An introduction to Python through practical examples

Fletcher Heisler

0) Introduction

Whether you're new to programming or a professional code monkey looking to dive into a new language, this book 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 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.

0.1) Why Python?

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 ()
{
  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.

0.2) Why this book?

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:

1) They aren't practical.
2) 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 book 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 book won't be enough by itself – but it will *still* be the best starting point. Once you've mastered the material in this book, 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.

0.3) How to use this book

For the most part, you should approach the topics in the first half of this book 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 book 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.

If you get stuck, you can always log in at RealPython.com and ask for help on the members' forum; it's likely that someone else has already experienced the same difficulty that you're encountering and might be able to guide you along.

This book 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.

0.4) License

1) Getting Started

1.1) Download Python

Before we can do anything, you have to download Python. Even if you already have Python on your computer, make sure that you have the correct version: **2.7.3** is the version used in this book and by most of the rest of the world.

! There's a newer version, Python 3.3, but it can't run code that was created with previous versions of Python (including a lot of useful and important packages that haven't been updated). As a result, Python 3 still hasn't caught on yet. Since most of the code you'll see elsewhere will be from Python 2.7, you should learn that first. The two versions are still *very* similar, and it will take you very little time to get used to the minor changes in Python 3 after you've mastered Python 2.

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 2.7.3 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 Python 2.7.3 installed by default. Open your Terminal application and type "`python --version`" to find out. If you have 2.7.1 or 2.7.2, you should go ahead and update to the latest version.

If you need to, go to http://www.python.org/download/ to download **Python 2.7.3** for your operating system and install the program.

1.2) Open IDLE

We'll be using IDLE (**I**nteractive **D**eve**L**opment **E**nvironment) 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.

Windows: Go to your start menu and click on "IDLE (Python GUI)" from the "Python 2.7" program folder to open IDLE. You can also type "IDLE" into the search bar.
OS X: Go to your Applications folder and click on "IDLE" from the "Python 2.7" folder to start running IDLE. Alternatively, you can type "`IDLE`" (without quotes) into your Terminal window to launch IDLE.
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!

1.3) 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
>>>
```

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
>>>
```

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"
```

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” (without the quotation marks) and save it somewhere you'll be able to find it later. The “.py” extension lets IDLE know that it's a Python script.

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), 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.

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.

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.

1.4) 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 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:

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

1.5) 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`.

! 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`.

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:

- Using the interactive window, display some text on the screen by using `print`
- Using the interactive window, display a string of text by saving the string to a variable, then `print`ing the contents of that variable
- 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.[1] 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" """
```

1 The # symbol is alternately referred to as a number sign, a hash, a crosshatch, or an octothorp. Really.

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.

2) Fundamentals: Strings and Methods

2.1) Learn to speak in Python

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):

```
phrase = 'Hello, world.'
myString = "We're #1!"
stringNumber = "1234"
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:

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

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

Here we assigned one value to the variable `longString`, then we overwrote that value with a new string literal. Try putting this in a script and then `print` the variable `longString`; 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!

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:

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

! 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. We'll mostly stick to camelCase in this course, but another popular method is to separate words using underscores (i.e., `my_long_string`).

Review exercises:

- `print` a string that uses double quotation marks *inside* the string
- `print` a string that uses an apostrophe (single quote) *inside* the string
- `print` a string that spans across multiple lines
- `print` a one-line string that you have written out on multiple lines

2.2) 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:

```
>>> myString = "abc"
>>> lengthOfString = len(myString)
>>> print lengthOfString
3
>>>
```

First we created a string named `myString`. Then we used the `len()` function on `myString` to calculate its length, which we store in the new variable we named `lengthOfString`. We have to give the `len()` function some *input* for its calculation, which we do by placing `myString` 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"
>>> magicString = string1 + string2
>>> print magicString
abracadabra
>>>
```

Or even like this, without creating any new variables:

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

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

! You'll see a lot of italicized 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**	...
Index / Subscript #:	0	1	2	3	4	5	...

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.

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:

```
myString = "goal"
myString[0] = "f"  # this won't work!
```

Instead, we would have to create an entirely new string (although we can still give `myString` that *new* value):

```
myString = "goal"
myString = "f" + myString[1:]
```

In the first example, we were trying to change *part* of `myString` 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 `myString`; *then* we took that *new* string and completely reassigned `myString` to this new value.

Review exercises:

- Create a string and `print` its length using the `len()` function
- Create two strings, concatenate them (add them next to each other) and `print` the combination of the two strings
- 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
- `print` the string "`zing`" by using subscripting and index numbers on the string "`bazinga`" to specify the correct range of characters

2.3) 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.[2] 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

2 Strings are actually called `str` objects in Python, because programmers are lazy and don't want to type more than is absolutely necessary. But you'll always hear string objects referred to as just "strings."

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:

```
>>> loudVoice = "Can you hear me yet?"
>>> print loudVoice.upper()
CAN YOU HEAR ME YET?
>>>
```

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

! 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(loudVoice)
```

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 `loudVoice`:

```
loudVoice.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 `raw_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:

```
userInput = raw_input("Hey, what's up? ")
print "You said:", userInput
```

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![3] Enter a response, and it will store that answer in the `userInput` 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 `raw_input()` with the string method `upper()` in a script to modify the user's input:

```
response = raw_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:

```
>>> myString = "kerfuffle"
```

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

```
>>> myString.
```

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!

3 Notice that we added a space at the end of the string we supplied to `raw_input()`. This isn't mandatory, but that way the user's input doesn't show up on the screen *right* next to our prompt.

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 "`myString.u`" and then hit the `TAB` key, IDLE will automatically fill in "`myString.upper`" because there is only one method belonging to `myString` that begins with a "u". In fact, this even works with variable names; try typing in just the first few letters of "`myString`" and, assuming you don't have any other names already defined that share those first letters, IDLE will automatically complete the name "`myString`" for you when you hit the `TAB` key.

Review exercises:

- Write a script that takes input from the user and displays that input back
- 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 2.3: 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 chapter 5.

3) Fundamentals: Working with Strings

3.1) Mix and match different objects

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:

```
>>> myNumber = "2"
>>> print myNumber + myNumber
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:

```
>>> myNumber = "12"
>>> print myNumber * 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 `myNumber` into an integer or a "float" like so:

```
>>> myNumber = "12"
>>> print int(myNumber)
12
>>> print float(myNumber)
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:

```
>>> myNumber = "12.0"
>>> print int(myNumber)

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print int(myNumber)
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 to do 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:

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

3.2) Streamline your `print` statements

Suppose we have a string object, `name = "Zaphod"`, and two integer objects, `numHeads = 2` and `numArms = 3`. We want to display them in the following line:

```
Zaphod has 2 heads and 3 arms
```

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(numHeads), "heads and", str(numArms), "arms"
```

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

```
print name+" has "+str(numHeads)+" heads and "+str(numArms)+" 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, numHeads, numArms)
```

The pairs of empty curly braces (`{}` without any space in between the two) serve as

place-holders 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, numHeads, numArms)
```

Here we've inserted `name` into the `{0}` place-holder because it is the 0th input listed, and so on. Since we numbered our place-holders, 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(numArms, numHeads, 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.
>>>
```

Finally, 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 {numHeads} heads and {numArms} arms".format(name="Zaphod",
numHeads=2, numArms=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.

! 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 is being phased out completely in Python 3 (and the "new" `format()` style works just fine in Python 2.7), so there's not much point in learning to master this outdated method.

Review exercises:

- 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.
  ```

- Display the same line using `format()` and empty `{}` place-holders
- Display the same line using `{}` place-holders that use the index numbers of the inputs provided to the `format()` method
- Display the same line by creating new string and float objects inside of the `format()` method

3.3) 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:

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

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

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

Review exercises:

- In one line, display the result of trying to `find()` the substring "`a`" in the string "`AAA`"; the result should be `-1`
- 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
- Write and test a script that accepts user input using `raw_input()`, then displays the result of trying to `find()` a particular letter in that input

Assignment 3.3: 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.
>>>
```

4) Fundamentals: Functions and Loops

4.1) Do futuristic arithmetic

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.

! 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
>>>
```

Wait a second... 5 / 2 = 2? For a computer, that doesn't sound very accurate!

Unfortunately, the mathematics aficionados (read: über-nerds) who developed Python made division a little counter-intuitive at the cost of being *technically* correct. Since we're dividing two integers (i.e., whole numbers), Python is doing *integer* division and ignoring the remainder.

Since this is an annoying quirk that we usually don't want, we have to change how Python treats division by adding this bizarre line once at the very beginning of our script:

```
from __future__ import division
```

That's `__future__` with *double* underscore characters (two _'s on each side). **I promise that this is the least sensible line of code I will ever have you write.** Just remember to add it to the top of any script where you might want to divide one integer by another, and all will be well. Coincidentally, "normal" division by default is one of the few useful changes in Python 3, hence why we need to `import` this functionality "from the future."

If you *really* wanted to do integer division, just use the `//` operator instead. For instance, `5 // 2 = 2`, even if you've already imported futuristic division.

This only applies to the `//` operator. The `**` operator is something different entirely, and actually calculates one number raised to the power of another number. For instance, `5 ** 2 = 25`.

Now that we've imported the ability to divide integers correctly, we can run a script like this:

```
from __future__ import division
print "1/2 =", 1/2
```

And we'll get the expected result of `1/2 = 0.5` instead of `0`.

However, notice that if we try to evaluate `1/2` in the interactive window, we still get `0` as our answer - even after running our new script. This is because we haven't actually added this new "division" functionality to the interactive window. But we can always do that separately as well:

```
>>> from __future__ import division
>>> 1/2
0.5
>>>
```

And now our interactive window will also divide numbers the way we would expect.

Review exercises:

- In a script, `print` the result of dividing one integer by a second, larger integer; this is *integer division*
- Import division from the "future" into the same script, then save and rerun the script; compare these results to the previous version

Assignment 4.1: Perform calculations on user input

Write a script "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 highlighted in bold):

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

Keep the following in mind:

- **In Python, x^y is calculated by using the expression `x ** y`**
- **Before you can do anything with the user's input, you will have to store the results of both calls to `raw_input()` in new objects**
- **The `raw_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**
- **You should use the string `format()` method to `print` the result**
- **You can assume that the user will enter actual numbers as input**

4.2) 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.

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 * number
    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 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 5^2 and returns the value of the variable `sqr_num`, which gets stored in our new variable `output_num`.

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 returnDifference(num1, num2):
    return num1 - num2
```

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

```
print returnDifference(3, 5)
```

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

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 returnDifference(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(returnDifference)` and see:

```
>>> help(returnDifference)
Help on function returnDifference in module __main__:

returnDifference(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.

Review exercise:

- 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
- Write a function `multiply()` that takes two numbers as inputs and multiples them together, returning the result; test your function by saving the result of `multiply(2, 5)` in a new integer object and `print`ing that integer's value

Assignment 4.2: 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:

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

4.3) 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.

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**.[4]

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

4 ...or at least until you manage to forcibly quit out of IDLE using your panic key combination of choice.

would get confused - even though the spacing *looks* the same to you!

(You *could* use tab characters everywhere as indentation - or even use a different number of space characters - as long as you always use the *same* type of indentation for a particular section of indented text.)

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.[5] 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`.

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`

5 If you have a very large `range()` of integers and memory considerations are an issue, then you should use the function `xrange()`, which works the exact same way as `range()` and is more efficient. This is because `xrange()` looks through the integers one at a time, while `range()` creates the full list of numbers first. (In Python 3, `range()` works like `xrange()`, which in turn no longer exists.)

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.

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:

- Write a `for` loop that prints out the integers 2 through 10, each on a new line, by using the `range()` function
- Use a `while` loop that prints out the integers 2 through 10
  (Hint: you'll need to create a new integer *first*; there's no good reason to use

a `while` loop instead of a `for` loop in this case, but it's good practice…)

– Write a function `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 4.3: 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.[6] 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
```

6 The only math involved here is that the initial (principal) investment amount is multiplied by 1 + the annual return rate every year. For instance, after one year at a 0.05 rate, $100 → $100 x 1.05 = $105.

***Hint*: 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:

- **You will need to use a `for` loop over a `range` that's based on the function's `time` input.**
- **Within your loop, you will need to reassign a *new* value to the `amount` every year based on how it grows at `1 + rate`.**
- **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.**
- **Using the `print` statement by itself 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.

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. 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:

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

```
phrase = "hello!"
print addUnderscores(phrase)
```

What we *meant* for the function `addUnderscores()` 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) 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-clicking) 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.

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.

5) Fundamentals: Conditional logic

5.1) 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:

`a > b`	`a` **`greater than`** `b`
`a < b`	`a` **`less than`** `b`
`a >= b`	`a` **`greater than or equal to`** `b`
`a <= b`	`a` **`less than or equal to`** `b`
`a != b`	`a` **`not equal to`** `b`
`a == b`	`a` **`equal to`** `b`

The last two symbols might require some explanation. The symbol "`!=`" is a sort of short-cut 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 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:

- 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"
```

5.2) 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.

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):

- cats have four legs
- 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 < 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`
- **`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`
- **`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 in a table:

Combination using `and`:	Result is:
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`

False and False	False

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."

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 "ex*clusive 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 in a table:

Combination using `or`:	Result is:
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
>>>
```

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

Effect of using `not`:	Result is:
`not True`	`False`
`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:

  - We know that `1 == 1` is `True`, therefore…
  - `1 != 1` is `False`, therefore…
  - `not (1 != 1)` can be simplified to `not (False)`
  - `not False` is `True`
  - Now we can use this partial result to solve the full expression…

- `True and not (1 != 1)` can be simplified to `True and True`
- `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:

- We'll start with the expression on the left side of the "`or`" keyword
- We know that `"A" == "A"` is `True`, therefore…
- `"A" != "A"` is `False`, and the left side of our expression is `False`
- Now on the right side of the "`or`", `2 >= 3` is `False`, therefore…
- `not (2 >= 3)` is `True`
- So our entire expression simplifies to `False or True`
- `False or True` evaluates to be `True`

Note 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.

! You should feel very comfortable using various combinations of these keywords and boolean comparisons. Play around in the interactive window, 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:

- 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)
```

5.3) 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 each 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`. 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 be `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 was `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 more possible options after an `if` statement. For instance, we could combine an `if`, and `elif`, and an `else` in 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` are working 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` or the `else` blocks 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, first we 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:

- Write a script that prompts the user to enter a word using the `raw_input()` function, stores that input in a string object, 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 5.3: Find the factors of a number

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 an 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 `raw_input()` returns a string, so you will need to convert this value to an integer before using it in any calculations.

5.4) 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
>>>
```

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`:

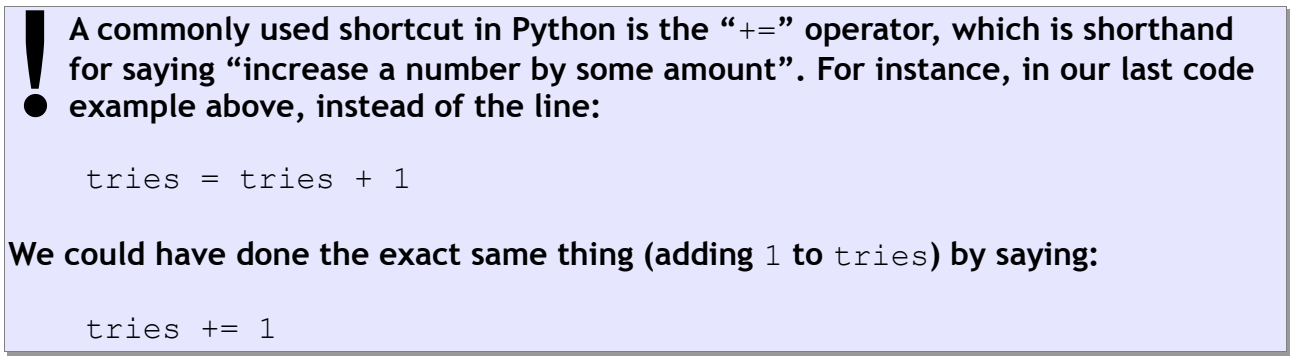
```
phrase = "it marks the spot"
for letter in phrase:
    if letter == "X":
        break
else:
    print "There was no 'X' in the phrase"
```

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, however, we reached the else statement and displayed that "X" wasn't found. If you try running the same program on the phrase "it marks Xthe 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.[7]

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 = raw_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
```

7 Of course, it would have been easier to use the result of phrase.find("X"), but you can't expect *all* these examples to be sensible…

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:

```
trials *= 3
```

Review exercises:

- 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:`"
- 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 as explained in Assignment 5.3

5.5) 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 red text.

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(raw_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 have "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:

```
from __future__ import division
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(raw_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.[8]

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
```

8 Notice how we used 10.0 in the actual division? This was just an easy way to avoid importing integer division by using a float (10.0) instead of another integer (10) in the division.

