Johnny Wei-Bing Lin

Lecture Notes on Programming Theory for Management Information Systems

Version 8

http://www.johnny-lin.com/infosys

2019

Contents

Preface

Why this book and who it is for

This book is for management and information systems (MIS) students who are taking a second course in programming. In it, I aim to teach the programming principles (structured programming, classes, data structures, search and sorting, and recursion) through learning how to solve MIS problems. As a result, this book does not work well as a computer science reference, as the programming theory is embedded inside MIS problems.

But, if you're an MIS student, I figure that this way of approaching the topic is a plus. After all, if you wanted to become a programmer, you would have majored in Computer Science instead of MIS. (When I was in college, I certainly did not want to become a programmer, which is why I took only one CS course and majored in Mechanical Engineering. How I ended up teaching programming for a living is a funny story ☺.) This book, then, is a book on Python programming for students who are interested in learning Python but who want to learn first and foremost how Python will help them in their *own* work. The computer science will just come along for the ride.

I assume that readers will have had some background in procedural programming (perhaps Java or C++) and so already understand variables, arrays, looping, conditionals (`if/then`), simple input/output, and subroutines/functions. I also assume that in their work, most readers use a procedural programming methodology, writing programs that are broken up into subroutines and functions where input is passed in and out using argument lists (or common blocks or modules).

Software you'll need

I assume that you are running Python 3.x through an installation of the Anaconda or Canopy distributions. If not, see:

- Anaconda: https://www.anaconda.com/download/
- Canopy: https://www.enthought.com/products/canopy/

Both distributions have a version that is free and can be installed without administrator privileges.

Typesetting and coloring conventions

Throughout the book, I use different forms of typesetting and coloring to provide additional clarity and functionality (text coloring is available in only some versions of the book; later on in this preface I discuss the different versions of the book). Some of the special typesetting conventions I use include:

- Source code: Typeset in a serif, non-proportional font, as in `a = 4`.
- Commands to type on your keyboard or printed to the screen: Typeset in a serif, non-proportional font, as in `print('hello')`.
- Generic arguments: Typeset in a serif, proportional, italicized font, in between a less than sign and a greater than sign, as in <*condition*>.
- File, directory, and executable names: Typeset in a serif, proportional, italicized font, as in */usr/bin*.

Please note that general references to application, library, module, and package names are not typeset any differently from regular text. Thus, references to the matplotlib package are typeset just as in this sentence. As most packages have unique names, this should not be confusing. In the few cases where the package names are regular English words (e.g., the time module), references to the module will hopefully be clear from the context.

Usually, the first time a key word is used and/or explained, it will be bold in the text **like this.** Key words are found in the glossary, and when useful, occurrences of those words are hyperlinked to the glossary. Many acronyms are hyperlinked to the acronym list. The glossary and acronym lists start on p. 126.

All generic text is in black. All hyperlinks (whether to locations internal or external to the document), if provided, are in blue.

Finally, because the focus on this book is on how to do MIS using Python, not on the Python language itself, when I do focus on the nitty-gritty of the Python language as a language, it will be in sections with headers beginning with "⌨ Python Sidebar." The symbol in front is a keyboard ☺. This will make it easier to find these sections in the Table of Contents, if you're looking only for the parts where I focus in on Python syntax and structure.

Updates to this book

The book's website, http://www.johnny-lin.com/infosys, contains any updates, links to resources, etc. available for the book. This includes a link to a list of addenda and errata.

Personal Acknowledgments

While I often use first person throughout this book, I am acutely aware of the debt I owe to family, friends, and colleagues who, over many years, generously nurtured many of the ideas in this book: Indeed, we all do stand on the shoulders of giants, as Newton said. All praise I happily yield to them; any mistakes and errors are my own. Much of this book is based on my book, *A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences,*[1] and the acknowledgments I made there equally apply to this text.

[1]Lin (2012)

In addition to those people, I also want to acknowledge Hannah Aizenman for her contributions to the present work. Thanks to Evangeline Abrigo for correction help and to students who provided anonymous feedback.

I am personally grateful for those who gave me permission to use material they created: These are acknowledged in the Notices section starting on p. viii and in the captions of the included or adapted figures.

Finally, I thank my wife Karen, my children Timothy, James, and Christianne, for their encouragement and love, and my Lord and Savior Jesus Christ for giving me life itself, both physically and spiritually: "… I have come that they may have life, and have it to the full" (John 10:10b, NIV).

Johnny Wei-Bing Lin
Bellevue, Washington
August 30, 2019

Notices

Trademark Acknowledgments

Copyright Acknowledgments

Part I

Basic Business Data Analysis

Chapter 1

Python as a Basic and Business Calculator

Chapter Contents

Python is a **multi-paradigm language**, which is a computer science-ese way of saying that you can use it in a bunch of ways. The simplest (and sometimes most useful) way is as a calculator. In fact, when I'm on my laptop, I don't usually bother with firing up a calculator app if all I need is to do some simple (or not-so-simple) arithmetic. Instead, I start a Python session and just type in what I want to calculate. In the process, if I need to do more heavy-duty calculations, I have all the power of Python's mathematical and statistical libraries at my disposal.

1.1 Starting a Python interpreter session

Python is an interpreted language, meaning that you just type in a command in Python's **interpreter**, press Enter, and Python will execute that command right then and there. It's not like a compiled language like Java where you have to write the code, process the code using the compiler, then run the code. In Python, it's instant gratification ☺.

But, you have to first start up the Python interpreter. I'll describe two ways of getting Python's interpreter up and running: Using the Canopy environment's **Interactive Development Environment (IDE)** and running a session in a **terminal window**.

1.1.1 The Canopy IDE

Canopy comes with its own IDE consisting of a code editor and an interpreter environment. (Though, you don't have to use Canopy's IDE with Canopy's version of Python.) For those who have used other IDEs such as BlueJ (for Java), you're used to how this works. You start up the IDE, open a window to edit code, and compile and run the code from the IDE application (usually a menu choice). In a Python IDE, you do the same thing except you don't have to compile. Once you write the code, you just select the menu option to run the code and an interpreter opens up and runs the code.

When you use Python as a calculator, you don't even have code to run, so instead of writing code then running it, use the IDE to start an interpreter and type the code in the interpreter; pressing Enter will run the code. Here's how it works in Canopy:

1. Start Canopy the usual way, by finding the application on your computer and double clicking it to start it. You should get a welcome window similar to Figure 1.1. Click the Editor button in the welcome window.

2. Next, you'll get a window that says "Create a new file" or "Select files from your computer". Instead, select the menu choice View → Python. In the bottom window, you'll get a Python interpreter, as shown in Figure 1.2.

3. In the interpreter, you can type in Python commands where it says `In` (e.g., `In [1]`). When you press Enter, the Python calculation run and the result is printed out after the `Out` prompt (e.g., `Out [1]`). Figure 1.3 gives an example.

