` searches for a match anywhere in the string.

Before looking at an example, we'll be using `group()` to further isolate portions of the matching text. You specify groups with parenthesis within the expression.

Example 1

```
>>> import re
>>> text = "easier said than done"
>>>
>>> # re.match
>>>
>>> find_match = re.match(r'done', text)
>>> if find_match:
...     print("Found: {}".format((find_match).group()))
... else:
...     print("Not found.")
...
Not found
>>>
>>> #re.search
>>>
>>> find_match = re.search(r'done', text)
>>> if find_match:
...     print("Found: {}".format((find_match).group()))
... else:
...     print("Not found.")
...
Found: done
```

> **NOTE**: You could also use the expression `r\ds[a-z]*` to grab the word "done" (and some other possibilities) using the `search()` function. Try this out. Can you find other special characters which can be used to match the word "done"?

Example 2

```
import re

text = "My name is Inigo"

# re.match(pattern, string, flags=0)
m = re.match(r'(.*) name (.s) .*', text)

if m:
   print("group(0):", m.group(0))
   print("group(1):", m.group(1))
   print("group(2):", m.group(2))
else:
   print("Sorry. No match!!")
```

Results:

```
group(): My name is Inigo
group(1): My
group(2): is
```

What's happening here?

First, break down the regular expression:

1. `.*` - Matches zero or more of any character
2. `name` - Matches the entire word "name"
3. `.s` - Matches any single character and then the character "s".

As for the groups:

Group 0 defines the string matched from the entire regular expression, while Group 1 and 2 represent sub-groups (defined by anything put into parentheses).

Try changing the groups - i.e., `r'(.*) name .s (.*)'` to see how it affects the output.

Example 3

You can also define group names, which just makes the groups easier to read.

```
import re

string = "Inigo Montoya"

# re.match(pattern, string, flags=0)
m = re.match(r"(?P<first>\w+)\W+(?P<last>\w+)", string)

if m:
   print("group(0):", m.group(0))
   print("group(1):", m.group(1))
   print("group(2):", m.group(2))
   print("")
   print('group("first") : ', m.group("first"))
   print('group("last") : ', m.group("last"))
else:
   print("Sorry. No match!!")
```

Break down the regex on your own. What does each group match? What does each special character do? How about the group definitions? Do they make sense to you?

Like the previous example, group 0 defines the string matched from the entire regular expression, while Group 1 and 2 still represent the sub-groups. Since we also added in named groups using `(?P<name>...)` , we can also represent the substrings with "first" and "last".

Practice

Would you use `re.match()` or `re.search()` to get the word "said" from `text = "easier said than done"` ? Not sure? Test it out both ways, using regex and avoiding it. How about getting an unknown number out of a string of text?

`re.sub()`

Again, `re.sub()` is used for searching and replacing text: `re.sub(pattern, replacement, string, max=0)` . This function replaces all occurrences of a specific pattern unless a positive `max` is provided.

Example

```
>>> import re
>>>
>>> text = "Real Python teaches programming and web development through hands-on,
interesting examples."
>>> pattern = re.sub("hands-on", "practical", text, 0)  # no max; replace all
>>> print(pattern)
Real Python teaches programming and web development through practical, interesting
 examples.
```

Practice

Type the following into your shell:

```
>>> import re
>>> text = "Avoid having single statements on a single line"
```

Add a variable called `pattern` that uses the `re.sub()` function to change the output to: "Avoid having multiple statements on a single line". Stuck? Use those information retrieval skills. Find the answer on your own with a little help from Google, Stack Overflow, the Python documentation, and so forth.

More Practice

Problem 1

```
import re

validation = re.compile(r'[A-Za-zs.]')

name = raw_input("Please enter your name: ")

while not validation.search(name):
    print "Please enter your name correctly!"
    name = raw_input("Please enter your name: ")

print("\nYour name is {}!".format(name))
```

1. Read about the `re.compile` function from the official Python documentation. In your own words, describe how to use it and why you'd want to.
2. Next, run the program a few times, testing inputs that validate and don't validate.
3. Finally, refactor the program to ensure that the input is an email address: test for the existence of alphanumeric text, followed by the "@" symbol, another string, a period, and finally the ending "com".
4. Bonus: Allow for other domains - i.e., com, org, edu or net.

Problem 2

Open the *phone_list.py* file from the exercises folder, which contains a list of dictionaries:

```
data = [
    {'name': 'Debra Hardy', 'phone': '(140) 732-2619'},
    {'name': 'Claudia Baker', 'phone': '(833) 362-0448'},
    {'name': 'Justin Lara', 'phone': '(609) 832-1565'},
    {'name': 'Judah Battle', 'phone': '(199) 834-7433'},
    {'name': 'Florence Nielsen', 'phone': '(769) 666-4576'},
    {'name': 'Orlando Kirby', 'phone': '(618) 110-3675'},
    {'name': 'Tucker Webb', 'phone': '(990) 295-9494'},
    {'name': 'Abel Jacobs', 'phone': '(840) 537-3516'},
    {'name': 'Ann Crane', 'phone': '(345) 876-2223'},
    ...
]
```

Write a Python script called *phone-book-fun.py* to find all the people with the last name "Hardy" or a first name starting with the letter J. Output their first names, last names, and phone numbers.

Assignment: Data cleaning with regular expressions

Your assignment is simple: with Sublime Text (or your favorite text editor that supports regular expression), use the **Find** and **Replace** tools along with regular expressions to turn _*sloppy_data.tsv* into _*clean_data.tsv*.

> **NOTE** The answers will be displayed in Sublime Text. If you are moving on to the next courses, you will need to use Sublime Text or something similar, so it's a good idea to download it now. There are other text editors that support regular expressions such as TextWrangler for Mac and Notepad++ for Windows.

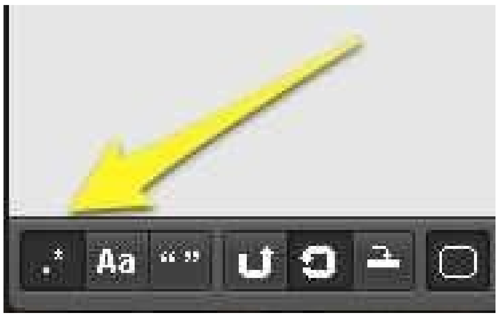
sloppy_data.tsv is essentially just a tab-separated value (TSV) file that includes unnecessary blank lines and spaces. Right now, it's difficult to open correctly in older versions of Microsoft Excel. Although there are only twenty rows, which wouldn't take too long to correct by hand, pretend this has 20,000 rows. That would not be fun to manually fix; instead, you'd want to automate it - which is exactly why we're going to use regular expressions. Make the following changes:

Remove:

1. Blank lines
2. Spaces from the beginning of each line
3. Bracketed numbers (e.g., `[1]`)
4. The first column of numbers
5. `Inc.` , `, Inc.` , `Inc` , and `Incorporated`
6. Space between the area code, `(XXX)` , and the phone number, `XXX-XXXX`

Before jumping into the assignment, let's take a look at a quick example. Open your text editor and add "N.Y.C" to a blank file. Now, using regular expressions, replace "N.Y.C" with "NYC".

To do this in Sublime text, click **Find** then **Replace**. Enter the regular expression `\.` into the **Find** field. Then leave the **Replace** field blank. This will replace periods with nothing, which is what we want. Before you click replace, activate regex search:



Now simply click replace all. You should now see "NYC". Do you know why we had to use `\.` instead of just `.`? A `.` is already a special character in regex, so we need to escape it using a backslash. Adding the backslash character right before the period made regex interpret it as an *actual* period instead of a special character itself.

Okay, have a go at the assignment. If you need help with the regular expression, check out the *regex_answers.txt* file.