```
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.

Review exercises:

- 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

5.6) 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 __future__ import division
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:

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

Assignment 5.6.1: 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.

If you want to check your work without looking at the sample script, the answer is in this footnote.[9]

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 5.6.2: 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 `flip = 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:

- You will need to use a `for` loop over a `range` of `trials`.
- For each trial, *first* you should check the outcome of the first flip.
- Make sure you add the first flip to the total number of `flips`.
- 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.

9 Candidate A has approximately a 63% chance of winning; Candidate B has roughly a 37% chance.

6) Fundamentals: Lists and Dictionaries

6.1) Make and update lists

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 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]
myList = ["one", 2, 3.0]
```

The first object, `colors`, holds a list of four strings. Our `scores` list holds five integers. And the last object, `myList`, 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']
>>>
```

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 strings in the list `animals` to the list `large_cats`, then we added another string into `large_cats`. But when we display the contents of `animals`, we see that we've changed our original list - even though we meant to assign in the other direction, 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, 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.[10]

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:

```
large_cats = animals[:]
```

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.

10 Note that this wasn't a problem with simple objects like strings because they're immutable; since we have to completely reassign these objects, we *can't* affect one reference by *changing* another object.

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.

Since a list is just a sequence of objects, 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"
>>> groceryList = groceries.split(", ")
>>> print groceryList
['eggs', 'spam', 'pop rocks', 'cheese']
>>>
```

Review exercises:

- Create a list named `desserts` that holds the two string values "`ice cream`" and "`cookies`"
- Sort the `desserts` in alphabetical order, then display the contents of the list
- Display the index number of "`ice cream`" in the `desserts` list
- Copy the *contents* of the `desserts` list into a new list object named `food`
- Add the strings "`broccoli`" and "`turnips`" to the `food` list
- Display the contents of both lists to make sure that "`broccoli`" and "`turnips`" haven't been added to `desserts`
- Remove "`cookies`" from the `food` list
- Display the first two items in the `food` list by specifying an index range
- Create a list named `breakfast` that holds three strings, each with the value of "`cookies`", by splitting up the string "`cookies, cookies, cookies`"

Assignment 6.1: 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:

- 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"

- 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.)

- Plug the words you selected into the structure above to create a poem string by using the `format()` string method

- 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.

6.2) 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:

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

```
>>> myTuple[2]
'change'
>>> myTuple.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.

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:

- Create a tuple named `cardinal_nums` that holds the strings “first”, “second” and “third” in order
- Display the string at position 2 in `cardinal_nums` by using an index number
- 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

6.3) Store relationships in dictionaries

One of the most useful structures in Python is a *dictionary.*[11] 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:

11 In fact, Python calls them `dict` objects, and sometimes people refer to them as “dicts” for short.

```
>>> 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 duplicates 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 in the form of a list by using the `keys()` method:

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

If 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:[12]

```
>>> for contactName in phonebook:
      print contactName, phonebook[contactName]

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 contactName in sorted(phonebook):
      print contactName, phonebook[contactName]

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

12 Here we've written a loop in the interactive window; the blank line is because we had to hit backspace in order to make this line unindented so that Python knows where we want to end the loop.

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

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

Here we created a new dictionary named `simpleDictionary` 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:

```
>>> simpleDictionary = dict([("string1","value1"), ("string2",2),
("string3",3.0)])
>>> print simpleDictionary
{'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.

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:

- Create an empty dictionary named `birthdays`
- Enter the following data into the dictionary:[13]

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

- 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
- 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
- Delete 'Darth Vader' from the dictionary
- Bonus: Make the same dictionary by using `dict()` and passing in the initial values when you first create the dictionary

13 All birth dates are rough approximations based on the Galactic Standard Calendar.

7) File Input and Output

7.1) 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:

```
myOutputFile = 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 `myOutputFile`. Let's write a line of text into the file with the `writelines()` method. Our full script should look like this:

```
myOutputFile = open("hello.txt", "w")
myOutputFile.writelines("This is my first file.")
myOutputFile.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.

! 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:

```
myOutputFile = open("hello.txt", "w")
linesToWrite = ["This is my file.", "\nThere are many like it,", "\nbut this
one is mine."]
myOutputFile.writelines(linesToWrite)
myOutputFile.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`:

```
myOutputFile = open("hello.txt", "a")
nextLine = ["\nNON SEQUITUR!"]
myOutputFile.writelines(nextLine)
myOutputFile.close()
```

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

```
myInputFile = open("hello.txt", "r")
print myInputFile.readlines()
myInputFile.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', 'NON 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:

```
myInputFile = open("hello.txt", "r")
for line in myInputFile.readlines():
    print line,
myInputFile.close()
```

! 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.

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:

```
myInputFile = open("hello.txt", "r")
line = myInputFile.readline()
while line != "":
    print line,
    line = myInputFile.readline()
myInputFile.close()
```

There is an additional short-cut 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 myInputFile:
    for line in myInputFile.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, `myInputFile`). 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, this clean-up work 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 myInput, open("hi.txt", "w") as myOutput:
    for line in myInput.readlines():
        myOutput.write(line)
```

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:

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

print "Line 0 (first line):", myInputFile.readline()
myInputFile.seek(0) # jump back to beginning
```

```
print "Line 0 again:", myInputFile.readline()
print "Line 1:", myInputFile.readline()
myInputFile.seek(8) # jump to character at index 8
print "Line 0 (starting at 9th character):", myInputFile.readline()
myInputFile.seek(10, 1) # relative jump forward 10 characters
print "Line 1 (starting at 11th character):", myInputFile.readline()

myInputFile.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.[14] 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 behaviour is only useful in very specific cases![15]

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:

- Read in the raw text file "poem.txt" from the chapter 7 practice files and display each line by looping over them individually, then close the file; we'll

14 If we had passed a 0 as the second argument, `seek()` would act exactly as if we had only provided the first number; since 0 is passed as the "default" value, we usually don't bother to provide it explicitly.
15 For instance, seeking is great if you need to read or modify particular entries in a fixed-width file.

discuss using file paths in the next section, but for now you can save your script in the same folder as the text file

- Repeat the previous exercise using the `with` keyword so that the file is closed automatically after you're done looping through the lines
- 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
- Re-open “output.txt” and append an additional line of your choice to the end of the file on a new line

7.2) 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:

```
myInputFile = 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
myPath = "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).[16] 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 7 practice files folder, the sample code would look like the following:

```
import os
```

16 Non-Windows users will probably need to replace "`C:/`" with "`/home/yourusername/`".

```
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files"
inputFileName = os.path.join(myPath, "example.txt")
with open(inputFileName, "r") as myInputFile:
    for line in myInputFile.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:

```
inputFileName = myPath + "/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.

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
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files/images"
# get a list of all files and folders
fileNamesList = os.listdir(myPath)
# loop over every file in the main folder
for fileName in fileNamesList:
    if fileName.lower().endswith(".gif"): # extension matches a GIF file
```

```
        print 'Converting "{}" to JPG...'.format(fileName)
        # get full path name and change the ".gif" to ".jpg"
        fullFileName = os.path.join(myPath, fileName)
        newFileName = fullFileName[0:len(fullFileName)-4] + ".jpg"
        os.rename(fullFileName, newFileName)
```

Since `endswith()` is case-sensitive, we had to convert `fileName` 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 `newFileName = fullFileName[0:len(fullFileName)-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:[17]

```
import glob
import os
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files/images"
possibleFiles = os.path.join(myPath, "*.gif")
for fileName in glob.glob(possibleFiles):
    print 'Converting "{}" to JPG...'.format(fileName)
    fullFileName = os.path.join(myPath, fileName)
    newFileName = fullFileName[0:len(fullFileName)-4] + ".jpg"
    os.rename(fullFileName, newFileName)
```

By providing a string with the full file path and a "`*`", 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:

17 Instead of using indexing to get the file extension, we also could have used the `os.path.splitext()` function, which takes a file path and returns a tuple where the first item in the tuple is everything before the extension and the second item in the tuple is the file extension.

```
import glob
import os
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files/images"
possibleFiles = os.path.join(myPath, "*/*.png")
for fileName in glob.glob(possibleFiles):
    print fileName
```

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.

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:

```
myPath = "C:\\Real Python\\Course materials\\Chapter 7\\Practice files\\images"
```

Although this would then *display* 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`.[18]

Keep in mind that `listdir()` returns a list of all files *and* folders in a given folder.

18 These pattern matching rules follow a Unix shell standard for path expansions; they are *not* the same as *regular expressions*, which are a much more involved pattern matching system that will be introduced in the chapter on web scraping.

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:

```
import os
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files/images"
filesAndFolders = os.listdir(myPath)
for folderName in filesAndFolders:
    fullPath = os.path.join(myPath, folderName)
    if os.path.isdir(fullPath):
        os.rename(fullPath, fullPath + " 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 `folderName` 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
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files/images"
for currentFolder, subfolders, fileNames in os.walk(myPath):
    for fileName in fileNames:
        print os.path.join(currentFolder, fileName)
```

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 `(currentFolder, subfolders, fileNames)` to loop over, where `currentFolder` 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 `fileNames` and joining them to `currentFolder` 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:

- Display the full paths of all of the files and folders in the `images` folder by using `os.listdir()`
- Display the full paths of any PNG files in the `images` folder by using `glob.glob()`
- 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
- Make sure that your last script worked by using `os.path.exists()` to check that the files "`png file - not a gif.jpg`" and "`/additional files/one last image.jpg`" now exists (by providing `os.path.exists()` with the full path to each of these files)

Assignment 7.2: Use pattern matching to delete files

? Write a script "remove_files.py" that will look in the chapter 7 practices 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.

7.3) 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 7 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.

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 in chapter 9.

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
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files"
with open(os.path.join(myPath, "wonka.csv"), "rb") as myFile:
    myFileReader = csv.reader(myFile)
    for row in myFileReader:
        print row
```

We opened a file just as we've done before, but this time we chose "`rb`" mode, which stands for ***read binary***. The additional "binary" part is important here, because CSV files (despite their appearance) aren't saved in the same way as raw text files. (Specifically, on Windows we will end up saving extra newline characters after every line if we only specify "`r`" mode.)

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 `myFileReader.next()` 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
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files"
with open(os.path.join(myPath, "wonka.csv"), "rb") as myFile:
    myFileReader = csv.reader(myFile)
    myFileReader.next()
    for firstName, lastName, reward in myFileReader:
        print "{} {} got: {}".format(firstName, lastName, reward)
```

After skipping the first header row, we assigned the three values in each row to the three separate strings `firstName`, `lastName` and `reward`, which we then used inside of the `for` loop, generating this output:

```
>>>
Charlie Bucket got: golden ticket, chocolate factory
Veruca Salt got: squirrel revolution
Violet Beauregarde got: fruit chew
>>>
```

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
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files"
with open(os.path.join(myPath, "tabbed wonka.csv"), "rb") as myFile:
    myFileReader = csv.reader(myFile, delimiter="\t")
    myFileReader.next()
    for row in myFileReader:
        print row
```

Here we used the special character "`\t`" to mean the "tab" character and assigned it to the argument `delimiter` when we created `myFileReader`.

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
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files/Output"
with open(os.path.join(myPath, "movies.csv"), "wb") as myFile:
    myFileWriter = csv.writer(myFile)
    myFileWriter.writerow(["Movie", "Rating"])
    myFileWriter.writerow(["Rebel Without a Cause", "3"])
    myFileWriter.writerow(["Monty Python's Life of Brian", "5"])
    myFileWriter.writerow(["Santa Claus Conquers the Martians", "0"])
```

We opened a new file in "`wb`" mode this time so that we could ***write binary*** 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
myPath = "C:/Real Python/Course materials/Chapter 7/Practice files/Output"
myRatings = [ ["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(myPath, "movies.csv"), "wb") as myFile:
```

```
myFileWriter = csv.writer(myFile)
myFileWriter.writerows(myRatings)
```

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.

Review exercises:

- Write a script that reads in the data from the CSV file "`pastimes.csv`" located in the chapter 7 practice files folder, skipping over the header row
- Display each row of data (except for the header row) as a list of strings
- 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()`
- 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
- 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 7.3: 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 7 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
```

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 24,000 of these "extra" packages 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
```

However, eventually you will come across a Python package that is not as simple to install for your particular operating system. For this reason, I recommend that you install a tool called `easy_install` now, *before* you want to use a difficult-to-install package. Ironically, `easy_install` can be difficult to install. But once it's set up properly, your future with Python will be much more hassle-free.

Before installing anything, you first need to make sure that your operating system can find Python; this way, you'll be able to run Python scripts outside of IDLE.

Windows: On your desktop, right-click on **My Computer** and click **Properties**.[19] Click on the **Advanced** tab, then click on **Environmental Variables**. Click to highlight the **Path** variable, then click **Edit**. You can now edit the system directories listed; each of these

19 If you don't have a **My Computer** shortcut, go to **Start** → **Control Panel** → **System** → **Advanced system settings**.

folders is separated from the previous folder by a comma. DO NOT delete any of the current paths. Go to the very end of this string and (assuming you installed Python into the default directory) add the two Python paths as follows, starting with a semicolon to separate the new paths from the previous entries in your PATH variable: `;C:\Python27\;C:\Python27\Scripts`
OS X: You should be fine as long as you've separately downloaded the correct version of Python 2.7.3 and are not relying on the built-in OS X version of Python. Type `python` into your Terminal to make sure it is recognized and loads as version 2.7.3 (see below).
Linux: You should be fine. Type `python` into your Terminal to make sure it is recognized and loads as version 2.7.3 (see below).

Double-check that Python is recognized by your system by opening a new command prompt[20] (Windows) or Terminal (OS X and Linux) window and simply typing the name `python` as a command. Once you hit enter, Python should load just the same way as if you were in the IDLE interactive window, giving you a "`>>>` " prompt as usual. You can exit out of Python in this window by typing `exit()`.

Now that your operating system knows how to access Python, you can get `easy_install` installed and set up:

Windows: If you are running 32-bit Windows, download and run the installer found here to set up `easy_install` automatically. If you have 64-bit Windows, download and run the script ez_setup.py; you may need to copy the code into a new script manually.
OS X: If you have XCode installed, you may already have a version of `easy_install` as well. If not, download the EGG file available here into your user home directory, then install it by typing the following command into your Terminal: `sh setuptools-0.6c11-py2.7.egg`
Debian/Linux: Use the following command to install `easy_install`: `sudo apt-get install python-setuptools`

20 Click `Start` (the Windows button) and type `cmd` to find and start the command prompt.

RPM-based operating systems should download and install the RPM file here.

You can now (generally) install most packages by typing "`easy_install package_name`" at the command prompt (for Windows) or "`sudo easy_install package_name`" at the Terminal (for non-Windows).

Since `easy_install` isn't always that easy to install, there *is* an alternative.

! You don't need to go through the following process in detail yet, but refer back to this section as needed if you have difficulty installing a package:

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:

- Download the `.zip` file for the package and unzip it into a folder in your user directory (i.e., "`C:\Users\yourname`")
- Double-check that there is a script named `setup.py` in the package folder
- 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
```