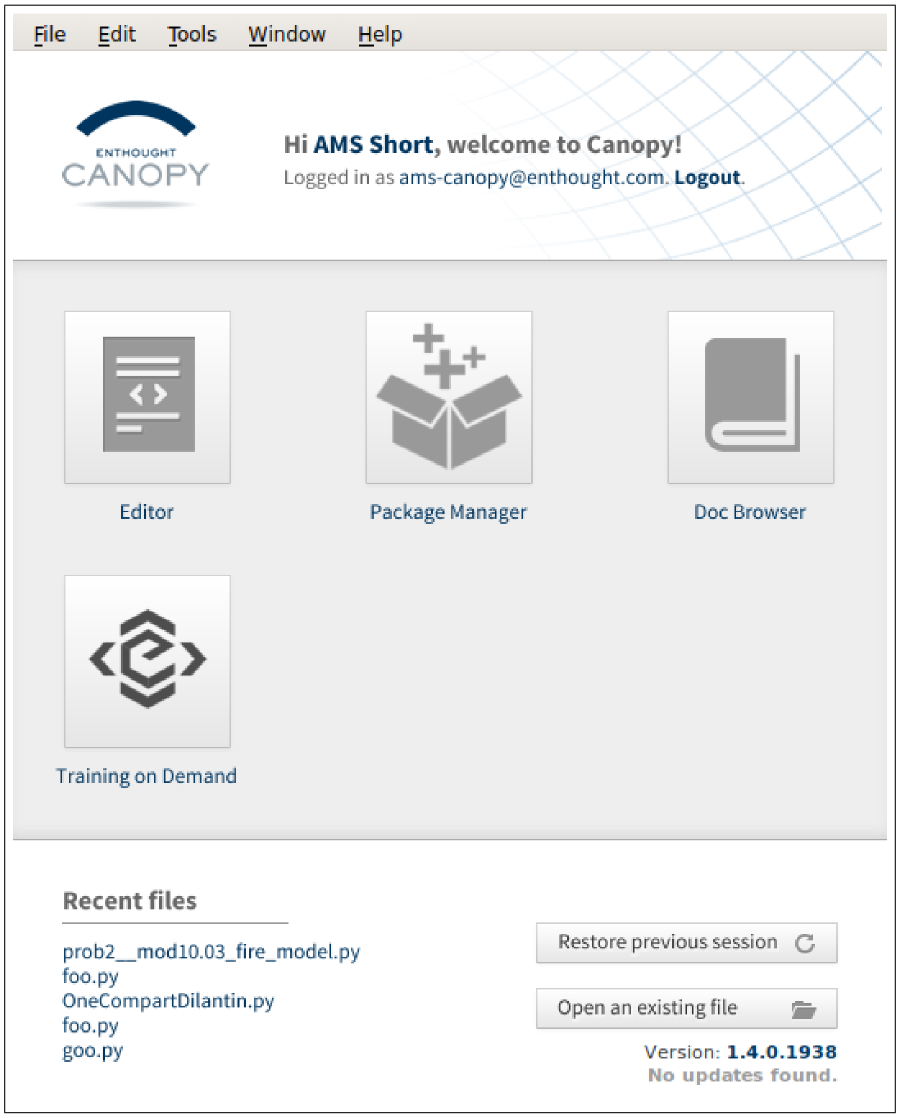
One final note: We'll find later on when we discuss saving and reading files and modules that the default location where we save and read files is somewhere called the "**current working directory**." In Canopy, that location is set by a drop-down menu in the upper-right corner of the Python shell, as shown in Figure 1.4.

1.1.2 Using a terminal window

When you log in to your Windows or Mac OS X computer what do you see? Chances are, a whole bunch of icons. Those icons represent files or programs and most of us are used to double-clicking them in order to open a file or run a program.

That's not how people used to use computers. In the olden days, the computer presented a cursor where you could type in a command (e.g., `mkdir NewDirectory`) that told the computer what you wanted it to do, in the previous example, making a directory called *NewDirectory*.

You can still do this today. To do so, you first open up a terminal window, which gives puts a window on your desktop that has a cursor awaiting you to type in your command to the computer.



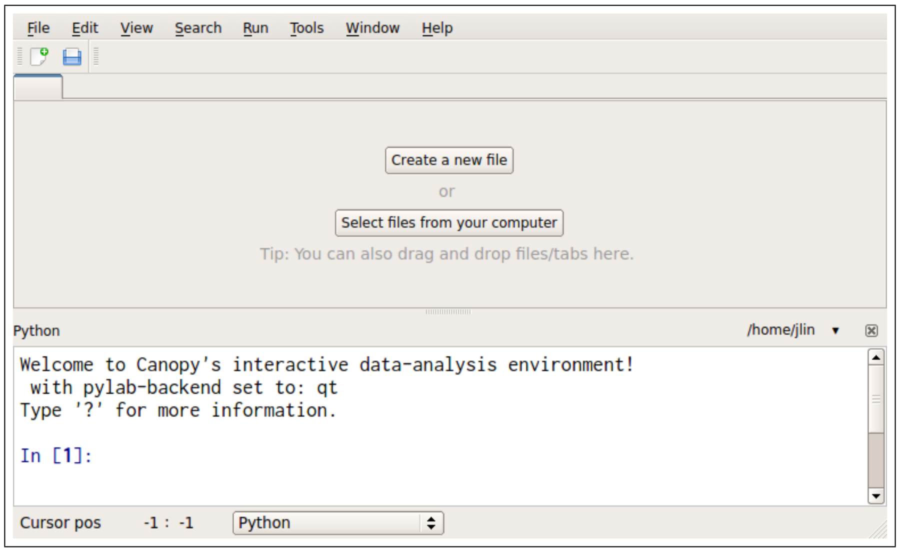
Hi AMS Short, welcome to Canopy!
Logged in as ams-canopy@enthought.com. Logout.

Recent files

prob2__mod10.03_fire_model.py
foo.py
OneCompartDilantin.py
foo.py
goo.py

Version: 1.4.0.1938
No updates found.

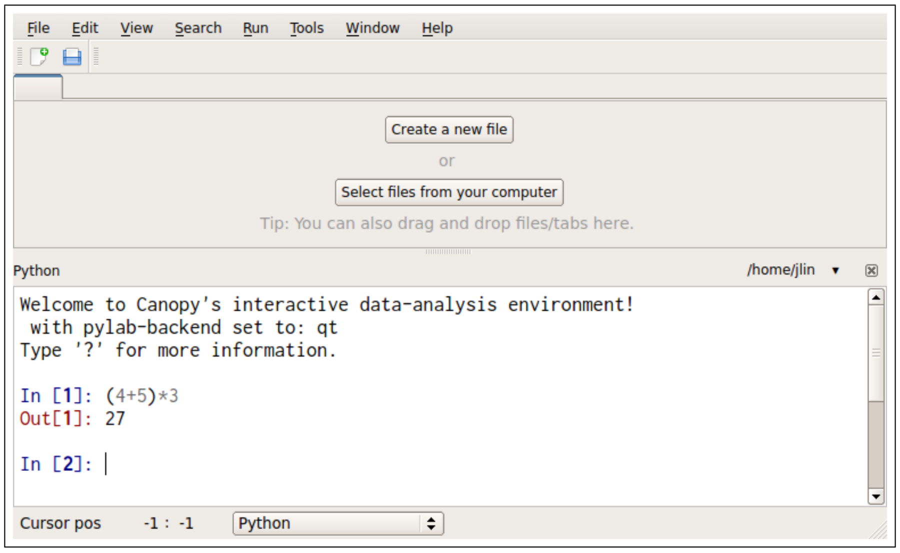
Figure 1.1: The Canopy welcome window.