Assignment: Reviewing regular expressions

1. Copy the code below and save it as *regex_review.py*.
2. Run the file. All `print` statements return `False`.
3. Modify the variables so that all of the `print` statements return `True`.

```
zero = "Real Ruby"
one = "5/25/14"
two = "A99 9AA"
three = r''
four = "6.76"
five = ["happy","birthday"]
six = r'\.(doc)$'
seven = "My email is michael@mherman.org"

# DO NOT CHANGE ANYTHING BELOW THIS LINE #
# -------------------------------------- #

print("zero:  {}".format(zero == re.search(r'[P].*', "This is Real Python").group(
)))
print("one:   {}".format(one == re.search(r'\d{1,2}\/\d{1,2}\/\d{4}', "5/25/2014")
.)group())
print("two:   {}".format(two == re.match()
    r'[A-Z]([A-Z](\d{1,2}|\d[A-Z])|\d{1,2})\s\d[A-Z]{2}',
    "A88 8AA",
    re.VERBOSE
).group())
print("three: {}".format(bool(re.search(three, "B4c r79").group())))
print("four:  {}".format(bool(re.search(r'\$[0-5]\.\d\d', four))))
print("five:  {}".format(bool(re.search(r'\ha{4,10}ppy\b', five[0]))))
files = ['test.doc', 'test.odt', 'test.ddt', 'doc', 'testodt', 'test.doc']
matched_files = [file for file in files if re.search(six, file)]
print("six:   {}".format(len(matched_files) == 3))
email_regex = r'\w+@\w+\.(com|org|edu|net)'
text = "My email is michael@mherman.org"
redacted_text = re.sub(email_regex, '(email redacted)', text)
print("seven: {}".format(seven == redacted_text))
```

For more examples and documentation, check out Python Module of the Week's post on regular expressions.

Appendix C: Primer on Object-Oriented Programming

Up until now, we have structured our programs around functions utilizing an approach to programming called procedural programming. This approach is like a recipe in that it provides a set of steps, in the form of functions and code blocks, which flow sequentially in order to complete a task.

The other major paradigm is objected-oriented programming which provides a means of structuring programs so that properties and behaviors are bundled into individual objects. For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending. Put another way, object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc.

The key takeaway is that objects are at the center of the object-oriented programming paradigm, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

> **NOTE**: Since Python is a multi-paradigm programming language, you can choose the paradigm that best suits the problem at hand, mix different paradigms in one program, and/or switch from one paradigm to another as your program evolves.

Classes

Focusing first on the data, each thing or object is an instance of some *class*.

The primitive data structures like numbers, strings, and lists are designed to represent simple things like the cost of something, the name of a poem, and your favorite colors, respectively. They've worked well up until this point.

What if you wanted to represent something much more complicated? For example, let's say you wanted to track a number of different animals. If you used a list, the first element could be the animal's name while the second element could represent its age. How would you know which element is supposed to be which? What if you had 100 different animals? Are you certain each animal has both a name and an age, and so forth? What if you wanted to add other properties to these animals? This lacks organization, and it's the exact need for classes.

Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an `Animal()` class to track properties about the Animal like the name and age.

It's important to note that a class just provides structure - it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. The `Animal()` class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal's name or age is.

It may help to think of a class as an *idea* for how something should be defined.

Instances

While the class is the blueprint, an instance is a copy of the class with *actual* values, literally an object belonging to a specific class. It's not an idea anymore; it's an actual animal, like a dog named Roger who's eight years old.

Put another way, a class is like a form or questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it contains actual information relevant to you. You can fill out multiple copies to create many different instances, but without the form as a guide, you would be lost, not knowing what information is required. Thus, before you can create individual instances of an object, we must first specify what is needed by defining a class.

Define a Class

Defining a class is simple:

```
class Dog(object):
    pass
```

You start with the `class` keyword to indicate that you are creating a class, then you add the name of the class (using CamelCase notation, starting with a capital letter), and finally add the class that you are inheriting from in parentheses (more on this below). As well, we used the key word `pass`. This is very often used as a place holder where code will eventually go. It allows us to run this code without throwing an error.

Instance Attributes

All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the `__init__()` method to initialize (e.g., specify) an object's initial attributes by giving them their default value (or state). This method must have at least one argument as well as the `self` variable, which refers to the object itself (e.g., Dog).

```
class Dog(object):

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

In the case of our `Dog()` class, each dog has a specific name and age, which is obviously important to know for when you start actually creating different dogs. Remember: the class is just for defining the Dog, not actually creating *instances* of individual dogs with specific names and ages; we'll get to that shortly.

Similarly, the `self` variable is also an instance of the class. Since instances of a class have varying values we could state `Dog.name = name` rather than `self.name = name`. But since not all dogs share the same name, we need to be able to assign different values to different instances. Hence the need for the special `self` variable, which will help to keep track of individual instances of each class.

> **NOTE**: You will never have to call the `__init__()` method; it gets called automatically when you create a new 'Dog' instance.

## Class Attributes

While instance attributes are specific to each object, class attributes are the same for all instances - which in this case is *all* dogs.

```
class Dog(object):

    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

So while each dog has a unique name and age, every dog will be a mammal.

Let's create some dogs...

Instantiating

Instantiating is a fancy term for creating a new, unique instance of a class.

For example:

```
>>> class Dog(object):
...     pass
...
>>> Dog()
<__main__.Dog object at 0x1004ccc50>
>>> Dog()
<__main__.Dog object at 0x1004ccc90>
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
>>>
```

We started by defining a new `Dog()` class, then created two new dogs, each assigned to different objects. So, to create an instance of a class, you use the the class name, followed by parentheses. Then to demonstrate that each instance is actually different, we instantiated two more dogs, assigning each to a variable, then tested if those variables are equal.

What do you think the type of a class instance is?

```
>>> class Dog(object):
...     pass
...
>>> a = Dog()
>>> type(a)
<type 'instance'>
>>>
```

Let's look at a slightly more complex example...

```
class Dog(object):

    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age


# Instantiate the Dog object
philo = Dog("Philo", 5)
mikey = Dog("Mikey", 6)

# Access the instance attributes
print("{} is {} and {} is {}.".format(
    philo.name, philo.age, mikey.name, mikey.age))

# Is Philo a mammal?
if philo.species == "mammal":
    print("{0} is a {1}!".format(philo.name, philo.species))
```

> **NOTE**: Notice how we use dot notation to access attributes from each object.

Save this as *dog_class.py*, then run the program. You should see:

```
Philo is 5 and Mikey is 6.
Philo is a mammal!
```

What's going on?

We created a new instance of the `Dog()` class and assigned it to the variable `philo`. We then passed it two arguments, `"Philo"` and `5`, which represent that dog's name and age, respectively. These attributes are passed to the `__init__` method, which gets called any time you create a new instance, attaching the name and age to the object. You might be wondering why we didn't have to pass in the `self` argument. This is Python magic; when you create a new instance of the class, Python automatically determines what `self` is (a Dog in this case) and passes it to the `__init__` method.

Review exercises:

1. Using the same `Dog()` class, instantiate three new dogs, each with a different age. Then write a function called, `get_biggest_number()`, that takes any number of ages (`*args`) and returns the oldest one. Then output the age of the oldest dog like so:

   ```
   The oldest dog is 7 years old.
   ```

   Save this file as *oldest_dog.py*.

Instance Methods

Instance methods are defined inside a class and are used to get the contents of an instance. They can also be used to perform operations with the attributes of our objects. Like the `__init__` method, the first argument is always `self`:

```
class Dog(object):

    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)

# Instantiate the Dog object
mikey = Dog("Mikey", 6)

# call our instance methods
print(mikey.description())
print(mikey.speak("Gruff Gruff"))
```

Save this as *dog_instance_methods.py*, then run it:

```
Mikey is 6 years old
Mikey says Gruff Gruff
```

In the latter method, `speak()`, we are defining behavior. What other behaviors could you assign to a dog? Look back to the beginning paragraph to see some example behaviors for other objects.

Modifying Attributes

You can change the value of attributes based on some behavior:

```
>>> class Email(object):
...     is_sent = False
...     def send_email(self):
...         self.is_sent = True
...
>>> my_email = Email()
>>> my_email.is_sent
False
>>> my_email.send_email()
>>> my_email.is_sent
True
>>>
```

Here, we added a method to send an email, which updates the `is_sent` variable to `True`.

Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes. It's important to note that child classes override *or* extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an `object`, which generally all other classes inherit as their parent.

Dog Park Example

Let's pretend that we're at a dog park. There are multiple Dog objects engaging in Dog behaviors, each with different attributes. In regular-speak that means some dogs are running, while some are stretching and some are just watching other dogs. Furthermore, each dog has been named by its owner and, since each dog is living and breathing, each ages.

What's another way to differentiate one dog from another? How about the dog's breed:

```
>>> class Dog(object):
...     def __init__(self, breed):
...         self.breed = breed
...
>>> spencer = Dog("German Shepard")
>>> spencer.breed
'German Shepard'
>>> sara = Dog("Boston Terrier")
>>> sara.breed
'Boston Terrier'
>>>
```

Each breed of dog has slightly different behaviors. To take these into account, let's create separate classes for each breed. These are child classes of the parent Dog class.

Child classes: Extending the functionality of the parent class

Create a new file called *dog_inheritance.py*:

```
# Parent class
class Dog(object):

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# child classes inherit attributes and
# behaviors from the parent class
jim = Bulldog("Jim", 12)
print jim.description()

# child classes have specific attributes
# and behaviors as well
print jim.run("slowly")
```

Read the comments aloud as you work through this program to help you understand what's happening, then before you run the program, see if you can predict the expected output.

You should see:

```
Jim is 12 years old
Jim runs slowly
```

We haven't added any special attributes or methods to differentiate a `RussellTerrier` from a `Bulldog`, but since they're now two different classes, we could for instance give them different class attributes defining their respective speeds.

Parent vs. Child Class

The `isinstance()` function is used to determine if an instance is also an instance of a certain parent class.

Save this as *dog_isinstance.py*:

```
# Parent class
class Dog(object):

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# child class (inherits from Dog() class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# child class (inherits from Dog() class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# child classes inherit attributes and
# behaviors from the parent class
jim = Bulldog("Jim", 12)
print(jim.description())

# child classes have specific attributes
# and behaviors as well
print(jim.run("slowly"))

# is jim an instance of Dog()?
print(isinstance(jim, Dog))

# is julie an instance of Dog()?
julie = Dog("Julie", 100)
print(isinstance(julie, Dog))

# is johnny walker an instance of Bulldog()
johnnywalker = RussellTerrier("Johnny Walker", 4)
print(isinstance(johnnywalker, Bulldog))

# is julie and instance of jim?
print(isinstance(julie, jim))
```

Output:

```
('Jim', 12)
Jim runs slowly
True
True
False
Traceback (most recent call last):
  File "dog_isinstance.py", line 50, in <module>
    print isinstance(julie, jim)
TypeError: isinstance() arg 2 must be a class, type, or tuple of classes and types
```

Make sense? Both `jim` and `julie` are instances of the `Dog()` class, while `johnnywalker` is not an instance of the `Bulldog()` class. Then as a sanity check, we tested if `julie` is an instance of `jim`, which is impossible since `jim` is an `instance` of a class rather than a class itself - hence the reason for the `TypeError`.

Child classes: Overriding the functionality of the parent class

Remember that child classes can also override attributes and behaviors from the parent class. For examples:

```
>>> class Dog(object):
...     species = 'mammal'
...
>>> class SomeBreed(Dog):
...     pass
...
>>> class SomeOtherBreed(Dog):
...     species = 'reptile'
...
>>> frank = SomeBreed()
>>> frank.species
'mammal'
>>> beans = SomeOtherBreed()
>>> beans.species
'reptile'
>>>
```

The `SomeBreed()` class inherits the `species` from the parent class, while the `SomeOtherBreed()` class overrides the `species`, setting it to `reptile`.

Review exercises:

1. Create a `Pet()` class that holds instances of dogs; this class is completely separate from the `Dog()` class. In other words, the `Dog()` class does not inherit from the `Pet()` class. Then assign three dog instances to the `Pet()` class. Start with the following code below. Save the file as *pet_class.py*. Your output should look like this:

   ```
    I have 3 dogs. Tom is 6. Mike is 7. Larry is 9. And they're all mammals, of c
   ourse.
   ```

   Starter code:

```
# Parent class
class Dog(object):

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)

# child class (inherits from Dog() class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)

# child class (inherits from Dog() class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

2. Using the same file, add a class attribute of `is_hungry = True` to the `Dog()` class. Then add a method called `eat()` which changes the value of `is_hungry' to` False` when called. Figure out the best way to feed each dog and then output "My dogs are hungry." if all are hungry or "My dogs are not hungry." if all are not hungry. The final output should look like this:

```
I have 3 dogs. Tom is 6. Mike is 7. Larry is 9. And they're all mammals, of c
ourse. My dogs are not hungry.
```

3. Next, add a `walk()` method to both the `Pet()` and `Dog()` classes so that when you call the method on the `Pet()` class, each dog instance assigned to the `Pet()` class will `walk()`. Save this as *dog_walking.py*. This is slightly more difficult. Start by implementing the method in the same manner as the `speak()` method. As for the method in the `Pet()` class, you will need to iterate through the list of dogs, then call the method itself. The output should look like this:

```
Tom is walking!
Mike is walking!
Larry is walking!
```

Assignment: Comprehension check

Answer the following questions:

1. What's a class?
2. What's an instance?
3. What's the relationship between a class and an instance?
4. What's the Python syntax used for defining a new class?
5. What's the spelling convention for a class name?
6. How do you instantiate, or create an instance of, a class?
7. How do you access the attributes and behaviors of a class instance?
8. What's a method?
9. What's the purpose of `self`?
10. What's the purpose of the `__init__` method?
11. Describe how inheritance helps prevent code duplication.
12. Can child classes override properties of their parents?

Assignment: Model a farm

In this assignment, you'll create a simplified model of a farm. As you work through this assignment, keep in mind that there are a number of correct answers. The focus of this assignment is less about the Python class syntax and more about software design in general, which is highly subjective. That said, if you still have syntax questions or want more practice, you should go through the Introduction to Classes track from Codecademy to further your understanding of the Python class syntax.

Before you write any code, grab a pen and paper and sketch out a model of your farm, identifying classes, attributes, and behaviors. Think about inheritance. How can you prevent code duplication? Take the time to work through as many iterations as you feel are necessary.

If you want, scan or take a photo of your model(s) and email them to us at info@realpython.com to get feedback on the design and structure of your program.

The actual requirements are open to interpretation, but try to adhere to these guidelines:

1. You should have at least four classes: The parent `Animal()` class, and then at least three child animal classes that inherit `Animal()`.
2. Each class should have a few attributes and at least one method, which details some behavior as appropriate for either all animals or a single animal - i.e., walking or running, talking, sleeping, etc.
3. Keep it simple. Utilize inheritance. Make sure you output details about the animals and their behaviors.

After you finish, compare your solution to ours. Find areas where you could improve. Then wait a few days and do the problem over again... and again... and again... Good luck!

Assignment: Github with class

Create a new class called `Github()`, which has two attributes:

1. `username` - the Github username
2. `base_url` - the base URL for the Github API (https://developer.github.com/v3/)

Make sure to import `urlopen` from the appropriate `urllib` library so that you can make the API call:

```
from urllib.request import Request, urlopen
```

Finally, add two instance methods:

1. `get_user_info()` - returns the Github user info in JSON from https://api.github.com/users/:USERNAME
2. `get_user_repos()` - returns the Github user repos in JSON from https://api.github.com/users/:USERNAME/repos

Make sure to create at least one instance and call the methods to get the user info and repos.

Conclusion

You should now know what classes are, why you would want or need to use them, and how to create both parent and child classes to better structure your programs. This is not easy subject matter, and to be honest, most of the programs that you will build in the next course will utilize procedural programming. The last course focuses much more on object-oriented programming, though. Fortunately, you have plenty of time to practice until that point.

Since the syntax isn't the most intuitive, you can get more practice using classes in our Codecademy exercise where we have you build different types of `Car()` classes with various attributes and methods. For further explanation of objected-oriented programming, please consult these resources:

1. Official Python documentation
2. Dive into Python
3. Learn Python the Hard Way
4. Dive Into Object-oriented Python

Acknowledgements

This book would not have been possible without the help and support of so many friends and colleagues.

For providing valuable advice and candid feedback, I would like to thank Brian, Peter, Anna, Doug, and especially Sofia, who by now has probably read this material more times than I have. Thanks as well to Josh for taking the time to share his valuable experience and insights.

A special thanks to the Python Software Foundation for allowing me to graffiti their logo.

Finally, my deepest thanks to all of my Kickstarter backers who took a chance on this project. I never expected to gather such a large group of helpful, encouraging people; I truly believe that my Kickstarter project webpage might be one of the nicest gatherings of people that the Internet has ever experienced.

I hope that all of you will continue to be active in the community, asking questions and sharing tips. Your feedback has already shaped this course and will continue to help me make improvements in future editions, so I look forward to hearing from all of you.

This book would never have existed without your generous support:

Benjamin Bangsberg, JT, Romer Magsino, Daniel J Hall, John Mattaliano, Jordan "DJ Rebirth" Jacobs, Al Grimsley, Ralf Huelsmann, Germany, Amanda Pingel Ramsay, Edser, Andrew "Steve" Abrams, Diego Somarribas B., John McGrath, Zaw Mai Tangbau, Florian Petrikovics, Victor Pera (Zadar, Croatia), xcmbuck@yahoo.com, Daniel R. Lucas, Matthew C, Duda, Kenneth, Helena, Jason Kaplan, Barry Jones, Steven Kolln, Marek Rewers, Andrey Zhukov, Dave Schlicher, Sue Anne Teo, Chris Forrence, Toby Gallo, Jakob Campbell, Christian "DisOrd3r" Johansson, Steve Walsh, Joost Romanus, Jozsef Tschosie Kovacs, Back Kuo, James Anselm, Christian Gerbrandt, Mike StoopsMichael A Lundsveen, David R. Bissonnette, Jr., Geoff Mason, Joao da Silveira, Jason Ian Smith, Anders Kring, Ruddi Oliver Bodholdt Dal, edgley, Richard Japenga, Jake, Ken Harney, Brandon Hall, B. Chao, Chinmay Bajikar, Clint LeClair, Davin Reid-Montanaro, Isaac Yung, Espen Torseth, Thomas Hogan, Nick Poenn, Eric Vogel, Jack Salisbury, James Rank, Jamie Pierson, Christine Paluch, Peter Laws, Ken Hurst, Patrick "Papent" Tennant, Anshu Prabhat, Kevin Wilkinson, Joshua Hunsberger, Nicholas Johnson, Max Woerner Chase, Justin Hanssen, pete.vargasmas@gmail.com, James Edward Johnson, Griffin Jones, Bob Byroad, Hagen Dias, Jerin Mathew, Jasper

Blacketer, Jonathan Lundstrom, Django O Gato, Mathias Ehret, Interphase Technologies, Stanton Hooley, Jerome P., Michael Edwards, Robert Bownds, Patrick Corbett, Akshay Rao, Hendricks Weatherborne, Werner B, Paul May, Ismail Hassan, Akash Gordhan, Brian Joseph Bixon, Peter Gorys, Cydroiid, Nicolas Hauviller, Jack Carroll, Dave Kin, Michael Anaya, Dave Kilsheimer, Jay, Jake Stambaugh, Dustin CS Wagner, Faridi Qaium, Michael Ramos, JimT, Sven Borer, Locutus, James "Baker" Stuart, Chris Browne, Jose Agraz, Kasper Rugaard, Joseph Klaszky, Brandon Cotton, D, Lucas Derraugh, S Patterson, Ian Miles, Chuck McLellan, JBo796, D.J.Murray, Tom Randle, Anna, Navtej Saggu, Dylan Brabec, J Caldwell, Joseph Tjon, Andy Bates, Charles Meyer, Sebastian Sahlin, Michael Phillips - Waves & Tidings, Toh Chee Seng, Garry "Gorio" Padua, Eric Wallmander, Shazbot, Diante Edwards, AlmostHuman, Andy E, Matt Stultz, Eric Koerner, Gareth Shippen, Kendal Edwards Robert Sholl, ROdMX, Kyle Hinkle, Timmy Crawford, Ron W, Trevor Miller, Drew Kamthong, Yatin Mehta, smashbro35, Kelsey Kaszas, Mike, Duppy, Mikk Kirstein, Adib Khoury, Lou Ward, Milan Janošik, Benjamin Schlageter, Koen Van den Eeckhout, Richard Wojtaszak, Mathias Brenner, Magnus Nygren, Harrison Tan, Amin Azam, Chris Awesome, Andy Welsh, Kyle Hughes, J. Owen, Nick Manganelli, Ivan Zamotokhin, Liam Jensz, Matthew Korytowski, David L Sprague, Philipp Beckmann, Lorenzo R V, Miles Wilkins, Andrew J. Dimond, Steve Harris, Will Knott, Odd Magnus Rosbach, Juan Hernaiz, Tomoyuki Sugihara, Calum Mackenzie, JB, Nick G., Thomas Perrella, Ian McGuire, Neelfyn, James Rodgers, Mark Dixon, Chezzas, Jessica Ledbetter, Ariel Fogel, Davide Varni, Sasha Brogan, Clint Myers, Rob Trimarco, Samuel Judd, trademarques.co, Stefan, Jim O., Simon Hedegaard, Ryan Fagan, Ed Woods, Iacton Qruze, Mentat, Thomas Clarke - NitroEvil, Saym Basheer, Kevin Spleid, Evaristo Ramos, Jr., Grant Lindsay, Erin I, twistcraft, Carter Thayer, Mark Shocklee, Gerard Cochrane Jr, Michael J Carl, Anthony Allan, Howard Cheetham, Ben Piper, Richard Herwig, Tyler Piderit, JimminiKin, Robert Ng, Penguins Are From Space, James L Hays, Kyle K., Nicholas J. Atkins, R. Travis