- To install the package, enter the command:

```
python setup.py install
```

Non-Windows:

- Download the `.tar.gz` file for the package and decompress it into a folder in your users directory (i.e., "`/home/yourname`")
- Double-check that there is a script named `setup.py` in the package folder
- 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
```

– 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.

8) Interact with PDF files

8.1) Read and write PDFs

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 pyPdf and can be found here. You will need to download and install this package before continuing with the chapter.

Windows: Download and run the automated installer (pyPdf-1.13.win32.exe).
OS X: if you have `easy_install` installed, you can type the following command into your Terminal to install pyPdf: `sudo easy_install pypdf` Otherwise, you will need to download and unzip the `.tar.gz` file and install the module using the `setup.py` script as explained in the section on installing packages.
Debian/Linux: Just type the command: `sudo apt-get install python-pypdf`

The pyPdf 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 pyPdf import PdfFileReader
path = "C:/Real Python/Course materials/Chapter 8/Practice files"
inputFileName = os.path.join(path, "Pride and Prejudice.pdf")
inputFile = PdfFileReader(file(inputFileName, "rb"))

print "Number of pages:", inputFile.getNumPages()
print "Title:", inputFile.getDocumentInfo().title
```

We created a `PdfFileReader` objected named `inputFile` 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 and have to interpret 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:[21]

```
>>> print inputFile.getDocumentInfo()
{'/CreationDate': u'D:20110812174208', '/Author': u'Chuck', '/Producer':
u'Microsoft\xae Office Word 2007', '/Creator': u'Microsoft\xae 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, pyPdf has made the process of parsing out text somewhat easier, and we can use the `extractText()` method on each page:

```
>>> print inputFile.getPage(0).extractText()
  The Project Gutenberg EBook of Pride and Prejudice, by Jane Austen  This
eBook is 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 terms of the Project Gutenberg License included with this eBook or online
```

21 Sometimes this information (called *meta-data*) gets filled in automatically or in unexpected ways; for instance, "`Chuck`" (most likely a helpful but unwitting volunteer) appears to be the author of this file…

```
at www.gutenberg.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 pyPdf. 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 mobile device):

```
import os
from pyPdf import PdfFileReader

path = "C:/Real Python/Course materials/Chapter 8/Practice files"
inputFileName = os.path.join(path, "Pride and Prejudice.pdf")
inputFile = PdfFileReader(file(inputFileName, "rb"))
outputFileName = os.path.join(path, "Output/Pride and Prejudice.txt")
outputFile = open(outputFileName, "w")

title = inputFile.getDocumentInfo().title # get the file title
totalPages = inputFile.getNumPages() # get the total page count

outputFile.write(title + "\n")
outputFile.write("Number of pages: {}\n\n".format(totalPages))

for pageNum in range(0, totalPages):
    text = inputFile.getPage(pageNum).extractText()
    text = text.replace("  ", "\n")
    outputFile.write(text)
outputFile.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.

You may find that a PDF document includes unusual characters that cannot be written into a plain text file - for instance, a trademark symbol. These characters are not in the ASCII character set, meaning that they can't be represented using any of the 128 standard computer characters. Because of this, your code will not be able to write out a raw text file in "`w`" mode. Usually the way to get around this is by using the `encode()` method, like so:

```
text = text.encode("utf-8")
```

If we have a string `text` that has unusual characters in it, this line will allow us to change how each character is represented (using UTF-8 encoding) so that we can now store these symbols in a raw text file. These unusual characters might not appear the same way as in the original file, since the text file has a much more limited set of characters available, but if you do not change the encoding then you will not be able to output the text at all.

(If you *really* want to get a handle on text encoding and what's really happening, take a look at this talk.)

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 pyPdf import PdfFileReader, PdfFileWriter

path = "C:/Real Python/Course materials/Chapter 8/Practice files"
inputFileName = os.path.join(path, "Pride and Prejudice.pdf")
inputFile = PdfFileReader(file(inputFileName, "rb"))
outputPDF = PdfFileWriter()

for pageNum in range(1, 4):
    outputPDF.addPage(inputFile.getPage(pageNum))

outputFileName = os.path.join(path, "Output/portion.pdf")
outputFile = file(outputFileName, "wb")
outputPDF.write(outputFile)
outputFile.close()
```

We imported both `PdfFileReader` and `PdfFileWriter` from pyPdf 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 `outputPDF` *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 `outputFile` as usual and then call `outputPDF.write(outputFile)` in order to write the PDF contents *into* this file.

Review exercises:

- Write a script that opens the file named "The Whistling Gypsy.pdf" from the Chapter 8 practice files, then displays the title, author, and total number of pages in the file
- Extract the full contents of "The Whistling Gypsy.pdf" into a .TXT file; you will need to encode the text as UTF-8 before you can output it
- Save a new version of "The Whistling Gypsy.pdf" that does not include the cover page into the Output folder

8.2) 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 in Adobe Acrobat isn't a practical or feasible solution, we can automate any of these tasks using pyPdf.

Let's start with a surprisingly common problem: rotated PDF pages. Go ahead and open up the file "`ugly.pdf`" in the "`Chapter 8/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:[22]

```
import os
from pyPdf import PdfFileReader, PdfFileWriter
```

22 Likewise, there is a `rotateCounterClockwise()` method for the unimaginative. Unfortunately for the imaginative, you can only rotate by a multiple of ninety degrees.

```
path = "C:/Real Python/Course materials/Chapter 8/Practice files"
inputFileName = os.path.join(path, "ugly.pdf")
inputFile = PdfFileReader(file(inputFileName, "rb"))
outputPDF = PdfFileWriter()

for pageNum in range(0, inputFile.getNumPages()):
    page = inputFile.getPage(pageNum)
    if pageNum % 2 == 0:
        page.rotateClockwise(90)
    outputPDF.addPage(page)

outputFileName = os.path.join(path, "Output/The Conformed Duckling.pdf")
outputFile = file(outputFileName, "wb")
outputPDF.write(outputFile)
outputFile.close()
```

Another useful feature of pyPdf 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 8 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 pyPdf import PdfFileReader
>>> inputFile = PdfFileReader(file("C:/Real Python/Course materials/Chapter
8/Practice files/half and half.pdf", "rb"))
>>> page = inputFile.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 pyPdf import PdfFileReader, PdfFileWriter
path = "C:/Real Python/Course materials/Chapter 8/Practice files"
inputFileName = os.path.join(path, "half and half.pdf")
inputFile = PdfFileReader(file(inputFileName, "rb"))
outputPDF = PdfFileWriter()

for pageNum in range(0, inputFile.getNumPages()):
    pageLeft = inputFile.getPage(pageNum)
    pageRight = copy.copy(pageLeft)
    upperRight = pageLeft.mediaBox.upperRight # get original page corner
    # crop and add left-side page
    pageLeft.mediaBox.upperRight = (upperRight[0]/2, upperRight[1])
```

```
    outputPDF.addPage(pageLeft)
    # crop and add right-side page
    pageRight.mediaBox.upperLeft = (upperRight[0]/2, upperRight[1])
    outputPDF.addPage(pageRight)

outputFileName = os.path.join(path, "Output/The Little Mermaid.pdf")
outputFile = file(outputFileName, "wb")
outputPDF.write(outputFile)
outputFile.close()
```

! 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 `upperLeft` 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 pyPdf import PdfFileReader, PdfFileWriter
path = "C:/Real Python/Course materials/Chapter 8/Practice files"
inputFileName = os.path.join(path, "The Emperor.pdf")
inputFile = PdfFileReader(file(inputFileName, "rb"))
outputPDF = PdfFileWriter()

watermarkFileName = os.path.join(path, "top secret.pdf")
watermarkFile = PdfFileReader(file(watermarkFileName, "rb"))

for pageNum in range(0, inputFile.getNumPages()):
    page = inputFile.getPage(pageNum)
    page.mergePage(watermarkFile.getPage(0)) # add watermark image
    outputPDF.addPage(page)

outputPDF.encrypt("good2Bking") # add a password to the PDF file
outputFileName = os.path.join(path, "Output/New Suit.pdf")
outputFile = file(outputFileName, "wb")
outputPDF.write(outputFile)
outputFile.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.

Although pyPdf 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 researching the ReportLab toolkit, which is also free and open-source. Another popular choice for manipulation of existing PDF files in Python is PDFMiner, which offers slightly different functionality from pyPdf.

There is also a PyPDF2 in the works that aims to handle more difficult PDF files and make some PDF manipulation tasks even easier than in pyPdf. However, as of this writing (September 2012) there is not yet any documentation available on how to use it.

Review exercises:

- Write a script that opens the file named "Walrus.pdf" from the Chapter 8 practice files; you will need to decrypt the file using the password `IamtheWalrus`
- Rotate every page in this input file counter-clockwise by 90 degrees
- 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 8.2: Add a cover sheet to a PDF file

Write a script "cover_the_emperor.py" that appends the chapter 8 practice file named "`The Emperor.pdf`" to the end of the chapter 8 practice file named "`Emperor cover sheet.pdf`" and outputs the full resulting PDF to the file "`The Covered Emperor.pdf`" in the chapter 8 practice files Output folder.

9) SQL database connections

9.1) Communicate with databases using SQLite

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.

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 this:

```
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.

! 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.

! 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:

```
peopleValues = (
```

```
            ('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(?, ?, ?)", peopleValues)
```

Here, the question marks act as place-holders for the tuples in `peopleValues`; 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
firstName = raw_input("Enter your first name: ")
lastName = raw_input("Enter your last name: ")
age = int(raw_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('" + firstName + "','" + lastName +
"'," + 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 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
firstName = raw_input("Enter your first name: ")
lastName = raw_input("Enter your last name: ")
age = int(raw_input("Enter your age: "))
personData = (firstName, lastName, 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(?, ?, ?)", personData)
```

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

peopleValues = (
           ('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(?, ?, ?)", peopleValues)
    # 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:

```
>>>
(u'Ron', u'Obvious')
(u'Luigi', u'Vercotti')
>>>
```

The "`u`" before each string stands for unicode (as opposed to ASCII) and basically means that the string *might* contain complicated characters that can't be represented by the usual basic set of characters we normally see in English text.

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 == 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 == None
```

This comparison will return `True` if `objectName` exists but is empty and `False` if `objectName` holds any value.

Review exercises:

- Create a database table in RAM named `Roster` that includes the fields `'Name'`, `'Species'` and `'IQ'`
- Populate your new table with the following values:

```
Jean-Baptiste Zorg,  Human,  122
Korben Dallas,  Meat Popsicle,  100
Ak'not,  Mangalore,  -5
```

- Update the `Species` of `Korben Dallas` to be `Human`
- Display the names and IQs of everyone in the table who is classified as `Human`

9.2) 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:

- The pyodbc module allows connections to most types of databases.
- 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.
- 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 every different flavor 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.

10) Interacting with the web

10.1) Scrape and parse text from websites

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.

! 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:

- **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**
- **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**

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 (**H**yper**T**ext **M**arkup **L**anguage), 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 little while to read through the first dozen chapters of this brief HTML introduction. 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.

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:

```
import urllib2
myAddress = "http://RealPython.com/practice/aphrodite.html"
htmlPage = urllib2.urlopen(myAddress)
htmlText = htmlPage.read()
print htmlText
```

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:

```
import urllib2
myAddress = "http://RealPython.com/practice/aphrodite.html"
htmlPage = urllib2.urlopen(myAddress)
htmlText = htmlPage.read()
startTag = "<title>"
endTag = "</title>"
startIndex = htmlText.find(startTag) + len(startTag)
endIndex = htmlText.find(endTag)
print htmlText[startIndex:endIndex]
```

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 `myAddress = "http://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.

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. Although a full overview of regular expressions is outside the scope of this book, I'll cover just a few couple examples to get started.

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 in most instances is all we care to find. For instance:

```
>>> matchResults = re.search("ab*c", "ABC", re.IGNORECASE)
>>> print matchResults.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, then followed by the string. For instance:

```
>>> myString = "Everything is <replaced> if it's in <tags>."
>>> myString = re.sub("<.*>", "ELEPHANTS", myString)
>>> print myString
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[23], 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:

```
>>> myString = "Everything is <replaced> if it's in <tags>."
>>> myString = re.sub("<.*?>", "ELEPHANTS", myString)
>>> print myString
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:[24]

```
import re
import urllib2
myAddress = "http://RealPython.com/practice/dionysus.html"
htmlPage = urllib2.urlopen(myAddress)
htmlText = htmlPage.read()
```

23 Yes, that's a legitimate term.

24 Actually, this code would *also* remove any "<>" tags that appear inside of the title's text, but we'll assume here that we would want to remove such tags in the unlikely event that they do appear there.

```
matchResults = re.search("<title .*?>.*</title .*?>", htmlText,
re.IGNORECASE)
title = matchResults.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:

`<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
`.*`	Any characters can appear in between the `<title>` tags
`</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, although you're encouraged to take some time to study the very thorough Python Regular Expression HOWTO document.

! Fair warning: 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.

Review exercises:

- Write a script that grabs the full HTML from the page dionysus.html
- 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)

– 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

10.2) 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 for **Windows** or **OS X**, if you have `easy_install` set up then you can type the following command into your command prompt or Terminal:

```
easy_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
import urllib2
myAddress = "http://RealPython.com/practice/dionysus.html"
htmlPage = urllib2.urlopen(myAddress)
htmlText = htmlPage.read()
mySoup = BeautifulSoup(htmlText)
```

From here, we can parse data out of `mySoup` 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 mySoup.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 `BeautifulSoup` 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 mySoup.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 `BeautifulSoup` 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 = mySoup.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="http://RealPython.com" target="_blank">` would have *two* attributes, a "`href`" attribute that is assigned the value "`http://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 mySoup.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 mySoup.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:

```
>>> mySoup.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 still used frequently today, the code is no longer being maintained and updated by its creator. A similar toolkit, 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.

! 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:

- Write a script that grabs the full HTML from the page profiles.html
- 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
- 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

10.3) Interact with HTML forms

We usually retrieve information from the Internet by sending requests for webpages. A module like urllib2 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 `mechanize`.

Essentially, mechanize is an alternative to `urllib2` that can do all of the same things but has much more added functionality that will allow us to *talk back* to webpages.

Windows: If you have `easy_install` set up, you can type "`easy_install mechanize`" into your command prompt. Otherwise, download the `.zip` file, decompress it, and install the package by running the `setup.py` script from the command line as described in the section on installing packages.
OS X: If you have `easy_install` set up, you can type "`sudo easy_install mechanize`" into your Terminal. 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.
Debian/Linux: As usual: `sudo apt-get install python-mechanize`

You may need to close and restart your IDLE session for mechanize to load and be recognized after it's been installed.

Getting mechanize to create a new Browser object and use it to open a webpage is as easy as saying:

```
import mechanize
myBrowser = mechanize.Browser()
myBrowser.open("http://RealPython.com/practice/aphrodite.html")
```

We now have various information that the website returned to us stored in our mechanize browser as a "response" which we can return by calling the `response()` method. This response also has various methods that help us piece out information returned from the website:

```
>>> print myBrowser.response().geturl()
http://www.RealPython.com/practice/aphrodite.html
>>> print myBrowser.response().get_data()
<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>
>>>
```

Here we used the `geturl()` method to return the URL (i.e., the full address) of the webpage and the `get_data()` method to return the actual HTML code in the same way that we used `read()` with `urllib2`. We could then use Beautiful Soup or regular expressions to parse out the information we want from the string returned by the `get_data()` method.

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 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 response provided by login.php:

```
import mechanize
myBrowser = mechanize.Browser()
myBrowser.open("http://RealPython.com/practice/login.php")
myResponse = myBrowser.response()
print myResponse.get_data()
```

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.[25] In this case, the PHP 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.html 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 mechanize
myBrowser = mechanize.Browser()
myBrowser.open("http://RealPython.com/practice/login.php")

# select the form and fill in its fields
myBrowser.select_form("login")
myBrowser["user"] = "zeus"
myBrowser["pwd"] = "ThunderDude"

myResponse = myBrowser.submit() # submit form

print myResponse.geturl() # make sure we were redirected
```

We used the browser's `select_form()` method to select the form, passing in the name "`login`" that we discovered by reading the HTML of the page. Once we had that form selected in the browser, we could then assign values to the different fields the same way we would assign dictionary values; we passed the value "`zeus`" to the field named

25 PHP stands for **P**HP: **H**ypertext **P**repocessor… Yeah, I know, that's super helpful. PHP *used to* stand for **P**ersonal **H**ome **P**age.