File Edit View Search Run Tools Window Help
Create a new file
or
Select files from your computer
Tip: You can also drag and drop files/tabs here.
Python
/home/jlin
Cursor pos -1 : -1
Python

```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [1]:
```

Figure 1.2: The Canopy Python interpreter.

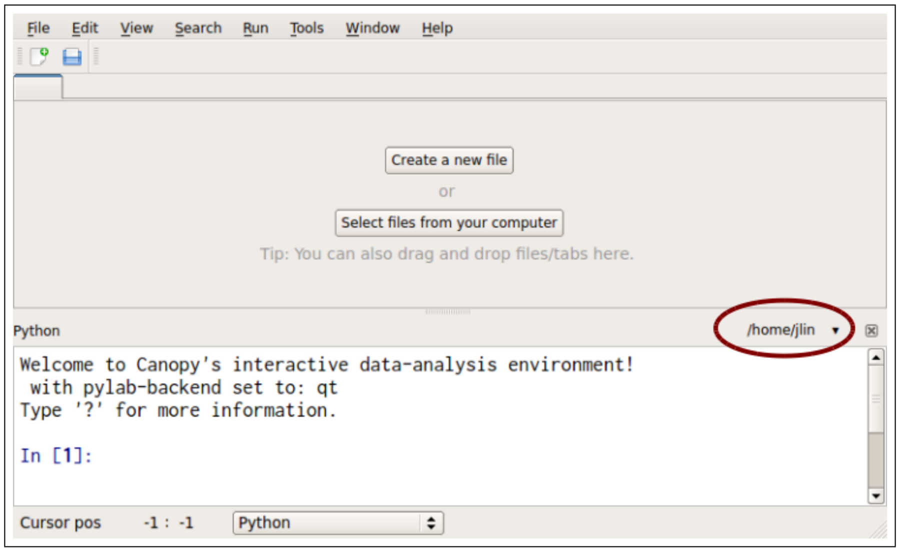

File Edit View Search Run Tools Window Help
Create a new file
or
Select files from your computer
Tip: You can also drag and drop files/tabs here.
Python
/home/jlin
Cursor pos -1 : -1
Python

```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [1]: (4+5)*3
Out[1]: 27

In [2]:
```

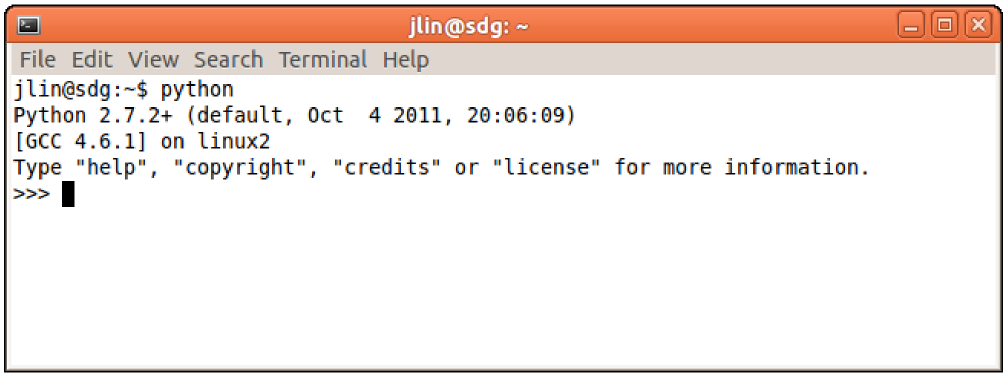
Figure 1.3: Using the Canopy Python interpreter as a simple calculator.



```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [1]:
```

Figure 1.4: Canopy's current working directory and drop-down menu to control changing the current working directory is circled in red.



```
jlin@sdg:~$ python
Python 2.7.2+ (default, Oct  4 2011, 20:06:09)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 1.5: Starting the Python interpreter in a terminal window.

Terminal windows are created on a Mac OS X computer by running the Terminal application and on a Linux machine by running xterm, GNOME Terminal, or any of a number of other terminal creating applications. On a Windows computer, you'd start an MS-DOS shell.

Why would you want to do this? We'll talk more about this later when we discuss automating Management Information Systems (MIS) tasks. For now, using a terminal is a quick way of getting access to the Python interpreter. In contrast with starting up and IDE, which sometimes takes a while, a terminal window usually pops up quickly.

To start the Python interpreter using a terminal window, after you've opened that window, do the following (everything you type will be in that window):

1. Type `python`. You should get something that looks like Figure 1.5. If this doesn't happen, here are some possible fixes:

    - You may have to type in the full path name to your Python application. On my machine,

Operation	**Symbol**
Add	$+$
Subtract	$-$
Multiply	$*$
Divide	$/$
Exponentiation	$**$

Table 1.1: Arithmetic operators.

my Anaconda Python is at `/home/jlin/anaconda/bin/python`.

- Information regarding the Canopy Python path is available here: https://support.enthought.com/hc/en-us/articles/204469730-Make-Canopy-User-Python-be-your-default-Python-i-e-on-the-PATH-.
- On some installations, you may have to type in the version of Python you want, e.g., `python2.7`.

The three greater-than signs (>>>) on the left of the line tells you you are now in the Python interpreter.

2. An alternate way of opening a terminal that you can use to run Canopy is to open the Canopy application and then select the Tools → Canopy Terminal menu option. This will open up a terminal window that has Canopy set as the default Python. If you type `python` in that terminal window, you'll launch Canopy's Python.[1]

3. You can now type in whatever you want to calculate, press Enter, and the answer will output. For instance:

```
>>> (4+5)*3
27
>>>
```

4. The interpreter immediately executes the command, printing the string `hello world!` to screen.

5. To exit the Python interpreter, type Ctrl-d. To exit from the terminal window, type `exit` once you're outside the Python interpreter. Typing `exit` while you're in the Python interpreter will do nothing but tell you what you need to do to leave the Python interpreter.

1.1.3 ⌨ Python Sidebar: Arithmetic operators in Python

Table 1.1 lists the arithmetic operators Python uses: Parentheses are the normal characters and arith-

[1]https://support.enthought.com/hc/en-us/articles/204469730-Make-Canopy-User-Python-be-your-default-Python-i-e-on-the-PATH- (accessed September 1, 2016).

metic operation order in Python follows the standard order of parenthetical blocks first, then exponentiation, then multiplication/division, and finally addition/subtraction. Operations at the same level of priority are executed left-to-right. See https://en.wikibooks.org/wiki/Python_Programming/Operators for more details.

Note that in Python 2.7.x, division of two integers will default to integer division. That is, it will return the quotient and discard the remainder and the output will be of integer type:

```
>>> 1/2
0
>>> 3/2
1
>>> 7/3
2
>>> type(7/3)
<type 'int'>
```

To force Python to use the kind of division we're all used to (floating point division), make at least one of the numbers floating point by putting a period (or a period and a decimal) after it:

```
>>> 1./2
0.5
>>> 1.0/2
0.5
```

In Python 3.x, the default is to include the remainder in the output, with the output being floating point type:

```
>>> 1/2
0.5
>>> 3/2
1.5
>>> 7/3
2.3333333333333335
>>> type(7/3)
<class 'float'>
```

To do integer division in Python 3.x, use "//" instead of "/".[2] Note that the // operator also works as integer division in Python 2.7.x.

1.2 Using functions

Any scientific, business, or financial calculator worth its salt does more than arithmetic. These calculators have specialized functions that enable you to quickly calculate quantities such as sines, logarithms, and future values.

[2]http://stackoverflow.com/a/5365702 (accessed September 2, 2016).

Python also has such function, but instead of typing in some numbers then pressing a button, you type in the name of these functions to execute them. Thus, to take the sine of $\pi/2$ radians (i.e., 90°):

```
>>> sin(3.1415/2)
0.99999999892691405
```

The sine function is called `sin` and the input into the function is put in between two parentheses immediately after the name of the function. After you press Enter, the answer or result of **calling** the sine function is returned.

But wait, don't type that in! What I wrote above is *not* what will happen; what will happen if you type in what I wrote above is:

```
>>> sin(3.1415/2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
```

Hmmm, so Python doesn't know anything about the sine function. The sine function, it turns out, is not built-in into the interpreter but rather is part of a **module** called SciPy (the name used in the interpreter is "scipy", without capitalization). In order to use `sin` I first need to import it. If I add an import line before the call to `sin`, everything works fine:

```
>>> from scipy import sin
>>> sin(3.1415/2)
0.99999999892691405
```

If there is more than one input in a function call, each input is listed one at a time, separated by commas. Each input is called a "**parameter**" (or "positional input parameter") and the sequence of multiple inputs is called a "**parameter list**." The `fv` function in SciPy which calculates future value takes four required inputs, in this order: the interest rate, the number of compounding periods, the payment paid (by default) at the end of each period, and the present value.[3] Thus, if you have an investment account with an annual interest rate of 3% (compounded annually), worth $1000 now, and you add $500 every year for ten years, how much will have you at the end of the ten years?

```
>>> from scipy import fv
>>> fv(0.03, 10, -500, -1000)
7075.8560350794924
```

Note the negative sign means "cash flow out (i.e. money not available today)."[4]

By the way, if you want to learn more about a function, you can use the `help` command. In the case of `fv`, you can type in `help(fv)` in the Python interpreter and Python will tell you about the function and how to use it. If you are running the interpreter from the command-line, type Space to page down the help display, `j` to scroll down the display, `k` to scroll up the display, and `q` to leave the display.

[3] http://docs.scipy.org/doc/numpy/reference/generated/numpy.fv.html (accessed September 1, 2016).
[4] http://docs.scipy.org/doc/numpy/reference/generated/numpy.fv.html (accessed September 1, 2016).

1.2.1 ⌨ Python Sidebar: Optional input into functions

In the future value function (`fv`) example above, we noted that by default, the payment paid each period is assumed to be paid at the end of each period. For this function, Python enables you to set a the payment to occur at the beginning of the period instead of the end, and if you do not explicitly set that differently, it defaults to the end of the period. Python accomplishes this through **keyword input parameters.**

In a regular parameter, the function knows what each value in the parameter list corresponds to based upon the position of the parameter. Thus, for instance, in a call of the `fv` function, the function knows the second parameter will be the the number of compounding periods. Keywords are set using the assignment syntax; if a function has a keyword input parameter called `k` you want set to 4, you do so by adding `k=4` in the calling line.

Thus, if we wanted to change our previous future value problem to have payments happen at the beginning of the periods, the code would be:

```
>>> from scipy import fv
>>> fv(0.03, 10, -500, -1000, when='begin')
7247.8142247515543
```

In this case, `when` is the keyword input parameter and we set it to the string `'begin'`. (We'll talk more about strings in Section 1.3.2.) In Section 1.3, we talk more about setting variables.

Keyword input parameter's are nice for inputs to functions that are optional rather than required. Since they have a default setting, you only need to specify them in the function's calling line if you want to pass in a value different than the default. (Positional input parameters are used for required inputs.) The function's documentation will tell you what is required (positional) and what is optional (keyword) input.

1.2.2 ⌨ Python Sidebar: Importing modules and using module items

In the `fv` example above, we imported the function by typing in:

`from` <*module*> `import` <*function*>

where *module* is the name of the module the function resides in and *function* is the name of the function we're interested in being able to use from the *module*.

But what if we wanted to make use of more than one function in a module? Would we have to use `from...import...` everytime we wanted access to the function? Thankfully, no ☺! The following import command:

`import` <*module*>

will make all the functions of the module available by typing <*module*>`.`<*function*>. That is:

```
>>> import scipy
>>> scipy.fv(0.03, 10, -500, -1000, when='begin')
7247.8142247515543
```

is the same as:

```
>>> from scipy import fv
>>> fv(0.03, 10, -500, -1000, when='begin')
7247.8142247515543
```

If we are lazy typists, we can use:

`import` *<module>* `as` *<alias>*

to make an even shorter reference to the module. That is:

```
>>> import scipy as sp
>>> sp.fv(0.03, 10, -500, -1000, when='begin')
7247.8142247515543
```

is the same as:

```
>>> import scipy
>>> scipy.fv(0.03, 10, -500, -1000, when='begin')
7247.8142247515543
```

Finally, modules do not only contain functions but may also contain variables set to certain values. For instance, the SciPy module has a variable called `pi` that is set to the value of π. When you import a module using `import`, you can reference the variables defined in the module in a way similar to how you reference functions, by using the period notation:

```
>>> import scipy
>>> scipy.pi
3.141592653589793
```

Notice that when we make use of `pi`, we do not put parenthesis after the variable name whereas when we use a function (like `fn`), we do put parenthesis after the function name. This is because with the function, we are *calling* the function. Python conveys this through appending the parameter list (in a pair of parentheses) to the function name. In the case of `pi`, because it's a variable, it isn't being called. (Remember that calling a function means giving input to a function, running the function, and getting whatever's returned.) In Section 1.3, we talk more about setting variables,and when we talk about objects in more detail on Section 5.3, we'll see some more of how this period notation is used.

1.2.3 ⌨ Python Sidebar: Writing and using your own functions

So far we've talked about how to access functions someone else has written. What if we want to use functions we've written? Here we'll describe two ways of doing so.

First, we can define the function in the interpreter session. Here is an example of a function `percent_to_decimal` which converts a percentage into its decimal or ratio form:

```
>>> def percent_to_decimal(input):
...     output = input / 100.
...     return output
...
>>> percent_to_decimal(42)
0.42
```

When defining a function, the first line begins with `def`, followed by the function name, then the parameter list. The number of variables in the parameter list is the number of parameters we need to provide when we call the function. In the above example, there is one parameter named `input`. All references in the body of the function definition with the name `input` refers to whatever value is passed into the function when it is called. The `def` line ends with a colon.

The second thing to note in the above function definition is that the body of the definition is delineated by an indentation of four spaces (which pressing Tab will give me when I'm in the Python interpreter). The "`...`" characters are automatically included by the interpreter as I'm typing; they just mean that Python recognizes you're typing in a function definition and is waiting for you to provide the body of the function.

Finally, in the example we see that the return value of a function is given by the `return` followed by whatever is being returned. In the above example, the variable `output` is returned, but the return value can itself be an expression. For instance, the code below works exactly the same:

```
>>> def percent_to_decimal(input):
...     return input / 100.
...
>>> percent_to_decimal(42)
0.42
```

and saves us an extra line of typing. (Note that we make the `100.` a decimal to ensure we never have integer division operating on `input`.

Typing in a function in an interpreter is easy and straightforward but as soon as you exit the interpreter, the function definition is lost. Thus, it often makes sense to instead define a function in a file and then use `import` to give you access to the function.

In our `percent_to_decimal` example, we put the code we would have typed into the interpreter in a file called *myfuncs.py*, as seen in Figure 1.6. Then, as long as *myfuncs.py* is in the current working directory for the Python interpreter,[5] I can import *myfuncs.py* and use its contents:

```
>>> import myfuncs
>>> myfuncs.percent_to_decimal(42)
0.42
```

But wait, isn't `import` used for importing modules? Well, yes, it is. But what this shows us is that a module is just a file containing Python commands. If you have a bunch of function, variable,

[5]The more complete answer is that *myfuncs.py* has to be on the path specified by the *PYTHONPATH* environment variable, but for our purposes right now, as long as *myfuncs.py* is in the same directory we started `python` in, it'll work.

```
def percent_to_decimal(input):
    return input / 100.
```

Figure 1.6: Contents of *myfuncs.py*.

or **class** definitions, just put them into a file and voilà, you have a module you can import and use! It's that easy to create a library of your own functions.[6]

1.2.4 Python Sidebar: Keyword input parameters

In Section 1.2.1, we introduced using keyword input parameters in a function call. How do we define keyword input parameters in functions we write? Let's say we wanted to write a function `annual_percent_to_period_decimal` that converts an annual percentage rate to a decimal for a given sub-annual period (e.g., a week, month, etc.). Most of the time, we expect function the sub-annual period will be a month, but we want to be able to change that period if it's not a month. We create a keyword input parameter `num_periods_per_year` to define the sub-annual period. By default, `num_periods_per_year` will equal 12, i.e., that the sub-annual period will be a month. The code is then:

```
def annual_percent_to_period_decimal(input, num_periods_per_year=12):
    return input / 100. / num_periods_per_year
```

If we call `annual_percent_to_period_decimal` without specifying `num_periods_per_year`, the function will do the calculation assuming `num_periods_per_year` is 12:

```
>>> annual_percent_to_period_decimal(6)
0.005
```

On the other hand, if we want to specify a different value for `num_periods_per_year`, we can do so in the call to `annual_percent_to_period_decimal`. In the case below, we assume the sub-annual period is a half-year:

```
>>> annual_percent_to_period_decimal(6, num_periods_per_year=2)
0.03
```

1.3 Saving values

On a calculator, you save values in memory using an "M+" key or something similar. You can usually only save one value or add to a value previously saved. In Python, you save values by setting them to variable names. You can then use the values by name. For instance:

[6]This is in contrast, for instance, with Fortran where you have to compile your function's source code into an object file, create a library from multiple object files, and link other compiled code to the library file to make use of the functions defined in the library. Whew! Even describing the Fortran process sounds horrible ☺!

```
>>> a = 2
>>> 3*a
6
>>> b = 5
>>> a*b
10
```

To see what the contents of a variable are, just type in the name of the variable:

```
>>> a = 2
>>> a
2
```

You can have any number of variables. You do not need the blank spaces on either side of the equal sign. Thus, `a = 2` and `a=2` work equally well (pun intended ☺).

1.3.1 ⌨ Python Sidebar: Naming, creating, and deleting variables

When using Python as a calculator, you probably won't have too many variables, so it's probably fine to use a single letter for your variable names. But, as you might expect, it makes more sense to name variables something descriptive, such as `rate_of_return`, in order to make it easier to understand your code.

Python variable names must begin with a letter. Python is also case-sensitive, so the variable name `Rate` and `rate` refer to two different variables.

Generally speaking, when you have multiple words in a Python variable name, you separate them by underscores rather than capitalizing the first letter of each word after the first (as is the convention in Java). That is, Python variables are usually written `rate_of_return` instead of `rateOfReturn`. Class names follow the CapWords convention, where every word in the name is capitalized. These are all conventions, however, so your program will work fine even if you choose a different naming convention. All style conventions for Python are given in PEP 8 (https://www.python.org/dev/peps/pep-0008/), the official style guide for Python code. ("PEP" stands for "Python Enhancement Proposal.")

Folks who come to Python from a compiled language like Java or C++ will notice that when we create a variable we don't have to declare the variable's type. That is, instead of the following in Java:

```
int a = 4;
```

in Python you just write:

```
a = 4
```

In contrast with Java, Python is a **dynamically typed** language, meaning that the type of a variable can change with time. The type of a variable is automatically set by Python based upon whatever is on the left-hand side. Thus, in the code below:

```
>>> a = 3
>>> type(a)
<type 'int'>
>>> a = 6.4
>>> type(a)
<type 'float'>
>>> a = "hello"
>>> type(a)
<type 'str'>
```

we see from our calls to the `type` function that the type of the variable `a` changes each time we assign `a` to a new value.

If you want to delete a variable so that Python doesn't recognize it, use the built-in `del` command:

```
>>> a = 3
>>> a
3
>>> del(a)
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

1.3.2 ⌨ Python Sidebar: A quick introduction to strings

We've seen integers and floating point numbers. Strings are combinations of characters that are set between matching apostrophes, quotation marks, or triple apostrophers/quotation marks. You can set a variable to them just as with numbers:

```
>>> a = 'hello there'
>>> a
'hello there'
>>> type(a)
<type 'str'>
```

Note that all of these will give you a string:

```
'hello there'
"hello there"
"""hello there"""
'''hello there'''
```

So, why are there multiple ways of specifying a string? Having both single quotes and quotation marks is useful to when your string needs to include the other kind of punctuation mark:

```
>>> a = "She'll be coming 'round the mountain when she comes."
>>> a
"She'll be coming 'round the mountain when she comes."
>>> b = '"Who is that?" he asked.'
>>> b
'"Who is that?" he asked.'
```

The triple quotes allow you to include newline characters and spaces through typing Enter and pressing the space bar and having those remembered as part of the string. This makes it easier to enter in more complexly formatted strings:

```
>>> a = '''Toy boat.
...     Toy boat.
... I said, "Toy boat."'''
>>> a
'Toy boat.\n    Toy boat.\nI said, "Toy boat."'
>>> print(a)
Toy boat.
    Toy boat.
I said, "Toy boat."
```

The character "`\n`" is the newline character. Tab is "`\t`". The `print` function prints out the contents of the string variable to the screen but represents formatting characters (such as newline) as they way they should look.