Brown, T J Woo, Alexander Arias, C.S.Putnam, Peter Webb, Dmitriy Litvak, Ciaran Regan, Warai Otoko, John Langhoff, andrew brennan, Andrew Grosner, jamorajj, Adiel Arvizu, Mark Norgren, Eric R. Stoeckel, Jr., Pedro Gomez, Chris Conner, @andersosthus, Chandra Nagaraja, Jan P. Monsch, Corra, Trentyz, Nicholas Hehr, manmoe, Lam Chuan Chang (Singapore), Roy G. Biv, David Szymczak, Ronald Singletary, Simone F., Joyce B (Australia), simon grensted, sebastien dominati, Stephen T. Truong, Michele Gargiulo, Jonathan @Ardua Doyle, Tommy A., Samuel Morgan, Nir Friedman, Pasha Kagan, Gregory Kurts, B.Wold, Brad F., RobbieP, Fernando Varela, Michael Lebowitz, Michael T Olivier, A. Backer, Fredrik Karlsson, Halmstad, Sweden, Dave Lewis, Andreas Matern, Jon Strom, Max Dayan, Joh.Lummel - Germany, Jigar Patel, Sean R, Elly & Andres, Hong Yi, D Carter, Thomas M. Johnson, Matthias Keller, Eirik Bilet, Daniel

Otero, Ted Davidson, Dougie "Fresh Fingers" Friedman, Amr Metwally, Dymatrix Solutions, Dylan Demoe, Turki N. Shahel, Cory S Spoerl, Ben B, Marc, Kari, Penelope, and Marylee Evans, Niklas "Cartina" Kallarsson, Paul W., B. Gunaratnam, Stephanie Delgado, coilercard, Dave TJ, Robert J. Romero, Matthew Peterson, Brant Winter, Darkdavy, Zax Tang, Anthony A Greenwood Jr., Zachary Howard, Hal, John A. Booth, Ranoka, Larry Younger, Georg Fischer (@snorpey), Thor G. Steinsli, Frode, Mike and Julia Nelson, code_more, Hunter Harris, Terrence DeBeatham, Ryan Wheeler, Jerrid D.C. Fuehrer, Remy VINCENT, Rob Weatherly, Simon A, Daniel Y, Andrzej Wiktor, Earthling: one of 6 billion + and counting, Apos Croquelune, Adam Kim, iTrain Malaysia, Madison May, William F Todoro, Mark Danbom, Carlos Anchia, Rick Chepeus, R. Klein, Sebastian Oort, Brooks Cutter, Dylan Hutchison, Kilian Riedl, Tom "Harry" Mudway, Al Billings, Andrii Sudyn, Andrew O'Gorman, ShaColby Jackson, Robert Ovington, Brandon Usher, Joshua LeFever, William Miller, T. Siesmayer, Niclas Berndtsson, Brian Arbogast & Brynn Arborico, Roberto Velez, Matt M., WDM, vee xiong, John Thayer, Baron Burkhart, Nicholas A. DeLateur, Ben Hamilton, Cole Mercer, Dougie Nix, Shaun Walker, Olof Bengtson, Marek Belski, Chris Cannon, Bob Putnam, Jeff McMorris, Timothy Phillips, Rodolfo F. Guzman, Joe Burgwin, Andreas Borgstrom, Philip Ratzsch, Kostas Sarafidis, R. Arteaga, fullzero, Petros Ring, Harold Garron, Thomas "Gears" Butler, Neil Broadley, JPao, Aviel Mak, Kjeld Jensen, I P Freely, Arturo Goicochea Hoefken, Leo Sutedja, Cameron Kellett, Werner Beytel, Muhammad Khan, Jason Trogdon, Dao Tran, Thomas Juberg, Andy Nguyen, Petr Tomicek, Erik Rex, Stephen Meriwether, Benjamin Klasmer, Derick Schmidt, Kyle Thomas, R.Nana, Arpan Bhowmik, Jacob A. Thias, Elliot, Isaiah St. Pierre, Josh Milo Drchuncks, Dr. Sam N Jaxsam, Matthew M. McKee, Kyle Simmons, Jason Nell, Darcy Townsend, Jesse Houghton, Evan D., Marcel Arnold, Thomas Siemens, C Hodgson, Adrien D., Bjørn Spongsveen, Jemma, Ed Matthews, Nik Mohd Imran - Malaysia, Jason Weber, JTR Haldr, Matthew Ringman, Yoshio Goto, Evan Gary Hecht, Eric L., Hannes Hauer, Robert Mais, Highland Dryside Rusnovs, Michael Torres, Mike Alchus, Jonathan L. Martin, Oliver Graf, David Anspaugh, Joe Griesmer, Garrett Dunham, Srujan Kotikela, Laurel Richards, Lovelesh, Sarah Guermond, Brian Canada, PhD, Assistant Professor of Computational Science, University of South Carolina Beaufort, Shao, Antti Kankaanpaa, Carl F. Corneil, Laird_Dave, Nyholm Family, Brandon Graham, M. A. Merrell, Kyle Frazier, PT_SD, Travis Corwith, Elliot Jobe, A R Collins, Ørjan Sømme, Jay B Robertson, Jim Matt, Christopher Holabird, Ronny Varghese, Claudia Foley, Andrew Simeon, D G, Jay V. Schindler, Douglas Butler AKA SherpaDoug, Jon Perez, Pieter Maes, Gabriel McCann, John Holbrook, Melissa Cloud, Inukai, Henning Fjellheim, Jason Tsang, Juliovn, Reagan Short, Carlos Clavero Aymerich, Vaughan Allan, James Dougherty, Miles Johnson, Shirwin, Thomas Taimre, Michael Urhausen Cody C. Foster, Christoph G.W. Gertzen, Mag, Matt Monach, Tabor, Ashwin.Manthena, Lance Tully, NoVuS ReNoVaTiO, Joshua Broussard, Laurence Livermore, Rob Flavell, Fabian und Holger Winklbauer, Adriano Lo conte, Decio Valeri, Stephen Ewell, Erik Zolan, Dharm Kapadia, Esteban Barrios, Mehul, Thomas Fauser, Nathan Pinck, Grant P, Gary, Jonathan Moisan, David Warshow, Erica Zahka, Frederik Laursen, Piotr Napiorkowski, Chris Webster, James Kobulyar, Cobalt Halley, Dewey Kang, Fall, Susan Engle, David Garber, Rebecka Wallebom, Pai Siwamutita, Joel Gerstein, Brant Bagnall, Mr. Backer 7, Cole Smith, Gary Androphy, Keith L Hudson, Anthony Jackman, Regis LUTTER, Charles Jurica, Jose Gleiser, Mike Henderson, Khalid M AlNuaim, Dan "CidonaBoy" Murphy, BrianF84, Gunnar Wehrhahn, Marc Herzog, Leon Duckham, Justin S., DC, Kit Santa Ana, Tom Pritchard, Hamilton Chapman, Taylor Daniels, Andrew Wood, Tomer Frid, Peter B. Goode, John Ford, Otto Ho, LCT, WinDragon68, Faber M., Douglas Davies, Jacob Mylet, Niels Gamsgaard Frederiksen, Mark Greene, Rob Davarnia, Alex, Zabrina W., William Ochetski Hellas, Jose I. Rey, Dustin T. Petersen, A Nemeth, Praveen Tipirneni, Derek Etheridge, J.W. Tam, Andrei George Selembo, Leo Evans, Sandu Bogi Nasse, Christopher J Ruth, Erin Thomas, Matt Pham, KMFS, Todd Fritz, Brandon R. Sargent, boo, Lord Sharvfish Thargon the Destructor, Kylie "Shadow" Stafford, Edd Almond, Stanley, Brandon Dawalt, Sebastian Baum, F. Iqbal, Mungo Hossenfeffer, Zubair Waqar, Matt Russell, Sam Lau, Jean-Pierre Buntinx, James Giannikos, Chris Kimball, Happy, Nathan Craike, arieals@live.com, Asad Ansari, J. Rugged, Stephanie Johnston, Shunji, Mohammad Ammar Khan, John-Eric Dufour, Brad Keen, Ricardo Torres de Acha, Denis Mittakarin, Jeffrey de Lange, Stewart C. Annis, Nicholas Goguen, Vipul Shekhawat, Daniel Hutchison, @lobstahcrushah, Bjoern Goretzki, Hans de Wolf, Ray Barron, Garrett Jordan, Benjamin Lebsanft, Alessandro A. Minali, carlopezzi, Patrick Yang, Kieran Fung, Niloc, David Duncan, Tom Naughton, Barry G.