"user" and the value "ThunderDude" to the field named "pwd"; once again, we knew what to call these fields by reading the "name" attribute assigned within each <input> HTML tag.

Once we filled in the form, we could submit this information to the webpage by using the browser's submit() method, which returns a response in the same way that calling open() on a webpage would do. 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 login.php, but we see that we were successfully redirected to profiles.html as planned.

! 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.html page, we could say:

```
>>> for link in myBrowser.links():
      print "Address:", link.absolute_url
      print "Text:", link.text

Address: http://RealPython.com/practice/aphrodite.html
Text: Aphrodite
Address: http://RealPython.com/practice/poseidon.html
Text: Poseidon
Address: http://RealPython.com/practice/dionysus.html
Text: Dionysus
>>>
```

Each Link object has a number of attributes, including an absolute_url attribute that represents the address of the webpage (i.e., the "href" value) and a text

attribute that represents the actual text that appears as a link on the webpage.

The mechanize browser provides many other methods to offer us the full functionality of a standard web browser. For instance, the browser's `back()` method simply takes us back one page, similar to hitting the "Back" button in an ordinary browser. We can also "click" on links in a page to follow them using the browser's `follow_link()` method.

Let's follow each of the links on the profiles.html page using the browser's `follow_link()` and `back()` methods, displaying the title of each webpage we visit by using the browser's `title()` method:

```
import mechanize
myBrowser = mechanize.Browser()
myBrowser.open("http://RealPython.com/practice/profiles.html")

for nextLink in myBrowser.links(): # follow each link on profiles.html
    myBrowser.follow_link(nextLink)
    print "Page title:", myBrowser.title() # display current page title
    myBrowser.back() # return to profiles.html

print "Page title:", myBrowser.title() # back to profiles.html
```

By using the `follow_link()` method to open links, we avoid the trouble of having to add the rest of the URL path to each link. For instance, the first link on the page simply links to a page named "`aphrodite.html`" without including the rest of the URL; this is a *relative* URL, and mechanize wouldn't know how to load this webpage if we passed it to the `open()` method. To `open()` this address, we would have to create an *absolute* URL by adding the rest of the web address, i.e., "`http://RealPython.com/practice/aphrodite.html`".

The `follow_link()` method takes care of this problem for us since the `Link` objects all store absolute URL paths. Of course, in this case it still may have been more sensible to `open()` each of these links directly since this wouldn't require loading as many pages. However, sometimes we need the ability to move back and forth between pages.

But how did our code do? The output appears as follows:

```
>>>
Page title: Profile: Aphrodite
Page title: Profile: Poseidon
```

```
Page title: Profile: Dionysus</title /> </head> <body bgcolor="yellow">
<center> <br><br> <img src="dionysus.jpg"> <h2>Name: Dionysus</h2> <img
src="grapes.png"><br><br> Hometown: Mount Olympus <br><br> Favorite animal:
Leopard <br> <br> Favorite Color: Wine </center> </body> </html>
Page title: All Profiles
>>>
```

Unfortunately, mechanize's HTML parser couldn't find the closing `title` tag for Dionysus because of the errant forward slash at the end, and we ended up gathering the rest of the page. When one parsing method fails us (in this case, the mechanize browser's `title()` method) because of poorly written HTML, the best course of action is to turn to another parsing method - in this case, creating a `BeautifulSoup` object and extracting the webpage's title from it. For instance:

```
import mechanize
from bs4 import BeautifulSoup

myBrowser = mechanize.Browser()
htmlPage = myBrowser.open("http://www.RealPython.com/practice/dionysus.html")
htmlText = htmlPage.get_data()
mySoup = BeautifulSoup(htmlText)
print mySoup.title.string
```

Notice that we didn't have to revisit using `urllib2` since mechanize already provides the functionality of opening webpages and retrieving their HTML for us.

Review exercises:

- Use mechanize to provide the correct username "`zeus`" and password "`ThunderDude`" to the login page submission form located at: http://RealPython.com/practice/login.php
- Using Beautiful Soup, display the title of the current page to determine that you have been redirected to profiles.html
- Use mechanize to return to login.php by going "back" to the previous page
- 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

10.4) 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 `reload()` method of the mechanize 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 15.72, 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">15.72</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.[26] 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 mechanize and Beautiful Soup to find and display the current price, like so:

```
import mechanize
from bs4 import BeautifulSoup

myBrowser = mechanize.Browser()
htmlPage = myBrowser.open("http://finance.yahoo.com/q?s=yhoo")
htmlText = htmlPage.get_data()
mySoup = BeautifulSoup(htmlText)

# return a list of all the <span> tags where id="yfs_l84_yhoo"
myTags = mySoup.find_all("span", id="yfs_l84_yhoo")
```

26 Note that there's a lower-case "L" before the 84 in the tag, not the number 1.

```
# take the BeautifulSoup string out of the first (and only) <span> tag
myPrice = myTags[0].string
print "The current price of YHOO is: {}".format(myPrice)
```

We needed to use `format()` to print the price because `myPrice` is not just an ordinary string; it's actually a more complex `BeautifulSoup` string that has other information associated with it, and Beautiful Soup would try to display this other information as well if we had only said `print myPrice`.

Now, in order to *repeatedly* get the newest stock quote available, we'll need to create a loop that uses the `reload()` method. 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.[27] 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:

```
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.

27 There is nothing in the general Yahoo! Terms of Service to suggest that using automated tools to scrape pages isn't allowed, and a number of third-party products have been known to scrape Yahoo! Finance data for years, so we should be in the clear as far as using mechanize to access these webpages.

Using the `sleep()` method, we can now repeatedly obtain real-time stock quotes:

```
from time import sleep
import mechanize
from bs4 import BeautifulSoup

# create a Browser object
myBrowser = mechanize.Browser()

# obtain 1 stock quote per minute for the next 3 minutes
for i in range(0, 3):
    htmlPage = myBrowser.open("http://finance.yahoo.com/q?s=yhoo")
    htmlText = htmlPage.get_data()

    mySoup = BeautifulSoup(htmlText)
    myTags = mySoup.find_all("span", id="yfs_l84_yhoo")

    myPrice = myTags[0].string
    print "The current price of YHOO is: {}".format(myPrice)

    if i<2: # wait a minute if this isn't the last request
        sleep(60)
```

Review exercises:

- 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

11) Scientific computing and graphing

11.1) Use NumPy for matrix manipulation

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.

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, download the latest version here for Windows or here for Mac, then run the automated installer.[28] 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
```

28 OS X users may encounter problems using NumPy if XCode is already installed. The best work-around so far appears to be uninstalling XCode or using IPython, which is mentioned at the end of this chapter.

```
>>>
```

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]
 [4 5 6]
 [7 8 9]]
>>> 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.

! 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.

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:

```
>>> print matrix*2
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
>>>
```

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:

```
>>> secondMatrix = array([[5,4,3], [7,6,5], [9,8,7]])
>>> print secondMatrix - 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 secondMatrix * 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(secondMatrix, 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, secondMatrix]) # add secondMatrix below matrix
[[1 2 3]
 [4 5 6]
 [7 8 9]
 [5 4 3]
 [7 6 5]
 [9 8 7]]
>>> print hstack([matrix, secondMatrix]) # add secondMatrix 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:

```
>>> array3d = array([[[1,2],[3,4]], [[5,6],[7,8]], [[9,10],[11,12]]])
>>> print array3d
[[[ 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()` an `arange()`:

```
>>> array3d = arange(1,13)
>>> array3d = array3d.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 but incomplete Tentative NumPy 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 draft reference guide that covers most major topics. SciPy is available for download here.

Review exercises:

- Create a 3 x 3 NumPy array named `firstMatrix` that includes the numbers 3 through 11 by using `arange()` and `reshape()`
- Display the minimum, maximum and mean of all entries in `firstMatrix`
- Square every entry in `firstMatrix` using the `**` operator, and save the results in an array named `secondMatrix`
- Use `vstack()` to stack `firstMatrix` on top of `secondMatrix` and save the results in an array named `thirdMatrix`
- Use `dot()` to calculate the dot product of `thirdMatrix` by `firstMatrix`
- Reshape `thirdMatrix` into an array of dimensions 3 x 3 x 2

11.2) 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 out pictures or documents.

You will first need to download and install both NumPy (see the previous section) and matplotlib. Windows and OS X users can download an automated matplotlib installer

here. Debian/Linux users can get matplotlib by typing the command:[29]

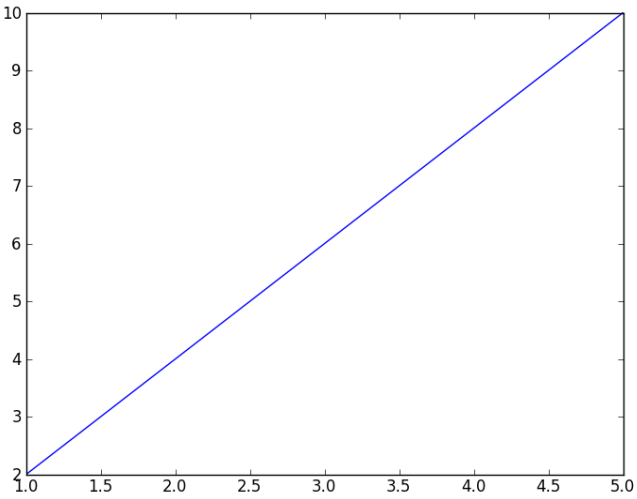
```
sudo apt-get install python-matplotlib
```

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()
```



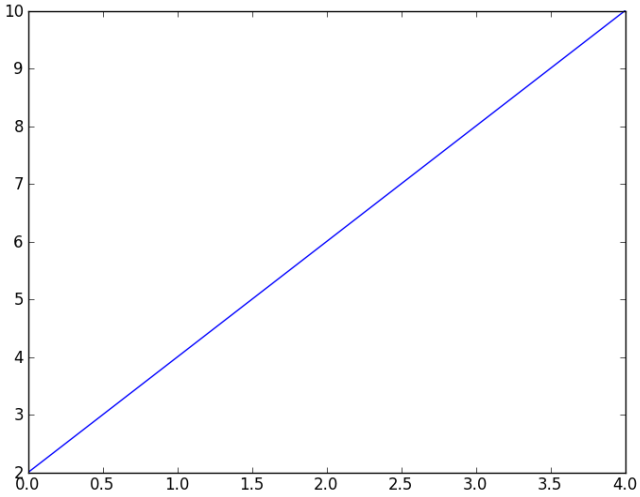
We created a plot, supplying a list of *x*-coordinate points and a matching list of *y*-coordinate points. When we call `plt.show()`, a new window appears, displaying a graph of our five (x,y) points. Our interactive window is essentially locked while this plot window is open, and we can only end the script and return to a new prompt in the interactive window once we've closed the graph.

29 Some Linux users might separately need to install code to support the graphical interface (see section 12.2) by typing the command "`sudo apt-get install python-tk`".

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()
```

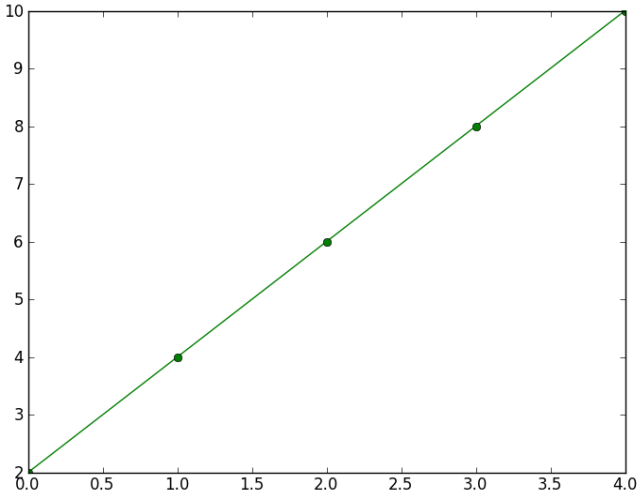


However, this isn't *exactly* the same graph because, as always, Python begins counting at 0 instead of 1, which is what we now see for the horizontal axis values as well.

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()
```



For reference, the full list of possible formatting combinations can be found here.

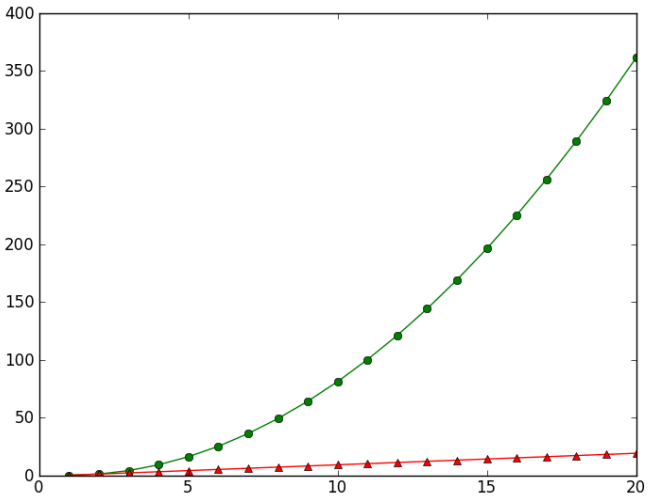
Plotting wouldn't be very convenient if we had to organize everything into lists ourselves first; matplotlib works with NumPy to accept arrays as well. (Even if you *can* get away with using a basic list, it's a good idea to stick with arrays in case you later discover that you *do* need to modify the 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:

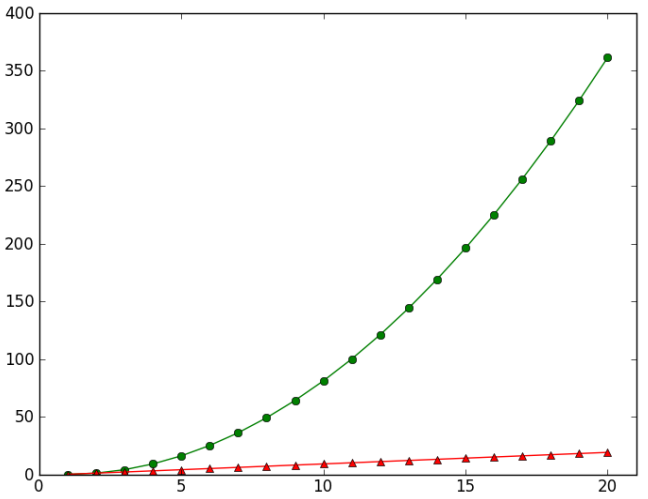
```
from matplotlib import pyplot as plt
from numpy import arange
xPoints = arange(1,21)
baseline = arange(0,20)
plt.plot(xPoints, baseline**2, "g-o", xPoints, baseline, "r-^")
plt.show()
```



We plotted two sets of points; the green dots plot the values in `xPoints` with the square of the `baseline` y values, while the red triangles just plot the values in `xPoints` with the original `baseline` values.