-A. Wilson, Dave Clarke, Shawn Xu, Kevin D. Dienst, Durk Diggler, Marcus Friesl, Krisztina J., V. Darre, Duane Sibilly, II, Marije Pierson, Anco van Voskuilen, Joey van den Berg, Gil Nahmani, Stephen Yip, Richard Heying, Patrick Thornton, Ali AlOhali, Eric LeBlanc, Clifton Neff, Steve "Bofferbrauer" Weidig, Jacob Hanshaw, daedahl, Lee Aber, Jan Heimbrodt, Aquib Fateh, Gary Patton Wolfe, Jim Long, Št?panka Šoustkova, Logan Ziegenfus, Paul Casagrande, Jason E Joyner, Yvonne Caprini, Grehg Hilston, Peter Wang, Ajay Menon, Jaya Godavarthy, Zack Loveless, Tim Cupp, VfishV, Ansel B, Morgan Hough, Mauricio R. Cruz, Ryan Jolly, Daryl Walleck, Derek Mayer, Dopest, Jakesmiley, LuminoCityGroup, Jeff Ngo, Ronn Trevino, Adam Wilson, Ron Collins, Charles M Brumit, Charles Guerrero, Adan Juan Chavez Martinez, Zakir Hussain, Jawhny Cooke, R. George Komoto, Niko Chiu, Sean Murphy, M. Neukirch, Rob Kaplan, Boštjan Košir, modopo, Kenny "KeDeZ" Hernandez, Hector Gracia, Eric Finkenbiner, Achal Gautam, Stephane QUENARD, Wee Wing Kwong, Alan Carpilovsky, roland kohler (mondtagskind), Jim Brasic, Marjan Golob, Gareth Casey, Zainab Al-Ansari, Lee Smith, Christopher "Shibumi" Harwood, Sangeetha James, Sami alaaraji, Robert S Shaffer Jr, John Porter, Hua, Roxy Martinez, Dennis Hong, Fille-Loup, sodiume, Matt Nasi, Luis Rosado, Taylor Barnett, Dan Miller, Christopher Soviero, ArmyTra1n3d, The Pebble, Jacob Ledel, Michael Dax Iacovone, Don Rose, CL, Carl Bulger, David Ryan, Bill Horvath II, Charles Schrumpf, David J. Kay, C. K. Olson, Joe Dietrich, Maxim Kuschpel, Andrew K. Thomas, yelinaung, Fayerealian, Carl H. Blomqvist, Bob Sheff, Eric, Thom Maughan, Ronald Reyes, Kevin Kapka, Rami Taibah, Rahul Anantharaman, Renato Bonini Neto, Peter Young, Richard Drezdon, Hypermediaz, Mark Cater, Mark Thomas Griffin, Alain Brisard, Kevin Yuan, Sigmund Nylund, Jacob McClanahan, Julius Foitzik, Brian Garside, Gary Meola, Jr., Raymond Elward, Alex Pekarovsky, Scott Tolinski @stolinski, Jarle Eriksen, Havar Falck-Andersen, Akay Quinn, British, Jakov, Steve Edholm, Tom "Damage" Minnick, Nicolas Leu, Patrick D'appollonio Vega (@Marlex), Michael Conner, Jonathan Renpening, Mario Brazina, Simon, @MayankMisra, Thia "T" Nilawat, Carolyn Harp, Justin Blaisdell, Ferris Chia, Jeffrey Joy, Luke Baze, Sun Environmental, Carol Stanek- Markel, Sean Gomer, Pulsar9, Peter D, Pierre Gauthier, Andrew J Nicholls, Kevin Bosak, Sean Johnson, Adan Gabriel Gutierrez, W. D. Powell, John Kent, Keiran O'Neill, Beau Hardy, Michael Herman, Timothy Kim, Zach Hagen, Cheryl Howard, Ivan Ryabiik, Skyler Kanegi, Josh Browning, VoidedGin, Wonuk Joshua Jo, Rami AlGhanmi, Jeffrey Pannett, Alex W., Peter Digby, Moutemoute38, Barry Hay, Owen Gibson, Sohan S, Michael DiBlasi, Cmdr. Squirrel, Mijo Safradin, ChiHong Lin, Tomatico, Sergio Esteban Galvez Martell, Travis Hawk, Dex, Russ Neff, Anthony R. Junk, Nicholas DeFonde, Joel A. Garza, Kyle McCleary, Zach Toben, Peter Ward, Joaquin Johnson, Lai XueYang, Ryan M. Allen, Gern Blanston, Marc 'Hiro' Horstmann, K Roberts, Dan Perrera, Christopher Rush, Shane Boyce, Stephen W., Gershom Gannon-O'Gara, Brent Jones, Brian Dickens, Chris Corbin, Matt Weeks, Tripp Cook, Arman Chahal, terryowen, Daniel Isaac, Grant A., George Dwyer, Joseph Corcoran, Drew Crabb, John Gallagher, Hari.T, Avi Keith Jansma, Skyler Forshage, Martie F., E A, Devin A Kluk, Ken Brown, Thanasis Dimas, Jake Prime, David Pong, Jonathan Kennedy, Matt W., Alan Lehman, Benny W. Toho, Becki Bradsher, Jacob Goldstein, Jason Hulin, Sean Greenhalgh, Rebekah K., Vairavan Laxman, dwight, Timothy Boan, Liz Cherene, Suvasin Hosiriluk, Evan Wiederspan, Shane J. Veness, DMontgomery, MK, C Pedersen, Jeremy Faircloth, Andrew Mullen, Mark B., Michelle Xue, choward8, Ratherjolly Sponholz, Eliana Rabinowitz, Clarence Ingram, amir, Kai Nikulainen, Fishnets88, Philippe J. Giabbanelli, Joseph Andersen, Karol Werner, Alex Lee, Arturo Sanz, Justin Lewer, Evan Vandermast, Arvindar Narasimhan, Alex Cason, mindwarp, Truls Bjørvik, Adam Reed, Humberto "SilverOne" Carvalho, PE Reiman, Adrien Arculeo, Marcus "Makzter" Olander, Asif Ahmed, Jenny Lau, Afif Whykes, Wolfgang Amadeus Mozart, Jessica Rausch, Bilawal Hameed, Philip M. Woite, Daniel Milam, Brian Luckenbill Francesco Rinaudo, Robert L. Townsend, Joe Watkins, Michael Woo, Yui-Man Mak, Drew Johnston, AnotherKlog, Alik S, James Ingham, Tony Clough, Slaughtz, Chelsea - QU- BD, Tj c, Efrat Pauker, Snappy the Snapster, GhostReven, Dan Jenkins, Kallahir, Nathan Greco, Bilal Ahmed, Napoleon Wennerstrom, Brad Sparks, John Barrett, maxCohen, Yvonne Law, Scott Forney, Matthew R. House, Robert M. Froese, Bryan Ng, Phill Portlock, Matthew Erwin, Gorle, Radhakrishna, Mike Richardson, Abundance media Group, SF, CA, TheFatOne, Michal Sedlak, Mike Patterson, Louis Helscher, Josh Philips, Phossil, Charlie Tan, Maxim Kharlamov, Jordan Vittitow, Marcel Barros Aparici, Piotr Boczo?, Matthew Cheok, Frank Claycomb, Stig Harald Gustavsen, Noel Santiago, Jason Campisi, Debora Richards, Ninjabot, Nickolas Carter, Oregon Manuel Succo, MrMookie, Joshua Benton, M Cochran, Brian Ricard, Tabi, Cody Oss, Bren Rowe, Jonathan Tomek, Jonathan Wambach, Erih G., Maria K, matt, Karl "hair10" Markovich, Jace Ferguson, Gretchen Imbergamo, Alexander Tushinsky, Michael & Brandi, Mark Klamik, Erik R Velez, Michael Pellman, Sheldon Sawchuk, dragon4ever, Matthew Fisher, Scott Clausen, David Hunter, Kevin McKenzie, Hector Saldana, Andrew Saunders, Athanasios Kountouras, Christopher Kay-Chalk, Rusty Stiles, M. C. Rosen, Kai Zhao, Michelle Poirier, Andrew K., Anna Rozenbaum, Nader Khalil, Ismail Morgenegg, Chris Roark, Dean Van Beelen, Ron Vickers, Nick Tome, Eric Nikolaisen, SLA, Kick Molenveld, VIC, Nicholas Smith, Hugh Jass, Eric Rodriguez, Hannes Honkasaari, Anthony