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:

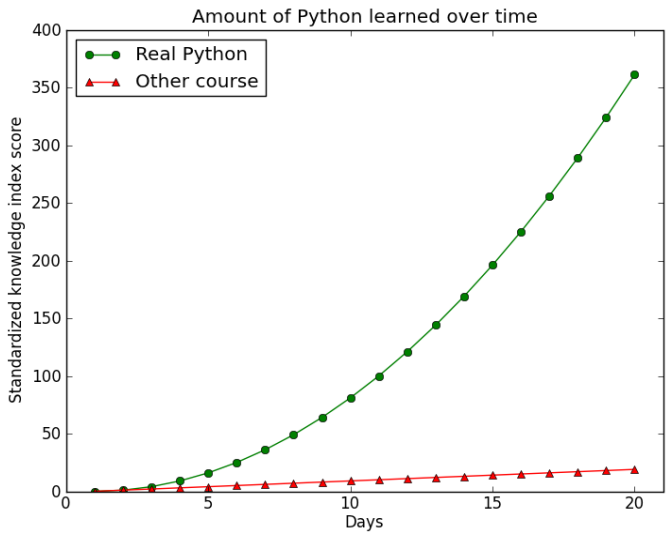
```
from matplotlib import pyplot as plt
from numpy import arange
xPoints = arange(1,21)
baseline = arange(0,20)
plt.plot(xPoints, baseline**2, "g-o", xPoints, baseline, "r-^")
plt.axis([0, 21, 0, 400])
plt.show()
```



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 title, a legend and some titles for the axes:

```
from matplotlib import pyplot as plt
from numpy import arange
xPoints = arange(1,21)
baseline = arange(0,20)
plt.plot(xPoints, baseline**2, "g-o", xPoints, 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()
```

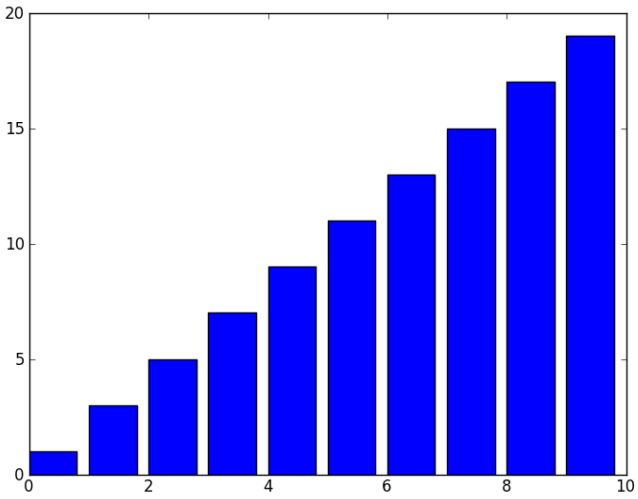


There are many more complicated ways to create a legend, but the simplest is to supply a tuple that lists the labels to be applied to all the plotted points in order. We specified `loc=2` in the legend because we want the legend to appear in the top left corner, while the default is for the legend to 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.

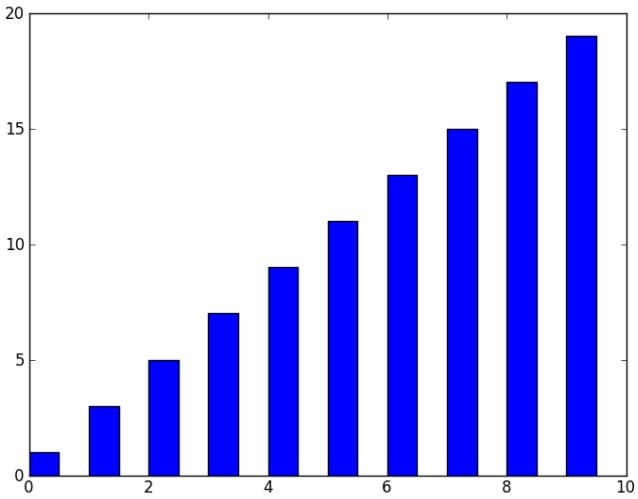
Another frequently used type of plot in basic data visualization is a bar chart. Let's start with a very simple example of a bar chart, which uses the `bar()` plotting function:

```
from matplotlib import pyplot as plt
from numpy import arange
plt.bar(arange(0,10), arange(1,21,2))
plt.show()
```



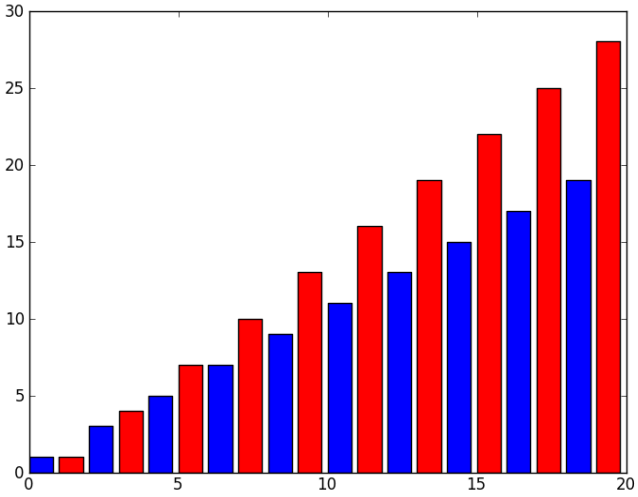
The first argument takes a list or an array of the x-axis locations for each bar's left edge; in this case, we placed the left sides of our bars along the numbers from 0 through 9. The second argument of `bar()` is a list or an array of the ordered bar values; here we supplied the odd numbers 1 through 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 an `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 our bars are located:

```
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()
```

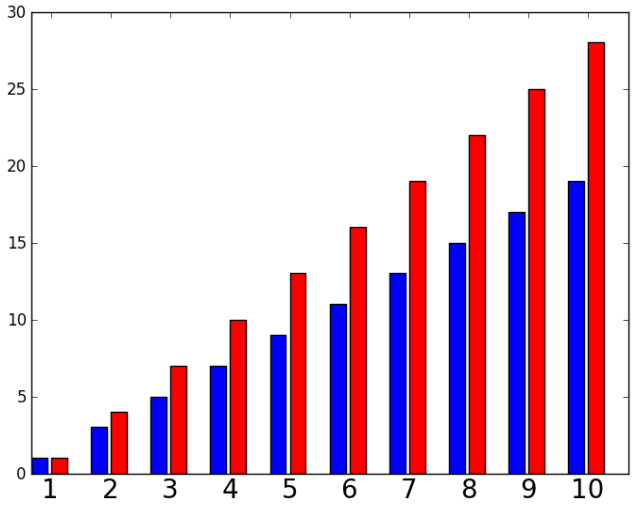


For the x-axis of the first set of bars, we supplied the even numbers 0 through 18 by multiplying every number in our `arange()` by 2. This allows us to leave some space for

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()
```

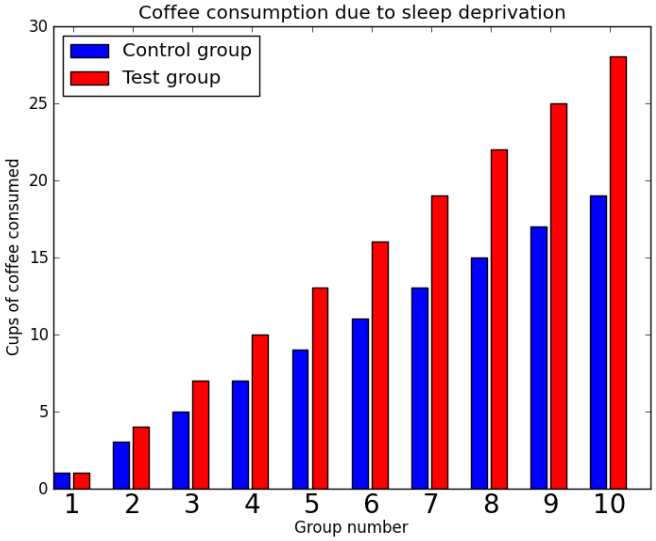


Since we wanted to show the x-axis labels in between each of our pairs of bars, we specified the left side of the second (red) set of bars as the position to show the label. We then gave the numbers 1 through 10 as the actual labels to display; we could just as easily have used a list of strings as labels 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:

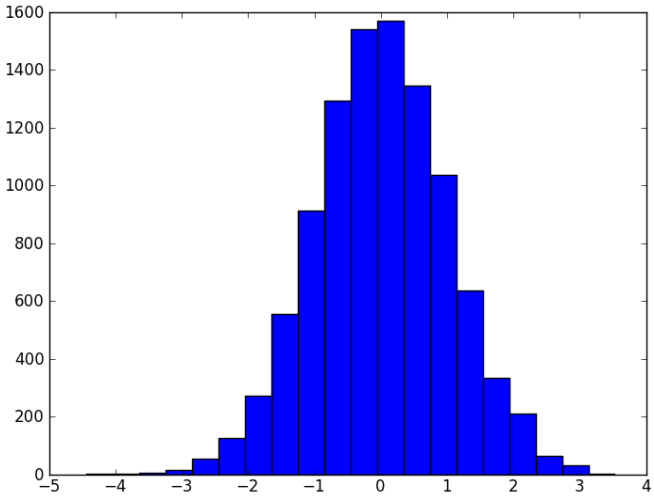
```
from matplotlib import pyplot as plt
from numpy import arange
plt.bar(arange(0,10)*3, arange(1,21,2))
plt.bar(arange(0,10)*3 + 1, arange(1,31,3), color="red")
plt.xticks(arange(0,10)*3 + 1, arange(1,11), fontsize=20)
plt.title("Coffee consumption due to sleep deprivation")
plt.xlabel("Group number")
```

```
plt.ylabel("Cups of coffee consumed")
plt.legend(("Control group", "Test group"), loc=2)
plt.show()
```



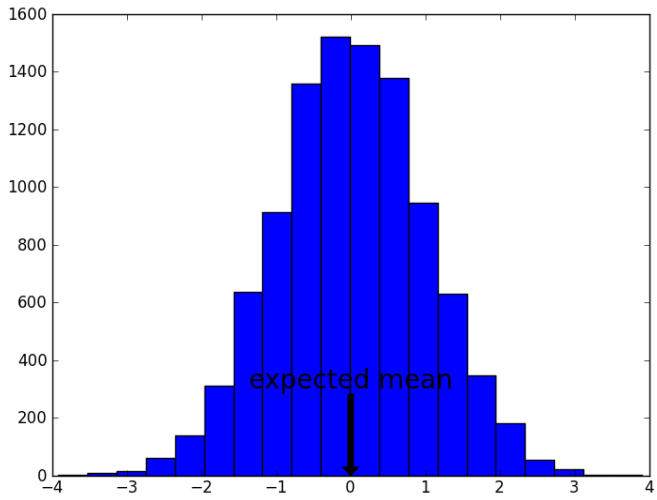
Another commonly used type of graph is the histogram, which is notoriously difficult to create in other programs like Microsoft Excel. We can make simple histograms very easily using matplotlib with the `hist()` function, which we supply with a list (or array) of values and the number of bins to use. 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:

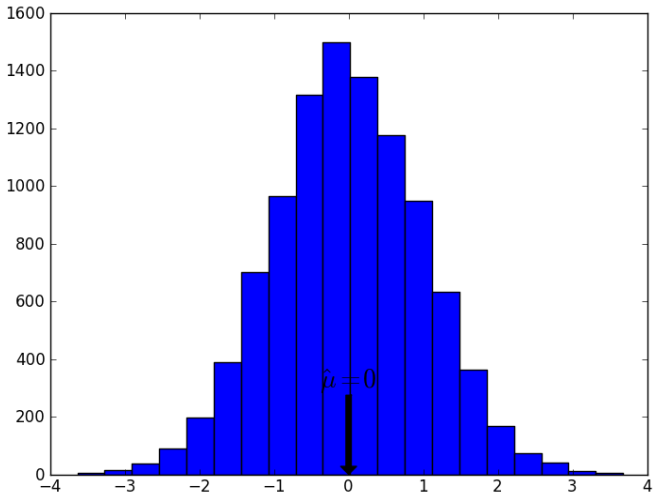
```
from matplotlib import pyplot as plt
from numpy import random
plt.hist(random.randn(10000), 20)
plt.annotate("expected mean", xy=(0, 0), xytext=(0, 300), ha="center",
arrowprops=dict(facecolor='black'), fontsize=20)
plt.show()
```



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 out annotation a little more scientific by adding the symbol μ 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 out histogram on the screen, let's save it out as both a PNG image and a PDF file to our chapter 11 output folder:

```
from matplotlib import pyplot as plt
from numpy import random
plt.hist(random.randn(10000), 20)
path = "C:/Real Python/Course materials/Chapter 11/Practice files/Output/"
plt.savefig(path+"histogram.png")
plt.savefig(path+"histogram.pdf")
```

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()
>>>
```

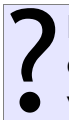
You can then create a plot as usual, *without* having to call the `show()` function to display it, and then make changes to the plot while it is still being displayed by typing commands into the interactive window.

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:

- Recreate all the graphs shown in this section by writing your own scripts without referring to the provided code

Assignment 11.2: Plot a graph from CSV data



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 11 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.**

12) Graphical User Interfaces

12.1) Add GUI elements with EasyGUI

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.

We'll start out in GUI programming with a module named EasyGUI. First, you'll need to download this package:

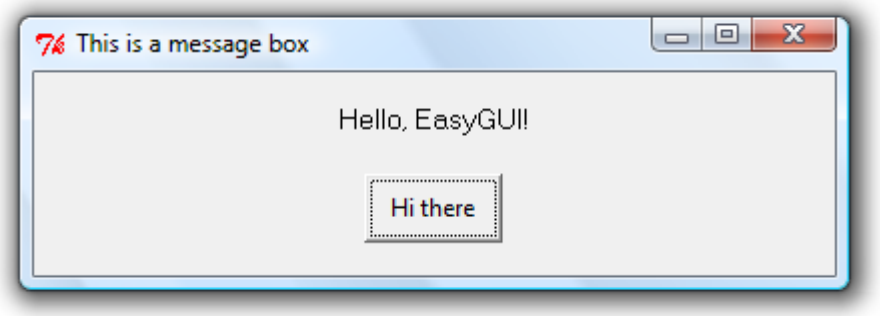
Windows: If you have `easy_install` set up, then you can install the package by typing "`easy_install easygui`" at a command prompt. Otherwise, download the compressed `.zip` file, unzip it, and install the package by running Python on `setup.py` from the command prompt as described in the installation section.
OS X: If you have `easy_install` set up, then you can install the package by typing "`sudo easy_install easygui`" into the Terminal. Otherwise, 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: Type "`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.