Barranco, Espen Nilssen, Chris Stratford, Callum Griffiths, Kjetil Svendsen, Brad Rosenboom, Christophe MOUCHEL, Jacob Shmerkovich, GaryWith1R, Craig O Connor @Craigieoc, Justin Haydt, William Woodward Sharp, Glowhollow, Michael Lass, Lachlan Collins, Little Rory, Eng. Ahmed A. Alshaia, Dave Ruppel, JC, Aguirrex, Max Lupo, Greg Horvath, Daniel Gourry Bustos, Gregor Wegberg, Dillon Development Corporation, Kyle Owen, Mark Wert, Stephen Holt, Tiffanie Chia, Naim Busek, Anthony Z., Jacob Evan Horne, James H Lin, Rob Scott, Sam Libby, Joseph Cicero, Martin Schmidinger, Logan Gittelson, P4XEL, Eric Blohm, sipp11, Supporter from Concord, MA, Hadora, Tyler Shipe, Spencer Cree, Steve Sartain Sartain, Peter Gerrit Lowe, Ron L., Nadine Campbell, Aiman Ahmed, Jeffrey J.F. Rebultan, Michael Brown, burnabytom, Jigga Jay Ricciardi, Michal Muhlpachr, Aaron Fader, Daniel Scott, Crashman2600, Ian R Wright, Mirza Joldic, Robert Blackwell (rblackwe), David "Crutch" Crutchfield, Grant "Beast" Harper, Spence Konde, Michael Wan, Eric Kilroy, ron ursem, Rebecca H Goldman, Samuel E. Giddins, Jeanclyde Cruz, Vince P, Johnny J., Alexander Zawadzki, William Clements / Chicago Technology Group, Allen Porras, Udrea M, Joled Koh & Laywen, Renno Reinurm, Alex Wixted, Levi Blue Maggard, Ken Mcewan, Austin Butcher, Benjamin Durie, Jimmy Hsieh, Dwinf, Sir Adelaide, Blake Laroux, Lucas Gronlund, Simon Jubinville, Aerie Ward, Thim Frederik Strothmann, Chasester, Laughing Magi, Wagner, Earl, brianwik, Owen Ness, Flavio Massayoshi Oota, bfishadow, Bill E, Frank Wiebenga, Nick Keehn, Alex Tirrell, Tom Higgins, LP, Tyler Cushing, Irina Pinjaeva, Tim Roark, JP, Boris Gutbrod, Adam, Benjamin Miramontes, Sr., Anatoly Myaskovsky, Joseph Welt, domhaase, Jonathan H. Chaney, Carlos Piñan, Matt Varian, Michael Ruppe, Bernie Sobieraj, Kevin Bacon the Third, Yusuke Kaji, Lee "TieGuy" Bratina, Joel Galeran (Twitter @ChibiJoel), az, E. Scott, Will Phule, Jonathan Hill, Daniel Waugh, Alberto Rodriguez Barbon, Omar Zahr, Juuhhhstin, Gav Reid, Conny Johansson, LV, Monique L. Sakalidis, Martin Glassborow, Dan Gadigian, Vincent.gagnon5@gmail.com, Ed O'Brien, Jimmy Bhullar, James McNamara, Hamad B. Al-Gharabally, Michael Yu, John Saunders, Mark Kent Malmros, Anton Harris, I Ketut Parikesit, Kevin O'Donnell, Sait Mesutcan Ilhaner, Conor D. Cox, Lucas Eaton, Mike Rotch, Jess Zhu, Anne Marie Uy, G Malone, Nathan Hurd, John L. Grant, III, Eric Joel, Adil Shakour, Scott Wiley, Sam Hunter, Laura R. Alonso, Doug K., NardoXing, Ben Blair, Andrew James, Joe Sparacino, Shawn Albert, Puipui Ng, Shane Alexander Mott, Seth Williams, Kristian Hill, Drew S, Sandra Hor, Gpurd, Robin Orlic, Armel C., Justin Duino, Christophe R. (Darwiin), etchasketcher, _r00k, Tomas Hernandez (Ttoti), Elena Murphy, Ryan Riley Aiden Fisher, Luke Thomas, Riel Chu, nameasitappearsonlist, Philip Freidin, tf, Nathan Emery, Andrew, Jesse Ventura, Dan Kohlbeck, Oz du Soleil, Data Mercenary, Phil Watts, John Pinto, Yuval Langer, Simon Evans, Steve Turczyn, GetAdditive.com, Nick Snowden, Kris, Michael Litman (@mlitman), Peter Ung, Fargo Jillson, Patrick Yevsukov, Dee Bender, Batbean, Aris MIchalopoulos, Threemoons, Christopher Yap, Tim Williams, O. Johnson, David N Buggs, Myles Kelvin, M. Leiper, Brogan Zumwalt, Roy Nielsen, Jaen Kaiser, Joe C., Emily Chen, Bryan Powell, SayRoar.com (Film & TV), Alan Hatakeyama, Chris Pacheco, Alex Galick, p a d z e r o, Juri Palokas, Gregg Lamb, Lani Tamanaha Broadbent, Ami noMiko, Aaron Harvey, Angel Sicairos, Shiloh N., Katherine Allaway, AlamoCityPhoto.Com, John Laudun, Greg Reyes, Jagmas, Dan Allen (QLD, Australia), Dustin Niehoff, Ag, Scott M., Esben Mølgaard, Ian McShane, Timothy Gibson, Que, Janice Whitson, Babur Habib, Brent Cappello, Meep Meep, Justin G., Stuart Woods, Ryan Cook, Mike R., John Surdakowski, Ehren Cheung, powerful hallucinogens, Robert Drake, Steve Rose, Trenshin, Meeghan Appleman, Hanning "Butch" Cordes!, Jose de Jesus Medina Rios, Guadalupe Ramirez, Andrew Willerding, NathAn Talbot (MB), Jorge Esteban, bchan84, irfans@gmail.com, Krish, Vaughn Valencia, Jeromy hogg, Jorge Hidalgo G., zombieturtle, Mors, Rick D, Rob Schutt, Wee Lin, incenseroute.com, IceTiger, James E. Baird Hulusi Onur Kuzucu, TheHodge, Yannick Scheiwiller, Robin arvidsson, Oliver Diestel, Daniel Knell, Elise Lepple, Frank Elias, Shaun Budding, Shane Williams, Chin Kun Ning, Eike Steinig, Hogg Koh, AaronMFJ, John P. Doran, M Blaskie, Eric 'Harry' Brisson, Chris "Krispy89" Guerin, Duck Dodgers, Jonathan Coon, Sally Jo Cunningham, Joe Desiderio, Anon, Mike Sison, Shane Higgins, Russell Wee, Gabriel B., Thomas Sebastian Jensen, Amy Pearse, James Finley, Mikhail Ushanov, James AE Wilson, Michael Dussert, Felix Varjord Soderstrom, Eric Metelka, Stephen Harland, muuusiiik, Shandra Iannucci, Joe Racey, Cook66, Nicholas R. Aufdemorte, Justin Marsten, Barrett, Zachary Smith, mctouch, Donald D. Parker, Rob Silva, Phillip Vanderpool, David Herrera, Otto, Roland Santos, Peter the Magnificent, Brandon B., Brett Russell, Joe Carter, Andrew Paulus, Peter Harris, Brian Kaisner, Stefan Gobel, Melissa Beher, Jesse B - PotatoHandle, pwendelboe, Matthias Gindele, Andy Smith-Thomas, Elizabeth S., Erez Simon, Andrew Cook, Wouter van der Aa, Iain Cadman, Kyle Bishop, Andrew Rosen, Alessandro "juju" Faraoni, GeekDoc, Arran Stirton, AMCD, Eddie Nguyen, Steve Stout, Richard Bolla, John Hendrix III, Pallav Laskar, Scrivener, Bobbinfickle, Vijay Shankar V, Zach Peskin, Mark Neeley, oswinsim, Joe Briguglio, Stacy E. Braxton, Alan L Chan, Markus Fromherz, Jim Otto, Neil Ewington, Sarah New, Harish Gowda, Eva Vanamee, Peter Knepley, RVPARKREVIEWS.COM, Brad Wheel, Eric Gutierrez, Jeff Wilden, Dave Coleman, Brian S Beckham, Bill Long, Jeremy Kirsch, Tim Erwich, Ryan Valle, John Palmer, Rick Skubic, Vincent Parker, David Von Derau, Jonathan Wang, Chris Stolte, Thomas Boettge, Jochen Schmiedbauer, Dirk Wiggins, David Recor, Joshua Hepburn, Pelayo Rey, Jabben Stick, Amit paz, Rob B., Art?rs Nar?ickis, Merrick Royball, Jerome amos, Soba, Varian Hebert, Geoff A., Dave Strange, Roy Arvenas, Ryan S., Suresh Kumar, Stefan Nygren, John H, Justin A. Aronson, Dave C, Keegan Willoughby, Martin-Patrick Molloy, David Hay, Jeff Beck, Sean Callaghan, Greg Tuballes, Mark Filley, Somashekar Reddy, Jorge Palacio, Glen Andre Than, Garrett N., Garry Bowlin, Sathish K, Lucas A. Defelipe, Michael Roberts, Norman Sue, Tommaso Antonini, Herbert, Frank Ahrens, Uberevan, Andy Schofield, Amir Prizant, Bennett A. Blot, Rob Udovich, Holli Broadfoot, Ray Lee Ramirez, Jeffrey Danziger, Kevin Rhodes, Brendon Van Heyzen, Jeff Simon, Jamie E. Browne, Vote Obama 2012 !, Wel, n33tfr33k, J. Phillip Stewart, dham, Ove Kristensen, Phillip Lin, Steve Paulo, Jerry Fu, Chris Wray, Daniel Schwan, Sean Cantellay, Azmi Abu Noor, Lucas Van Deun, The Mutilator, Isaac Amaya, chandradeep, B. Graves, Benji, Leonard Chan, James Smeltzer, George Ioakimedes, Andrew Keh, Bobby T. Varghese, Sir Nathan Reed Chiurco, Christian Nissler, Ethan Johns, David Strakovsky, Leslie Harbaugh, AdamGross, Darren Koh, Matt Palmer, Michael Anthony Sena, Blade Olson, Larry Sequino, jeremy levitan, Rahul Shah, Mike Schulze, Smallbee, Mark Johanson, MS, Pat Ferate, Dennis Chan, Matthew D Johnson, Jefferson Tan, Eric G, Jordan Tucker, Steffen Reinhart, Benjamin Rea, Brendan Burke, Oppa Gangam Style, P B Dietsche, Daniel Lauer, Jon D, JFV, Jeremy Grise, James Tang, Jean-Philippe Roy, The Guru Guys, Matthew Chan, Anna Parikka, moh maya, Armand, Robert Porter, Jeff Hanlon, Megan Olea & Chad Tungate, Marko Soikkeli, Jamie Finlay, Jack Collison, Ollie Silviotti, Andrew Hayden, Jay D

Wilson, Pureum Kim, Mike Mudgett, Jerry Horn, Luca Rossetto, gulaschsuppe, Martin Langbraten, filupmybowl, J. Lawler, Ron Averyt, Christopher H. Y. Chiu, John Lee, Justin Nguyen, Antoine BERTIER, fuultier@gmail.com, Adrienne C, Larry Tierney, Maris Gabalins, Marco Bulgarini, Simon Vahr, Sebastian Hojas, S Liew, Helena Montauban, Xiaoying Zhang, Alan H, HicEtNunc, Jonathan B. Johnson, Siam, Thomas Richter, Marc-Andre Parent, Kristian Nybo, Jim and Sam Maloney, ian remulla, emmanuelordonez12@yahoo.com, Ulf Eliasson, James Murphy, Adrian, Rodrigo Konrad, Simon (pigeon head) Jack, Richard Huber, Andrew Hinett, Robert C., Catherine, Felipe Ferreira de Oliveira, Mike P, Nate Fowler, Simon Khaimov, Kurt Liebegott, segnorblues, Kevin Glaser, Bill Fischofer, Graham Peacock Heath, Jared Pennell, Blaine Hilton, Maurizio, Italy, Juha Jaaskelainen, Sarah Leadbeater, Brian J Lester, somair, Jakub Stano, John Buzzi, The Fish, Greg A., Nichole Austin, Erik Wilthil, Nathaniel Griggs, M.Bair, Marwan Alzarouni, Simon Belkin, Thebeersnobs.com, Gary Harmon, Johan Oberg, Stephen Brown, Matthew Dwyer, Julian Naydichev, Greg Ortega, Will Manidis, Jogender Nagar, Brian Paul, Keith Sanders, Chris Ferreira, Chad Whittaker, Sonic, Phinehas Bynum, Jeremy Jones, Mark Lindsey, Koty Yell, Benjamin Fernandez, Christopher "Berryman" Berry, Aaron Gersztoff, Kel Stewart, Brian Taylor, John Hutchins, Michael S. Watson, Robert, Dave Sergeant, Ted Reardon, Darcy Kohls 6apcyk, David Rosen, Jacob Minne, Rich, Ron Swanson, Badboychoi, Shane Bailey, Xiaoyang, M. Wuerz, Marty Otto, Leonardo M. M. Jorge, arkuana, mary grace lim, Bellefonte, K. Mathis, Joel Fishbein, Neal Mullin, Stephan H. Wissel, Alex Jeuck, Bri Bear, Dave LeCompte, Alessandro B., Boding Zhang, Timothy Jack, Daniel Simaan, Troy "Wrongtown" Hall, Veli Kerim Çelik, Bhaskar Krishna, Pawel Wlodarski, Curtis Richardson, Anders Lindvall, Branden Laurie, Angus Haynes, Nathan Bain, Jacques Schaefer, Spongefile, John Reeser, William Limratana, Sascha Kiefer, Sikander Lhote, Stefan Larsson, Bobby Rester, David L Hill, Daniel To, Austin HIll, Bruce Turner, Darryl D'Aoust, James Salazar, Gregory D.Tsikis, John Goerl, Megat Denney, Jon T. Nilsson, Lam Kee Wei, Chia Yeak,Wong, Ondrea Graye, David M. Graf, Mark Rogerson, Troels Bager, Don Healey, Brandon Cushenberry, Daniel Jeffery, john liu, Iddeen Khairul, YanTing, deadly22sniper, ARNLAO, Jonah Eaton, Mark Ferguson, J Wagstaff, Tina Preston, Kevin J. Lee, Anthony Perrett, Rich Watts, Robert Reagh, Sukanta Ganguly, D. Nielsen, Maryam L., Jeremy Self, Marcus Pearl, Andrew R Wasem, Philipp C. Opitz, Joshua Michael Nicholas "Lefty" Turner, Emmanuel Beltran, Bryan Thurnau, DBone, Daniel "Heru-ur" Lagerman, Oliver Marshall, Matthew Deloughry, Jothi, Joe Staton, Steve Seward, Hotau, Joshua Christain, Bryan Parry, Doug Philips, Brian Chia, Mauricio @Reyes, Erik Schwartz, George Cammack, Nico S., J. Gizard, Jeff Kujath, Dave, Brian Huqueriza, Guðmundur Borgar Gislason, Dave Poirier, Ferran Selles, Michael "Maka" Gradin, Jack Dinze, Klas Gelinder, Kyle Ingrelli, peter, Christine C. Heisler, Kyle Brown, Yannik Schreckenberger, Shiki Cheefei Lee, Sean Hames, David Tran, James Roberts, Joshua Bowden (aka Jaxrtech), Amiya Gupta, Bryan A, Brion Cook, C Gunadi, Scott Morrison, Hagbard Celine, Thomas Vassøy, Cliffton Hillmann, Chance Evans, David W Luna, Adi Kamdar, Peter M. Kersulis, Joseph Perera, Rifqi, Adam Schuster, Nadine Ong, AkNetworking, Aaron Zukoff, Frank Tilley, Dustin Hall, John M. Kuzma, Felix Kaefer, BarbMacK, Rohan Martin, Francis James Basco, W. Montes, Howard M. Lawson II, Michael Milliken, Klaus Thiel (Munich-GER), Richard Dusold, John T McGee II, William Nichols, Gerad Troje, Keith Rebello, bytemap, Jason S., Haynes Chewning, Otinane Epidio, Aftah Ismail, John K. Estell, David Byworth, FIGGE, Jason Gassel, Greg Matyola, Eric L. Wold, Glenn Fetty, Kevin McGill, Nathan J. Sheusi, M A Shevloff, Katie Camille Friedman, Greg Thompson, Galina Zubkov, Adam D, Scott Paterson, Hutch-CaneVentures.com, Thomas Thrash, Benoit Lachance, Pablo Rivas, Skip Surette, Mathew Buttery, FRiC, Celeste H. Galvez, Christopher Bonner