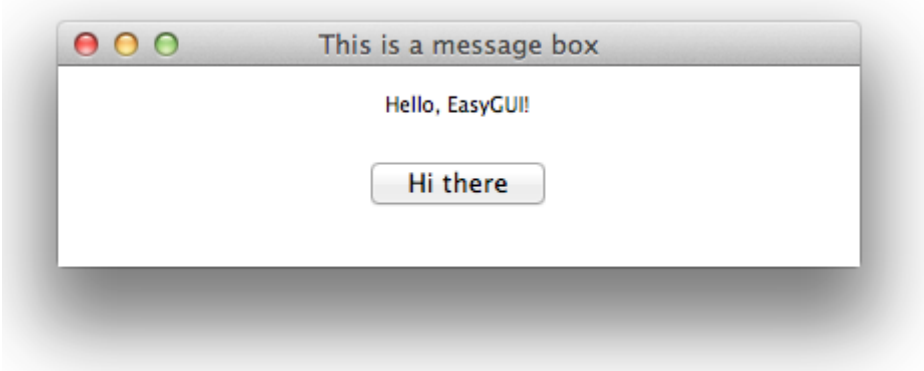
Let's start by importing the functionality from EasyGUI into the interactive window and displaying a simple message box:

```
>>> 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:

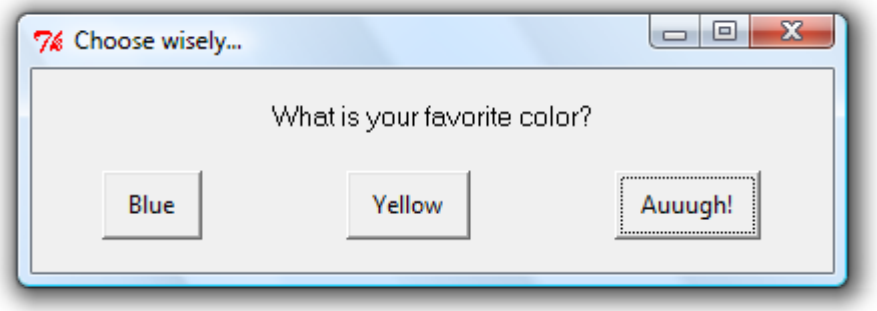
If nothing appears, see the box below. For the sake of the majority, I'll be sticking to screenshots from a Windows GUI perspective only.

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.

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)
```




```
'Auuugh!'
>>>
```

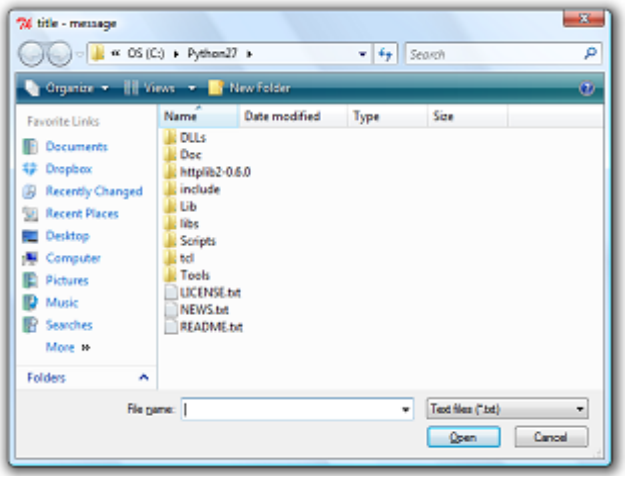
Now we were able to tell that our user chose “Auuugh!” as the favorite color, and we could set that return value equal to a variable to be used later in our code.

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. (See the call-out box at the very end of section 9.1 for a more complete explanation of the `None` keyword.) 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 pyPdf import PdfFileReader, PdfFileWriter

# let the user choose an input file
inputFileName = fileopenbox("", "Select a PDF to rotate...", "*.pdf")
if inputFileName == None: # exit on "Cancel"
    exit()

# let the user choose an amount of rotation
rotateChoices = (90, 180, 270)
message = "Rotate the PDF clockwise by how many degrees?"
degrees = buttonbox(message, "Choose rotation...", rotateChoices)

# let the user choose an output file
outputFileName = filesavebox("", "Save the rotated PDF as...", "*.pdf")
while inputFileName == outputFileName: # cannot use same file as input
    msgbox("Cannot overwrite original file!", "Please choose another
file...")
    outputFileName = filesavebox("", "Save the rotated PDF as...", "*.pdf")
if outputFileName == None:
    exit() # exit on "Cancel"

# read in file, perform rotation and save out new file
inputFile = PdfFileReader(file(inputFileName, "rb"))
outputPDF = PdfFileWriter()
for pageNum in range(0, inputFile.getNumPages()):
    page = inputFile.getPage(pageNum)
    page = page.rotateClockwise(int(degrees))
    outputPDF.addPage(page)
outputFile = file(outputFileName, "wb")
outputPDF.write(outputFile)
outputFile.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), 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), 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:

- Recreate the three different GUI elements pictured in this section by writing your own scripts without referring to the provided code
- Save each of the values returned from these GUI elements into new variables, then `print` each of them
- 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 12.1: Use GUI elements to help a user modify files

Write a script "partial_PDF.py" that extract 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:

- **Let the user choose a file using a `fileopenbox`**
- **Let the user choose a beginning page to select using an `enterbox`**
- **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`**
- **Let the user choose an ending page using another `enterbox`**
- **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`**
- **Let the user choose an output file name using a `savefilebox`**
- **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`**
- **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 ≥ the start)**

12.2) Create GUI applications 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!

! 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*.

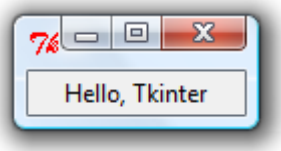
Let's start with a window that only contains a single widget. In this case, we'll use a `Label` to show a single string of text:

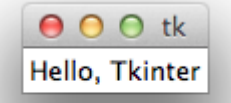
```
from Tkinter import *

myApplication = Tk() # create a new window
greeting = Label(text="Hello, Tkinter") # create a text label
greeting.pack() # add the label to the window

myApplication.mainloop() # run the application
```

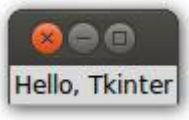
This is already significantly more complicated than easyGUI! First, we had to `import` functionality from the `Tkinter` module. Then we created a new Tkinter window by calling `Tk()` and made a `Label` object containing some text. To actually add the `Label` to our `app` window and make it visible, we had to call the `pack()` method on the `Label`, which “packs” it into the window. Finally, we called the `mainloop()` method to

make our `myApplication` 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:







Again, for the sake of the majority, I'll stick to displaying only the resulting Windows application version for the remained 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:

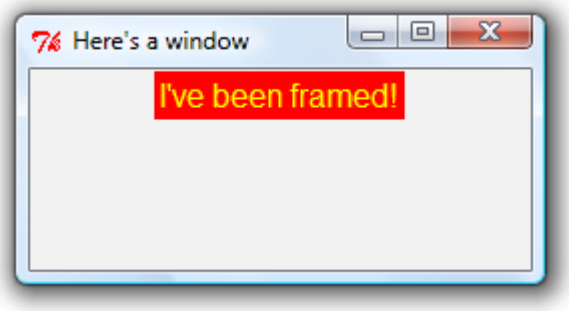
```
from Tkinter import *

# 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
myApplication = App(window)
myApplication.master.title("I made a window!")

myApplication.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.

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

myFrame = Frame()
myFrame.pack()

labelText = Label(myFrame, text="I've been framed!", bg="red", fg="yellow",
font="Arial")
labelText.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 (`myFrame`) 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 organizing the packing of widgets into a frame. For instance, running the same script as above but packing our label with `labelText.pack(fill=X)` will make the label span across the entire width of the

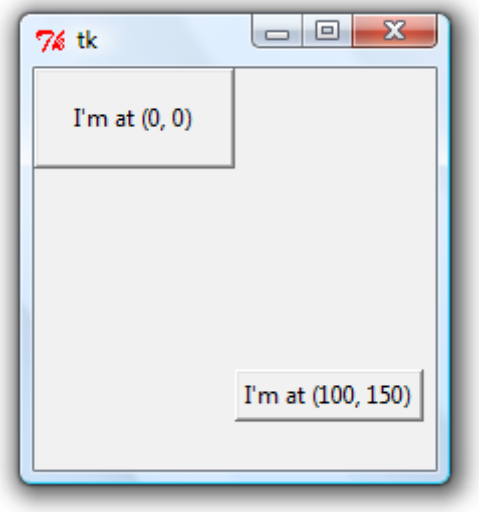
frame. Widgets stack vertically on top of each other by default, but we can pack them side-by-side as well:

```
from Tkinter import *
window = Tk()
myFrame = Frame()
myFrame.pack()

# a bar to span across the entire top
labelText1 = Label(myFrame, text="top bar", bg="red")
labelText1.pack(fill=X)

# three side-by-side labels
labelLeft = Label(myFrame, text="left", bg="yellow")
labelLeft.pack(side=LEFT) # place label to the left of the next widget

labelMid = Label(myFrame, text="middle", bg="green")
labelMid.pack(side=LEFT) # place label to the left of the next widget

labelRight = Label(myFrame, text="right", bg="blue")
labelRight.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("200x200") # default window size
myFrame = Frame()
myFrame.pack()

button1 = Button(myFrame, text="I'm at (100, 150)")
button1.place(x=100, y=150)
```

```
button2 = Button(myFrame, text="I'm at (0, 0)")
button2.place(x=0, y=0, width=100, height=50)

window.mainloop()
```

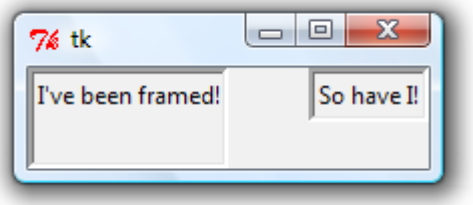


First we set a specific window size, which is 200 pixels wide and 200 pixels tall. We created `button1` using `Button()` and placed it at the location (100, 150), which is halfway across the window and ¾ of the way down the window. We then placed `button2` at the location (0, 0), which is just the upper left corner of the window, and gave it a width of 100 pixels (half the width of the window) and a height of 50 pixels (¼ the height of the window).

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
frameLeft = Frame(bd=3, relief=SUNKEN) # give the frame an outline
frameLeft.place(relx=0, relwidth=0.5, relheight=1)
frameRight = Frame(bd=3, relief=SUNKEN) # give the frame an outline
frameRight.place(relx=0.7, relwidth=0.3)

# add a label to each frame
leftLabel = Label(frameLeft, text="I've been framed!")
leftLabel.pack()
```

```
rightLabel = Label(frameRight, text="So have I!")
rightLabel.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.

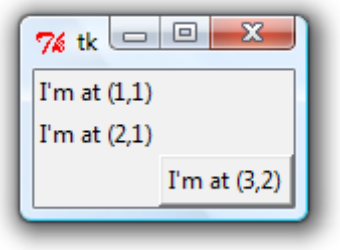
This time we used `relx` to provide a *relative* x placement, which takes a value from 0 to 1 to represent a fraction of the window's width. The first label has `relx=0`, so it appears all the way at the left of the window. Since we gave the second label a `relx=.7` and `relwidth=.3`, it appears 70% 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.

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.

```
from Tkinter import *
window = Tk()
myFrame = Frame()
myFrame.grid() # add frame to take up the whole window using grid()

labelTopLeft = Label(myFrame, text="I'm at (1,1)")
labelTopLeft.grid(row=1, column=1)

labelBottomLeft = Label(myFrame, text="I'm at (2,1)")
labelBottomLeft.grid(row=2, column=1)

buttonBottomRight = Button(myFrame, text="I'm at (3,2)")
buttonBottomRight.grid(row=3, column=2)

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.[30]

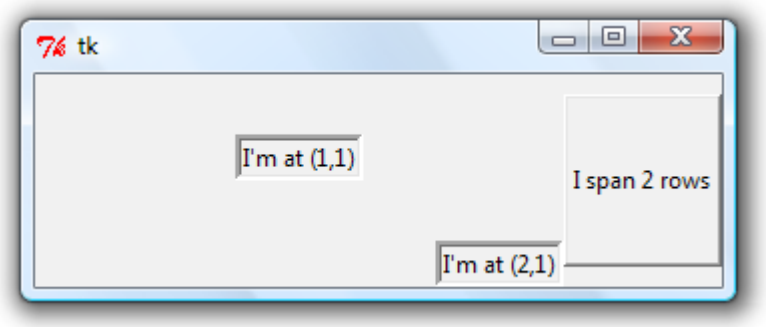
Don't try to combine different ways of adding widgets into a single frame - for instance, don't use `grid()` on some widgets and `pack()` on others. Tkinter has no way to prioritize one method over the other and usually ends up freezing your application entirely when you've given it conflicting packing instructions.

We can assign values to the arguments `padx` and `pady` for any given widget, which will

30 Actually, they start counting at 0, but cells without any widgets in them are ignored; we could have started with the first cell at `row=10` and `column=25` if that were the location of our top left widget, and it would still appear in the upper left corner.

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()
myFrame = Frame()
myFrame.grid() # add frame to window using grid()

labelTopLeft = Label(myFrame, text="I'm at (1,1)", bd="3", relief=SUNKEN)
labelTopLeft.grid(row=1, column=1, padx=100, pady=30)

labelBottomLeft = Label(myFrame, text="I'm at (2,1)", bd="3", relief=SUNKEN)
labelBottomLeft.grid(row=2, column=1, sticky=E)

buttonRight = Button(myFrame, text="I span 2 rows", height=5)
buttonRight.grid(row=1, column=2, rowspan=2)

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.

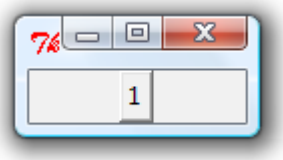
We added a button in the second column, specifying a height of 5 (text lines). This allows us to see that, even though we placed the button at `row=1`, since we also

specified `rowspan=2`, the button is centered vertically across both the first and second rows.

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 `incrementButton()` that takes the number displayed on our button and increments it by one:

```
from Tkinter import *
window = Tk()

def incrementButton():
    newNumber = 1 + button.cget("text")
    button.config(text=newNumber)

myButton = Button(text=1, command=incrementButton)
myButton.pack()

mainloop() # start the application
```



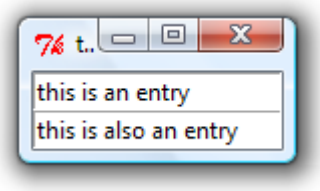
Now each time you click on the button, it will add one to the value by configuring the button with a new `text` attribute. We used the `cget()` method on the button to retrieve the `text` currently displayed on the button; we can use `cget()` in this same way to get the values taken by other attributes 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()
myText = entry1.get() # get the text from entry1
entry2.insert(0, myText)
entry2.insert(8, "also ") # add "also" to the middle of myText

mainloop() # start the application
```

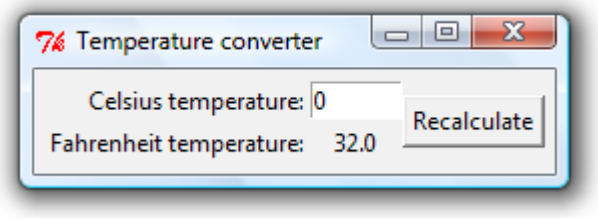


Here we packed two entry boxes into a window, adding new text at index position 0 to `entry1` first. We added the text seen in `entry2` by first using `get()` to return the text in `entry1`, inserting this same text into `entry2`, then inserting additional text into this string starting at index location 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 chapter 4 to create a GUI-based temperature conversion application:

```
from Tkinter import *

def recalc():
```

```
        celTemp = entryCel.get() # get temp from text entry
        try: # calculate converted temperature and place it in label
            farTemp = float(celTemp) * 9/5 + 32
            farTemp = round(farTemp, 3) # round to three decimal places
            resultFar.config(text=farTemp)
        except ValueError: # user entered non-numeric temperature
            resultFar.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
labelCel = Label(frame, text="Celsius temperature:")
labelCel.grid(row=1, column=1, sticky=S+E)
labelFar = Label(frame, text="Fahrenheit temperature:")
labelFar.grid(row=2, column=1, sticky=S+E)

# create and add space for user entry of text
entryCel = Entry(frame, width=7)
entryCel.grid(row=1, column=2)
entryCel.insert(0, 0)

# create and add label for text calculation result
resultFar = Label(frame)
resultFar.grid(row=2, column=2)

# create and add 'recalculate' button
btnRecalc = Button(frame, text="Recalculate", command=recalc)
btnRecalc.grid(row=1, column=3, rowspan=2)

recalc() # fill in first default temperature conversion
mainloop() # start the application
```



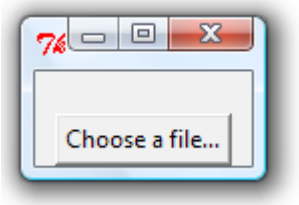
First we created the function `recalc()`, which gets the Celsius temperature from the entry box, converts it into Fahrenheit, and sets that (rounded) number as the text of the Fahrenheit temperature label. Notice how we used the `try/except` structure to check if the user entered something that isn't 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 the separate `tkFileDialog` module to allow the user to open and save files, much like we did before with EasyGUI. 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 *
import tkFileDialog

window = Tk()
frame = Frame()
frame.pack()

def openFile(): # ask user to choose a file and update label
    typeList = [("Python scripts", "*.py"), ("Text files", "*.txt")]
    fileName = tkFileDialog.askopenfilename(filetypes=typeList)
    labelFile.config(text=fileName)

# blank label to hold name of chosen file
labelFile = Label(frame)
labelFile.pack()

# button to click on for "Open" file dialog
buttonOpen = Button(frame, text="Choose a file...", command=openFile)
buttonOpen.pack()

mainloop() # start the application
```



Clicking the button will cause our function `openFile()` to run, which opens a file dialog using `tkFileDialog.askopenfilename()`. Make sure you import tkFileDialog as well as Tkinter, since they are actually separate modules!

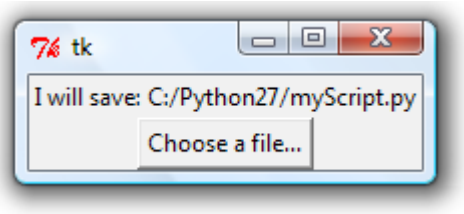
We passed `typeList` into the argument `filetypes` of the `tkFileDialog.askopenfilename()` function in order to provide a list of the different types of files that we want the user to be able to choose; these are provided as a list of tuples, where the first item in each tuple is a description of the file type and the second item in each tuple is the actual file extension.

Just like with EasyGUI, the `tkFileDialog.askopenfilename()` function returns the full name of the file, which we can then set equal to our string `fileName` 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 `tkFileDialog.asksaveasfilename()` function that takes the same arguments as `tkFileDialog.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 *
import tkFileDialog

window = Tk()
frame = Frame()
frame.pack()

def saveFile(): # ask user to choose a file and update label
    typeList = [("Python scripts", "*.py"), ("Text files", "*.txt")]
    fileName = tkFileDialog.asksaveasfilename(filetypes=typeList,
defaultextension=".py")
    myText = "I will save: " + fileName
```

```
    labelFile.config(text=myText)

# blank label to hold name of chosen file
labelFile = Label(frame)
labelFile.pack()

# button to click on for "Open" file dialog
buttonOpen = Button(frame, text="Choose a file...", command=saveFile)
buttonOpen.pack()

mainloop() # start the application
```



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 may 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:

- 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
- 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

Assignment 12.2: Return of the poet

Write a script "poetry_writer.py" based on the poem generator created in assignment 6.1 that assists the user in creating beautiful poetry through a GUI.

- The user should be instructed (in a label) to enter words into each list, separating words with commas. You can use the string `split()` method to generate lists of individual words.

- Provide a grid layout with labels for `Nouns`, `Verbs`, `Adjectives`, `Prepositions` and `Adverbs`, where the user can supply a list of words for each by typing them into entry boxes.

- Add a "`Create your poem`" button centered at the bottom of the window that, when clicked, will generate and display a random poem in a label below the button. If the user does not supply enough words, the "poem" label should instead display text letting the user know the error.

- Add a "`Save your poem`" button centered below the poem that, when clicked, allows the user to save the poem currently being displayed as a .TXT file by prompting the user with a "`Save As`" dialog box. Add a default file extension of "`.txt`" and make sure that your program does not generate an error if the user clicks "`Cancel`" on the file save dialog box.

- You can assume that the user will not supply any duplicate words in any of the word lists.

- Bonus: Add a function `checkUnique()` that takes a list of words as an argument and returns `True` or `False` based on whether or not that list contains only unique entries. Use this function to alert the user if duplicate words are present within any of the word lists by displaying the error in the "poem" label. (Without this check, we run the risk of our program entering an infinite loop!)

13) Web applications

13.1) Create a simple web application

You know how to write useful Python scripts, and now you want to show them off to the world... but how? Most non-programmers won't have any use for your `.py` script files. Programs like PyInstaller and cx_Freeze help turn Python scripts into executable programs that can be run by themselves on different platforms without the need to use Python to interpret the code. More and more, however, we're seeing a trend away from "desktop"-based applications and toward web applications that can be accessed and run through Internet browsers.

Historically, websites on the Internet were full of plain webpages that offered the exact same information to every user; you would request a page, and the information from that page would be displayed. These webpages were "*static*" because their content never changed; a web server would simply respond to a user's request for a webpage by sending along that page, regardless of who the user was or what other actions the user took.

Today, most websites are actually web *applications*, which offer "*dynamic*" webpages that can change their content in any number of ways. For instance, a webmail application allows the user to interact with it, displaying all sorts of different information, often while staying in a single webpage.

The idea behind creating a Python-driven web application is that you can use Python code to determine what content to show a user and what actions to take. The code is actually run by the web server that hosts your website, so your user doesn't need to install anything to use your application; if the user has a browser and an Internet connection, then everything else will be run online.

The task of getting Python code to run on a website is a complicated one, but there are a number of different web frameworks available for Python that automatically take care

of a lot of the details.

The first thing that you will need is a web hosting plan that allows and supports the ability to run Python code. Since these usually cost money (and since not everyone even has a website), we'll stick with a free alternative that is one of the simplest to set up: Google App Engine, which uses a web framework called webapp2.

There are a number of other alternatives (both free and paid) that are more customizable, and you can use webapp2 on its own later without relying on Google App Engine, but getting started with Google App Engine will be the quickest and easiest way to begin learning about web application development in Python.

First, go here to download and install the appropriate Python SDK for Google App Engine.[31] SDK stands for **S**oftware **D**evelopment **K**it. This particular SDK includes two main resources: a "web server" application, which will allow you to run your web applications on your own computer without actually putting them online, and the Google App Engine Launcher, which will help to put your web applications online.

Before we dive into writing a web application, let's get a very broad, generalized overview of what's about to happen. There are a lot of different pieces involved, and they all have to communicate with each other to function correctly:

- First, your user makes a "request" for a particular webpage on your website (i.e., by typing a URL into a browser).
- This request gets received by the web server that hosts your website.
- The web server uses App Engine to look at the configuration file for your application. App Engine matches the user's request to a particular portion of your Python script.
- This Python code is called up by App Engine. When your code runs, it writes out a "response" webpage.
- App Engine delivers this response back to your user through the web server.
- The user can then view the web server's response (i.e., by displaying the resulting webpage in a browser).

31 Linux users, install the folder in your `/home/username` directory to avoid typing out long paths later.

The application we're going to create will rely on a couple different files, so the first thing we need to do is make a project folder to hold all of these files. Create a new folder named "`first_app`" anywhere you like (just remember where it is).[32] First we will write a very simple Python script that can "respond" with the content of your webpage:

```
print "Content-Type: text/plain"
print ""
print "Congratulations, it's a web app!"
```

Save this code in a script named "`hello.py`" inside your "`first_app`" folder.

So what's with the first two `print` statements? Web servers communicate with users (usually browsers) through HTTP by receiving *HTTP requests* and sending *HTTP responses*. The HTTP response that our application sends can have both *header lines* and a *body*.

We added a header line to our HTTP response in the first line. Header lines contain optional information to let a browser know how to interpret the body of the response. In this case, setting our header's "`Content-Type`" equal to the value "`text/plain`" is the way that our HTTP response lets a browser know to expect the body to contain plain text as opposed to HTML code, an image, or some other type of file.[33] Leaving a blank line after this header line is how we told the browser, "the header lines are over now; here comes the actual body to display."

The body of the response is what we will actually see when we load the page in a browser. In this case, it's just a simple string of text: "`Congratulations, it's a web app!`"

Before we can run our web application, we need to provide App Engine with a *configuration file*. This is the file that the web server will use to get information about what Python code we want it to run. Open up any text editor (or another script window in IDLE) and copy the following text into a new file:

```
application: hello
version: 1
```

32 Linux users, you should save the folder in your `/home/username` directory to keep your life simple.
33 Most browsers are smart enough to correctly infer the content type without having to be told (especially if the content is text or HTML), but it's usually safer to include this header line just in case.

```
runtime: python27
api_version: 1
threadsafe: false

handlers:
- url: /.*
  script: hello.py
```

Now name this file "`app.yaml`" and save it in the same "`first_app`" folder as the Python script.[34] Make sure you get the spacing correct; the last line includes two leading spaces so that "`script`" lines up under "`url`". Just like Python, YAML[35] files rely on precise indentation.

The YAML configuration file gives App Engine all the necessary information it needs to run the web application. First, "`application: hello`" provides a unique identifying name for the application so that we will be able to launch it later; we are giving the name "`hello`" to our web app.

The line "`version: 1`" lets App Engine know that this is version 1 of our application. (If we update this later to "`version: 2`", App Engine will keep a copy of version 1 in memory so that we can go back to running this previous version if necessary.)

The lines "`runtime: python27`" and "`api_version: 1`" let App Engine know that we want to use Python 2.7 to run our application. (There is currently only one version of the Python API offered by Google.)

The line "`threadsafe: false`" means that our web application isn't designed to be able to receive multiple requests at once; if App Engine has multiple requests then it will send them to our application one at a time instead of all at once.

Finally we define our "`handlers`" to handle different webpage requests from our users (i.e., if a user requested the main page at "`/`" or another page at a different address on our site). These requested paths can each be assigned to a different piece of Python code. In this case, we only have one script, `hello.py`, so we want to direct user requests for *any* page on the website to the same script. In these last two lines of the

34 Make sure that you've save this file with a `.yaml` extension and that your text editor didn't add a file extension of its own (i.e., creating a file like `app.yaml.py` or `app.yaml.txt`).
35 If you thought the acronym for PHP was bad… YAML stands for **Y**AML **A**in't **M**arkup **L**anguage.

configuration file, we say that any URL matching the regular expression "`/.*`" (which is *any* URL on our site) should be directed to the "`hello.py`" script.

Okay, now we can *finally* take a look at our application! It's not online yet, but we can view it by running the application through our own "local" web server (that can't be accessed by other users) using Google App Engine. This will help us simulate what things will look like to a user once our application *is* online.

Open the Google App Engine Launcher program, then choose File → Add Existing Application… You can then browse to and select your `first_app` folder that contains the web application. Add the application using port 8080, then select the application in the main window and click the green "Run" button.

Linux users: you will need to navigate in your Terminal to the directory *just before* the `first_app` folder (i.e., its parent directory), then type the following command to launch the web application (which will run on port 8080 by default):

```
google_appengine/dev_appserver.py first_app/
```

The console window that appears will track lots of extra information, but your web application is up and running once you see a blinking cursor.

The "port" number can be thought of as selecting a particular channel to use, similar to broadcasting a television or radio channel. We chose to run the web application on port 8080, meaning that the user can essentially "tune in" to this port number and receive communication from our web server. (We could host a completely different web application using a different port number and the two would not interfere with each other.)

Once the web application is running (this might take a little while), we can click "Browse" to view the web application in the default web browser. This will open up the page at the URL "`localhost:8080`" (which we can also type into a browser manually to load the web application).[36] The web address "`localhost`" is just a way of saying "the web address of my own computer" (since the application isn't actually online yet).

36 Using Internet Explorer, you'll have to add "`http://`" to the URL. Alternatively, use a real browser.

The "`:8080`" specifies that we should listen for communication on port number 8080.[37] If everything has been set up correctly, your browser should load a page that displays the plain text:

```
Congratulations, it's a web app!
```

If you make any changes to your script, as long as Google App Engine Launcher is still running your application, all you need to do in order to view the newest version of the web application is to save the script and reload the webpage. App Engine will automatically listen for any changes that might have been made.

Well, that's a start. As far as an "application" goes, however, the Python script involved was fairly useless. In order to make something with a bit more potential, we need to create a special object in our Python code called a `WSGIApplication`. WSGI stands for **W**eb **S**erver **G**ateway **I**nterface and is a way to allow Python to communicate with the web server in a better way than simply "printing" a single chunk of information back as a response. Our new Python script, which still just displays the same line of text, is considerably more complicated:

```
import webapp2

class MainPage(webapp2.RequestHandler):
  def get(self):
      self.response.headers["Content-Type"] = "text/plain"
      self.response.write("Congratulations, it's a web app!")

routes = [("/", MainPage)]
myApp = webapp2.WSGIApplication(routes, debug=True)
```

We now had to import `webapp2` in order to use WSGI. In the last line of our script, we are using the webapp2 web framework to create a `WSGIApplication` object named "`myApp`" that will allow our code to communicate with the web server.

This time we are creating a new *class* using the "`class`" keyword; this means that `MainPage` is a new type of object (specifically, a type of `webapp2.RequestHandler` object) that can have its own methods and attributes just like any other object. Even though it looks like the `get()` method takes an argument of "`self`", this special

37 You will only need to specify a port number when running the web application from your own computer. HTTP communication online always occurs on port 80 by default (to prove this to yourself, try going to a site like http://google.com:80 in a web browser), so typically nobody needs to specify a port online.

keyword is automatically created and used internally by the new object as a way to keep track of itself. We have to use this "`self`" keyword within a new object that we've defined, because the object needs to know when it's referring to its *own* methods and attributes. For instance, we have to call `self.response` in order to access the `response` stored in our `MainPage RequestHandler` object.

As the first argument of our `WSGIApplication`, we passed a list of tuples called `routes`. In this case, `routes` only contains one tuple, `("/", MainPage)`. This "route" tells our WSGI application that anyone accessing the main directory of our application should get directed to `MainPage`. We call `MainPage` a "request handler" because it is in charge of handling any user requests for that particular webpage.

The "`/`" is how we represent the main "root" directory of our application (i.e., our website). If we had provided a second tuple, `("/images/", ImagesPage)`, then anyone who accesses the (hypothetical) "images" directory of our website would be directed to the class `ImagesPage` in the Python script.

If we make a mistake in the code, our application's "`debug=True`" argument will allow us to see the errors reported by Python in our web application; otherwise, all we would see is a "`500 Internal Server Error`" message.

Because of how we defined `routes`, when our web server receives a request to load the page "`/`", `webapp2` creates a new `MainPage` object (called a *request handler* because it responds to the request for a webpage). The `get()` method of `MainPage` is automatically called when it is created, which is how our `WSGIApplication` responds to the web server. The response is written out with these two lines:

```
self.response.headers["Content-Type"] = "text/plain"
self.response.write("Congratulations, it's a web app!")
```

Again, we have to write both a *header* line and a *body*. The header line gets packed into `headers` like in a dictionary, setting the value of the "`Content-Type`" equal to "`text/plain`". This time, we create the body of the response using the `write()` method. WSGI takes care of separating the header lines from the body, so this time there's no need to `write()` a blank line first.

We now have to update our YAML configuration file as well:

```
application: hello
version: 1
runtime: python27
api_version: 1
threadsafe: false

handlers:
- url: /.*
  script: hello.myApp
```

The only difference from the previous configuration file is the very last line, where we point the "`script`" to use `hello.myApp` instead of `hello.py`; instead of the entire Python script, we now want the web server to run the `WSGIApplication` object inside of our script named `myApp`, which we access using the simple dot notation of "`hello.myApp`".

If all goes well, you should now be able to save these two files, reload the webpage, and see the exact same thing as before:

```
Congratulations, it's a web app!
```

! If you make a mistake in your Python script, your web application might load a horrendous error page that makes it look as if you broke the Internet for good. Don't panic! Just look at the last line, which will point you to the specific error (usually a line of your code displayed in red) that caused the entire chain of failure.

If you do end up with an error that your browser refuses to give you further advice about (instead showing a "`500 Internal Server Error`", even when you've set your `WSGIApplication` to have `debug=True`), you can try running the Python script itself in IDLE. Although it won't run (because Python won't be able to find the `webapp2` module), IDLE can still point out if your code has any syntax errors.

Review exercises:

- Play around with your web application; make small changes to the script and see how App Engine responds
- Use your `WSGIApplication` to create a basic HTML page by changing the `Content-Type` header value to "`text/html`" and then using `write()`

multiple times to respond with lines of HTML code; for instance, the first `write()` statement could be `self.response.write("<html>")` to begin the HTML webpage

13.2) Create an interactive web application

Now that we've upgraded our web application to use WSGI, we can create multiple pages that interact with the user. We'll start by creating an HTML form that takes some text input from the user and displays that text back on a new page.

If you aren't familiar with HTML forms, you should take a moment to review the basics first. We already saw an example of a basic form with the login.php page used for web scraping practice in chapter 10.

The first URL of our web application will be the main "`/`" page that users see by default. Our second page will be located at "`/welcome`" (i.e., this path will appear after the name of the website). In order to do this, we need to include two tuples in our `routes`:

```
routes = [('/', MainPage), ('/welcome', Greeting)]
```

We don't need to update the configuration file for, because we still want *any* request for a webpage to be directed to our `WSGIApplication` object `myApp` inside the `hello.py` script. From there, `myApp` is responsible for directing any webpage requests to the correct "request handler" - either `MainPage` or `Greeting`. Our full web application code will look like this:

```
import webapp2

class MainPage(webapp2.RequestHandler):
    def get(self):
        self.response.headers["Content-Type"] = "text/html"
        self.response.write("""
          <html>
            <head><title>Enter your name...</title></head>
            <body>
              <form action="/welcome" method="post">
                <input type="text" name="myName"><br>
```

```
                <input type="submit" value="Sign In">
              </form>
            </body>
          </html>""")

class Greeting(webapp2.RequestHandler):
    def post(self):
        username = self.request.get("myName")
        welcomeString = """<html><body>
                        Hi there, {}!
                        </body></html>""".format(username)
        self.response.headers["Content-Type"] = "text/html"
        self.response.write(welcomeString)

routes = [('/', MainPage), ('/welcome', Greeting)]
myApp = webapp2.WSGIApplication(routes, debug=True)
```

This time, we specified the `Content-Type` to be "`text/html`" because we're writing out webpages in HTML code. We did this using multi-line strings, although we also could have broken the content into multiple `write()` commands.

As before, our main page (handled by the request handler `MainPage`) has a `get()` method so that it can respond to requests to "get" its information. In the content body, we included a title for our webpage and a simple form. Our form has a single text input field called "`myName`" and a submit button labeled "`Sign In`". When a user clicks on the "`Sign In`" button, our form posts its data (i.e., the value of `myName`) to the webpage located at "`/welcome`".

Meanwhile, our "`/welcome`" page (handled by the request handler `Greeting`) has a `post()` method so that it can respond when it receives "posted" data. We *could* have used `get()` to send the user's data (and we will in the next example), but information that is sent using `get()` becomes a part of the URL; this would become a problem if we later decided that we want the user to include sensitive information (like a password).[38]

Within `post()`, we used `self.request.get("myName")` to get the user-supplied value of `myName` from the user's "post" request, which we stored in a string variable called `username`. We then created a `welcomeString`, which is a multi-line string representing the full HTML code for our "`/welcome`" page, including the inserted value

38 Although it usually isn't needed, a single request handler can handle both "get" and "post" requests; just include both a `get()` method and a `put()` method inside the definition of the request handler.

of `username`. As before, we then `write()` out the header line and content body to display the resulting webpage.

Save your script and reload the browser to view your new web application. Notice how, when you "`Sign In`" to the web application, you are directed to the new webpage "`localhost:8080/welcome`" - which should hopefully display back what you typed!

Actually, there's one important change we need to make to our code. The fact that we take input from a user and display that input back *without* first investigating what we are about to display is a **huge security hole**. Sure, maybe you don't expect to have malicious users who are actively trying to break your application, but never underestimate the potential for users to do unexpected things that cause your application to break in unexpected ways.

For instance, maybe someone decides that an appropriate username to enter into our application is "`<b>`". Our "`/welcome`" webpage ends up displaying:

> Hi there, **!**

Since we're inserting this text into HTML code, the "`<b>`" was interpreted as an HTML tag to begin making text bold - so instead of greeting out user, we only change our explanation point to be displayed bold. (You can imagine how this might present a security problem; any user can now write code that runs on our web server.)

To avoid this, we can use Python's built-in `cgi.escape()` function, which converts the special HTML characters `<`, `>`, and `&` into equivalent representations that can be displayed correctly.[39] You will first need to `import cgi` into your Python script to use this functionality. Then, when you `get()` the value of `myName` from the user's request, you can convert any special HTML characters by instead saying:

```
username = cgi.escape(self.request.get("myName"))
```

With these changes, try re-running your web application and signing in with a username of "`<b>`" again. You should now see it display the username back to you correctly:

> Hi there, <b>!

39 CGI (Common Gateway Interface) programs are typically used by web servers to process user input.

Okay, so we can now create one webpage that helps the user "post" data to another webpage interactively. What about using the `get()` method to make a *single* page interactive? Let's look at an example with just a little bit more Python code behind it and revisit the "temperature converter" script that we wrote way back in chapter 4 and again as a simple GUI application.

The webpage we want to display will have a simple text field where the user can input a temperature in degrees Celsius. We will also include a "`Convert`" button to convert the user's supplied Celsius temperature into degrees Fahrenheit. This converted result will be displayed on the next line and will be updated whenever the user clicks the "`Convert`" button. The HTML for this page, with place-holders for the actual temperature values, will look like so:

```
<html>
  <head><title>Temperature Converter</title></head>
  <body>
    <form action="/" method="get">
      Celsius temperature: <input type="text" name="celTemp" value={}>
      <input type="submit" value="Convert"><br>
      Fahrenheit temperature: {}
    </form>
  </body>
</html>
```

This time, our form uses the "`get`" method with a form "`action`" that points back to the main page itself. In other words, when the user submits this form by clicking on the "`Convert`" button, instead of sending a "`post`" request to a new webpage, the user will send a "`get`" request for the same page, providing the page with some input data.

Just as we did before, we will want to put the temperature conversion into a function of its own. The full code will look as follows:

```
import webapp2

def convertTemp(celTemp):
    ''' convert Celsius temperature to Fahrenheit temperature '''
    if celTemp == "":
        return ""
    try:
        farTemp = float(celTemp) * 9/5 + 32
        farTemp = round(farTemp, 3) # round to three decimal places
```

```
            return str(farTemp)
        except ValueError: # user entered non-numeric temperature
            return "invalid input"

    class MainPage(webapp2.RequestHandler):
        def get(self):
            celTemp = self.request.get("celTemp")
            farTemp = convertTemp(celTemp)

            self.response.headers["Content-Type"] = "text/html"
            self.response.write("""
              <html>
                <head><title>Temperature Converter</title></head>
                <body>
                  <form action="/" method="get">
                    Celsius temperature: <input type="text"
                                           name="celTemp" value={}>
                    <input type="submit" value="Convert"><br>
                    Fahrenheit temperature: {}
                  </form>
                </body>
              </html>""".format(celTemp, farTemp))

    routes = [('/', MainPage)]
    myApp = webapp2.WSGIApplication(routes, debug=True)
```

As before, our `convertTemp()` function checks to see if the user supplied a valid value. Since the user won't have supplied *any* value when the page first loads (and we don't want to start out by saying "`invalid input`"), we check first for this condition and return an empty string if the text field is also blank.

We used `self.request.get("celTemp")` to get the user-supplied value of `celTemp` from the user's "get" request (just like we did for the user's "post" request in the previous example). The HTML code that we `write()` out as a multi-line string now includes two user-supplied values: the `celTemp`, which is the value that the user entered into the text field, and the `farTemp`, which is the calculated result returned by our function. If we had not written the value of `celTemp` into the text box, then this input box would be cleared when the user clicks on the button.

Save this script as "`temperature.py`" in the same "`first_app`" folder, then update the `app.yaml` configuration file accordingly. Since we renamed our script, we changed the name of the module where the `WSGIApplication` object `myApp` should be loaded:

```
application: temperature-converter
version: 1
runtime: python27
api_version: 1
threadsafe: false

handlers:
- url: /.*
  script: temperature.myApp
```

We also updated the name of the application just to provide a descriptive name for what the application actually does. Even if you left App Engine running, this name will update automatically in the Launcher. Notice how we used a dash but didn't use any capitalization in the name; application names for Google App Engine can only include lower-case letters, digits and hyphens.

You should now be able to use your new web application, supplying temperatures and seeing the converted result appear on the same webpage. Since we use a "get" request, we can also now see the user-supplied data appear in the URL. In fact, you can even circumvent the form and provide your own value for `celTemp` by supplying an appropriate address. For instance, try typing the URL `localhost:8080/?celTemp=30` directly into your browser and you will see the resulting temperature conversion.

Review exercises:

- Modify the "log in" web application example so that it only has a single main webpage that can receive "get" requests from the user; instead of a "`Log In`" button under the text field, make a "`Greet me!`" button that, when clicked, reloads the page to greet the user by name (if a name has been supplied) and display the greeting form again

Assignment 13.2: The poet gains a web presence

Recreate the random poem generator that we saw in assignments 6.1 and 12.2 by turning it into an interactive web application. You should use a web form like the one at poem-generator.appspot.com to help your user generate poems. You can view the source of this webpage to reverse-engineer most of the necessary HTML code. Your web application should use a single webpage with a single form that accepts "get" requests from the user.

Bonus: Learn how to use static files like CSS in your web application to improve its overall appearance and make your HTML code easier to read and maintain.

13.3) Put your web application online

It's finally time to share your beautiful work with the world. Getting your content online is actually a fairly simple process with Google App Engine.

Google provides a free service at appspot.com to allow you to host up to ten different web applications (as long as you don't have millions of users accessing them every month, in which case you will need to upgrade to a paid plan). You will first need to go to appengine.google.com and sign in to App Engine.[40]

Once you've logged into App Engine, you can create a new application and give it a unique "`Application Identifier`", which is the name of your web application. This name will appear in the URL of your application, which will look like:

```
http://your-application-name.appspot.com
```

Since this is a direct URL that must be unique, you might need to try a few times before you find a name that isn't already in use by someone else.[41] I'm going to use the temperature conversion application and name mine "`temperature-converter`" - which means you can't! Whatever you end up using as your application identifier, be sure to update your configuration file so that the name provided for the "`application`" in the very first line of the file matches this application name. The title that you provide for your application will be the name that appears on the webpage's title bar (so in this case, specifying a `<title>` in the HTML was superfluous).

Once you've registered an application ID and updated your web application to reflect this name, you're ready to go! Just click the "`Deploy`" button in App Engine Launcher and log in with your Google account to load the application onto your new webpage. A new window will appear, offering details of the upload process; it may take a minute or

40 If you don't have a Google account yet (seriously?), you will need to sign up to create one first.
41 If you use Google Apps (which also includes a free basic plan), you can put the web application on your Google Apps site as well.

two to complete.

Linux users: you will need to navigate to the folder just *before* the `first_app` folder, then use the following command for uploading the application through App Engine:

```
google_appengine/appcfg.py update first_app/
```

If you see the message "`This application does not exist`", then either you have not registered the application correctly or the name of your web application does not match the application identifier that you provided.

Congratulations! You now have a live webpage of your own running Python code.

Google has a number of other handy built-in features, including the ability to store "persistent" user data (that lasts even after the user leaves the website) within your application and even a way to integrate your application with your users' Google accounts. The current tutorials and documentation include functional but only halfway up-to-date example code, although it can still provide an idea of what is possible beyond the basics.

For complex websites, writing out long blocks of HTML is fairly inefficient and hard to manage, which is why a number of frameworks have been developed to help organize and speed up the web development process using templates and pre-built pieces of code. Django is perhaps the most popular web framework. CherryPy and web2py are other popular options; web2py is especially good at integrating with databases. All of these web frameworks can be run on their own or set up through Google App Engine instead of using webapp2. In fact, App Engine supports Django templates within webapp2 by default. Other "micro-frameworks" that are gaining popularity for getting smaller web development tasks done quickly include Flask and tipfy.

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. Log in at the RealPython.com members' forum and help out other new Python students; the only way you know you've *really* mastered a concepts is when you can explain it to someone else.

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*.

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 in the member forum. 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 | József Tschosie Kovács | Back Kuo | James Anselm | Christian Gerbrandt | Mike Stoops | Michael A Lundsveen | David R. Bissonnette, Jr. | Geoff Mason | João 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 | Jérôme 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 Jánošík | 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" Källarsson | 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 Thorsén | Mike and Julia Nelson | code_more | Hunter Harris | Terrence DeBeatham | Ryan Wheeler | Jerrid D.C. Fuehrer | Rémy 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 Borgström | 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 Kankaanpää | 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 | Régis LUTTER | Charles Jurica | José 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?pánka Šoustková | 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 | Héctor Gracia | Eric Finkenbiner | Achal Gautam | Stéphane QUENARD | Wee Wing Kwong | Alan Carpilovsky | roland köhler (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 | Håvar 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 Gálvez 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 Wennerström | 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 Tomé | 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 Mühlpachr | 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 Grönlund | Simon Jubinville | Aerie Ward | Thim Frederik Strothmann | Chasester | Laughing Magi | Wagner, Earl | brianwik | Owen Ness | Flávio 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 | Joël 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! | José de Jesús Medina Ríos | 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 Söderström | 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 Göbel | 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 Arvenäs | 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 | Adam Gross | 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 Langbråten | 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-André 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 Jääskeläinen | 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 Hlll | 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 Gíslason | 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 | Anonymous