Two final points about strings to mention, and then we'll leave more about strings to the discussion in Section 5.3.2. First, if you want to concatenate two strings together, use the "+" operator:

```
>>> a = "Boeing"
>>> b = "Airbus"
>>> a + " or " + b
'Boeing or Airbus'
```

Second, if you want to convert a number into a string, use the built-in `str` function:

```
>>> a = "Boeing"
>>> b = "Airbus"
>>> i = 1
>>> a + " or " + b + " is number " + str(i)
'Boeing or Airbus is number 1'
>>> a + " or " + b + " is number " + i
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

If you leave out the `str` call, Python doesn't know how to "add" together a string and an integer, and so complains to you.

```
import myfuncs
myfuncs.percent_to_decimal(42)
myfuncs.percent_to_decimal(2)
myfuncs.percent_to_decimal(83)
myfuncs.percent_to_decimal(12)
```

Figure 1.7: Contents of *myscript1.py*.

1.3.3 Python Sidebar: A programmable calculator

In Section 1.2.3, we saw how we could write (and use) our own functions in our Python calculator. We also saw that all we had to do was to copy-and-paste what we typed at the interpreter prompt and put it in a file, then import that file to use the functions defined in the file (i.e., module). We'll now see that we can use this copy-and-paste methodology (with minor adjustments) to "record" our entire calculator session, including the setting of variables and the use or calling of functions, so that we can run the calculator session again. This record of what would have been an interactive calculator session is often referred to as a "**script**" and the writing of those lines of code "**scripting**.." The neat thing about scripting is that we can use the script over-and-over again, without typing it in again.

Let's say we have four percentages that we wish to convert to decimals using our `percent_to_decimal` function (that is defined in *myfuncs.py*, in Figure 1.6). In an interpreter we would type this in:

```
>>> import myfuncs
>>> myfuncs.percent_to_decimal(42)
0.42
>>> myfuncs.percent_to_decimal(2)
0.02
>>> myfuncs.percent_to_decimal(83)
0.83
>>> myfuncs.percent_to_decimal(12)
0.12
```

But if we exited the interpreter and wanted to later redo these four recalculations, we'd have to type it in again. So, let's copy-and-paste these lines into a file, as seen in Figure 1.7. (We don't copy-and-paste the output or the interpreter prompt >>>.) We'll call this file *myscript1.py*.

To run the script from the terminal, type in:

```
python myscript1.py
```

(Using Canopy, to do the same thing, you would first open the file *myscript1.py* and then choose the menu item Run → Run File. Or, you can just click the green arrowhead icon in the task bar.)

But nothing happens! The results of the `percent_to_decimal` calls are not printed to screen! This is one of the adjustments you have to make when writing a script instead of typing in commands at the interpreter prompt. In the interpreter, when you type the name of a variable or call a function, the value of the variable or the function's return value are automatically displayed to the screen. In

```
import myfuncs
print(myfuncs.percent_to_decimal(42))
print(myfuncs.percent_to_decimal(2))
print(myfuncs.percent_to_decimal(83))
print(myfuncs.percent_to_decimal(12))
```

Figure 1.8: Contents of *myscript2.py*.

```
def percent_to_decimal(input):
    return input / 100.

print(percent_to_decimal(42))
print(percent_to_decimal(2))
print(percent_to_decimal(83))
print(percent_to_decimal(12))
```

Figure 1.9: Contents of *myscript3.py*.

a script, this does not occur. In order to see the contents of the return value, you have to pass the function call as an argument to the `print` function, as seen in the *myscript2.py* file in Figure 1.8.

Now, when you run the revised script, the expected output is produced:

```
$ python myscript2.py
0.42
0.02
0.83
0.12
```

Notice that when we run the script in the terminal, we automatically leave the Python interpreter at the end of running the script. If you want to stay in the interpreter at the end of running the script, add a "`-i`" between `python` and the script filename, as in:

```
$ python -i myscript2.py
0.42
0.02
0.83
0.12
>>>
```

Finally, if we don't want to keep the `percent_to_decimal` function in a separate file from our script, that's fine too. We can define functions and use them in the same script, as well as setting variables, etc. By putting the function in our script, we also do not need to import the myfuncs module. Figure 1.9 shows this revised script, *myscript3.py*.

1.4 Summary

Well, that's it! We now know how to use Python as a really fancy programmable business and financial calculator! As we'll see later on in this book, this is only a fraction of the powers of Python, but it's still very useful nonetheless!

Chapter 2

X-Y Plots

Chapter Contents

It is easy to underestimate the power of graphs. When we first learn about graphing, it's easy to think graphs are "merely" picture representations of functions or equations. In reality, graphs or visualizations are a key way of enabling us to understand what the data is telling us. If you have millions of point-of-sale customer transactions you are analyzing, it is unlikely you can find a single function to describe that data (or that that function will tell you much about the patterns in the data). If you need to display the raw data, you either have to do so in a table or in a graph that provides you insight as to the meaning of the data. Thus, graphs are essential analytical tools for business data.

Before we begin, credit where credit is due: I owe the matplotlib documentation (http://matplotlib.org) for a lot of the material in this chapter.

2.1 Matplotlib: Python's basic plotting package

Python's basic plotting **package** is called matplotlib. It consists of a main module and a bunch of submodules. The submodule pyplot defines the functional interface for matplotlib. Pyplot is often imported by:

```
import matplotlib.pyplot as plt
```

Unless otherwise stated, you may assume in the examples in this chapter that the above import has been done prior to any matplotlib calls being run.

The online pyplot tutorial is very good. After we've covered a little more about Python, I'd encourage you to go through it all on your own: http://matplotlib.org/users/pyplot_tutorial.html. The online gallery of examples is also very illuminating: http://matplotlib.org/gallery.html.

2.2 My first X-Y plot

We'll look at two basic x-y plots: a line plot (where the points are connected together) and a scatter plot (where the points are not connected together).

2.2.1 A line plot

Let's say you have the following list of point-of-sale customer transactions:

Decimal Hour in Day	Amount ($)
9.25	2.54
11	4.10
13.5	1.21
15	3.90
15.75	4.00

(A decimal hour representation of 1:30 pm is 13.5.) The following will create a line plot of amount versus decimal hour in day:

```
1 import matplotlib.pyplot as plt
2 plt.plot([9.25, 11, 13.5, 15, 15.75], [2.54, 4.1, 1.21, 3.9, 4])
3 plt.xlabel('Decimal Hour in Day')
4 plt.ylabel('Amount ($)')
5 plt.show()
```

The graph created is shown in Figure 2.1. You can either type in each line of code in a Python interpreter or in a file and run the script file (see Section 1.3.3 on how to write and run a script). If you do run the lines of code as a script from a terminal window, you have to keep the interpreter session open in order to see the plot.

Based on what the lines of code above say, what do you think each line does? Here's my description:

- Line 1: Import the pyplot module.

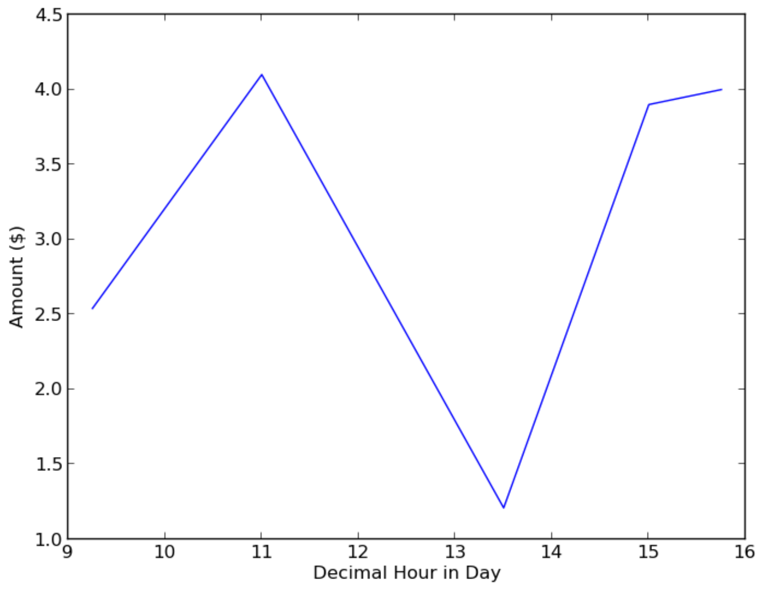


Figure 2.1: Graph created by the code in Section 2.2.1.

- Line 2: Create the plot. The first **argument** is the list of x-values (the decimal times) and the second argument is the list of the corresponding y-values. See Section 2.3 for a discussion of what is a list.
- Line 3: Write the x-axis label.
- Line 4: Write the y-axis label.
- Line 5: Show the plot.

Once a plot is created (line 2), matplotlib keeps track of what plot is the "current" plot. Subsequent commands (e.g., to make a label) are applied to the current plot.

Line 5 indicates that as you put each element of a graph onto matplotlib's virtual canvas, matplotlib does not render your addition but instead waits until you call the `show` command. Usually, you care only about how a plot looks when it's all done. By waiting until `show` is called, matplotlib avoids the extra computation involved in rendering intermediate steps. (If you have more than one figure, call `show` after all plots are defined to visualize all the plots at once.)

Matplotlib does pretty well using intelligent defaults for the graphs you ask it to make. But much of the time, we'll want to customize what our graphs look like. We'll talk about such customization in a second, but first we take a side-trip to introduce Python lists.

2.2.2 A scatter plot

Let's take the same data in Section 2.2.1 and make a scatter plot. This code will do the trick:

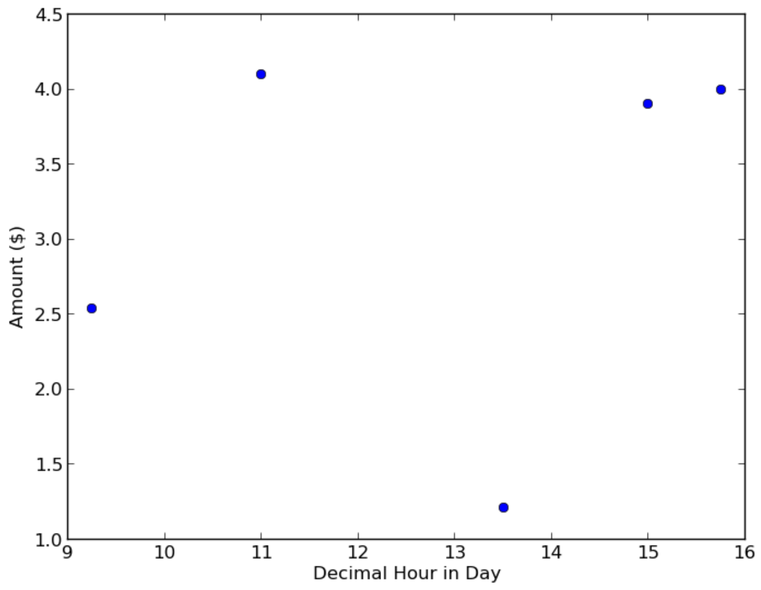


Figure 2.2: Graph created by the code in Section 2.2.2.

```
import matplotlib.pyplot as plt
plt.plot([9.25, 11, 13.5, 15, 15.75], [2.54, 4.1, 1.21, 3.9, 4], \
     linestyle=' ', marker='o')
plt.xlabel('Decimal Hour in Day')
plt.ylabel('Amount ($)')
plt.show()
```

The `linestyle` keyword is set to an empty space while the `marker` keyword input parameter is set to an `"o"`, the symbol for a circle plot marker. The graph created is shown in Figure 2.2.

One final note: In the code above, the end of line 2 terminates with a backslash ("\") character. The backslash at the very end of a line is Python's way of noting when a line continues on on the next line. Thus:

```
a = "Hello " + \
    "there " + \
    "everyone!"
```

is the same as:

```
a = "Hello " + "there " + "everyone!"
```

2.3 Python Sidebar: Lists and tuples

We've seen a few times now this Python construct we call a "list." In the times we've seen it, it appears to be a sequence of numbers. It can be, but it's more than that.

Lists are ordered sequences. What that means is they are a collection of items where the first item has the first position, the second the second position, and so on. They are like arrays, except each of the items in the list do not have to be of the same type. A given list element can also be set to anything, even another list. Square brackets ("`[]`") **delimit** (i.e., start and stop) a list, and commas between list elements separate elements from one another.

List element addresses start with zero, so the first element of list `a` is `a[0]`, the second is `a[1]`, etc. Because the ordinal value (i.e., first, second, third, etc.) of an element differs from the address of an element (i.e., zero, one, two, etc.), when we refer to an element by its address we will append a "th" to the end of the address. That is, the "zeroth" element by address is the first element by position in the list, the "oneth" element by address is the second element by position, the "twoth" element by address is the third element by position, and so on.

Finally, the length of a list can be obtained using the built-in `len` function, e.g., `len(a)` to find the length of the list `a`.

Example 1 (A list):

Type in the following in the Python interpreter:

```
a = [2, 3.2, 'hello', [-1.2, 'there', 5.5]]
```

What is `len(a)`? What does `a[1]` equal to? How about `a[3]`? `a[3][1]`?

Solution and discussion: The `len(a)` is 4, `a[1]` equals 3.2, `a[3]` equals the list `[-1.2, 'there', 5.5]`, and `a[3][1]` equals the string `'there'`. I find the easiest way to read a complex reference like `a[3][1]` is from left to right, that is, "in the threeth element of the list `a`, take the oneth element."

In Python, list elements can also be addressed starting from the end; thus, `a[-1]` is the last element in list `a`, `a[-2]` is the next to last element, etc.

You can create new lists that are slices of an existing list. Slicing follows these rules:

- Element addresses in a range are separated by a colon.
- The lower limit of the range is *inclusive,* and the upper limit of the range is *exclusive.*

Example 2 (Slicing a list):

Consider again the list `a` that you just typed in for Example 1. What would `a[1:3]` return?

Solution and discussion: You should get the following if you print out the list slice `a[1:3]`:

```
>>> print a[1:3]
[3.2, 'hello']
```

Because the upper-limit is exclusive in the slice, the threeth element (i.e., the fourth element) is not part of the slice; only the oneth and twoth (i.e., second and third) elements are part of the slice.

Lists are **mutable** (i.e., you can add and remove items, change the size of the list). One way of changing elements in a list is by assignment (just like you would change an element in an array):

Example 3 (Changing list element values by assignment):
Let's go back to the list in Example 1:

```
a = [2, 3.2, 'hello', [-1.2, 'there', 5.5]]
```

How would we go about replacing the value of the second element with the string `'goodbye'`?

Solution and discussion: We refer to the second element as `a[1]`, so using variable assignment, we change that element by:

```
a[1] = 'goodbye'
```

The list `a` is now:

```
[2, 'goodbye', 'hello', [-1.2, 'there', 5.5]]
```

Python lists, however, also have special "built-in" functions that allow you to insert items into the list, pop off items from the list, etc. We'll discuss the nature of those functions (which are called **methods**; this relates to object-oriented programming) in more detail in Ch. 5. Even without that discussion, however, it is still fruitful to consider a few examples of using list methods to alter lists:

Example 4 (Changing and examining lists using list methods):
Assume we have the list we defined in Example 1:

```
a = [2, 3.2, 'hello', [-1.2, 'there', 5.5]]
```

What do the following commands give you when typed into the Python interpreter?:

- `a.insert(2,'everyone')`
- `a.remove(2)`
- `a.append(4.5)`
- `a.index('hello')`
- `a.count('hello')`

Solution and discussion: The first command `insert` inserts the string `'everyone'` into the list after the twoth (i.e., third) element of the list. The second command `remove` removes the first occurrence of the value given in the argument. The third command `append` adds the argument to the end of the list.

For the list `a`, if we printed out the contents of `a` after each of the first three lines above were executed one after the other, we would get:

```
[2, 3.2, 'everyone', 'hello', [-1.2, 'there', 5.5]]
[3.2, 'everyone', 'hello', [-1.2, 'there', 5.5]]
[3.2, 'everyone', 'hello', [-1.2, 'there', 5.5], 4.5]
```

The fourth command, after running the first three commands, would then return the location in the list `a` whose value was `'hello'`; that index is 2. The `index` method returns the first occurrence of a match in the list with the value that is passed in via the parameter list of the method's call. The final command counts the number of occurrences of `'hello'` and returns the value 1. If any list elements are themselves lists (such as the last element of `a`), `count` does *not* look into the sub-lists to match the search target.

A little on tuples and strings: Tuples are nearly identical to lists with the exception that tuples cannot be changed (i.e., they are **immutable**). That is to say, if you try to insert an element in a tuple, Python will return an error. Tuples are defined exactly as lists except you use parenthesis as delimiters instead of square brackets, e.g., `b = (3.2, 'hello')`.

You can, to an extent, treat strings as lists. Thus, if:

```
a = "hello"
```

then:

```
"h" in a
```

will return `True`. The command:

```
"el" in a
```

will also return `True`, because the membership operator `in` will work on contiguous substrings. You can also slice strings as if each character were a list element. `a[1:3]` will return the substring `"el"`. See Section 4.8 for more on indexing strings.

2.4 Customizing an X-Y plot

Let's revisit our Figure 2.1 plot. It's fine and all, what if I'd like to change some things? Here, we discuss ways of making the most common changes.

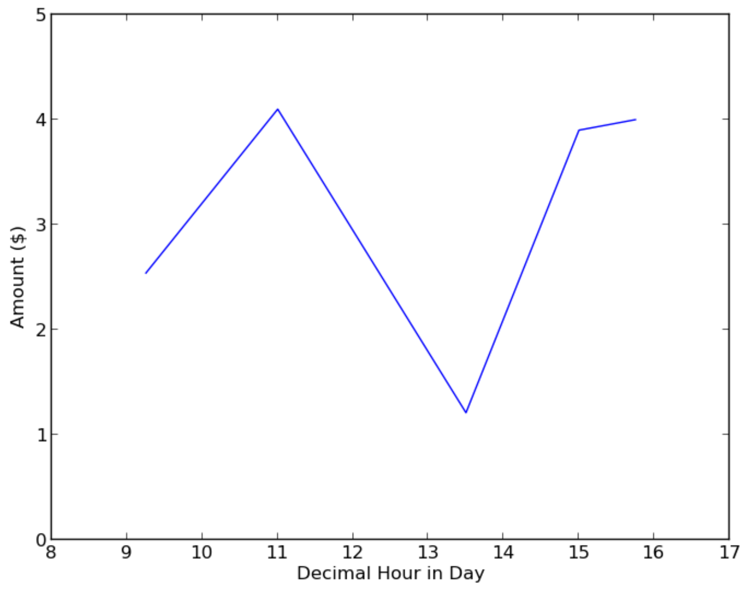


Figure 2.3: Graph created by the code in Section 2.2.1, resized.

2.4.1 Controlling the axes ranges

I'd like to re-center the plot so that the x and y axes ranges are different. In particular, because the hours my store is open is 8 am through 5 pm, I'd like the x-axis to go from 8 to 17. I'd also like someone looking at the graph to have a sense of the absolute point-of-sale amount fluctuations, so I'd like the y-axis to vary from 0 to 5. To do this, I just slip in an `axis` command in line 3 of the script below:

```
1  import matplotlib.pyplot as plt
2  plt.plot([9.25, 11, 13.5, 15, 15.75], [2.54, 4.1, 1.21, 3.9, 4])
3  plt.axis([8, 17, 0, 5])
4  plt.xlabel('Decimal Hour in Day')
5  plt.ylabel('Amount ($)')
6  plt.show()
```

The single argument to the `axis` function is a list where the first two elements are the lower and upper x-axis bounds and the third and fourth elements are the lower and upper y-axis bounds. The resulting graph is shown in Figure 2.3.

2.4.2 Controlling line and marker formatting

To control line and marker features, you can use the appropriate keyword input parameters with the `plot` function, e.g.:

```
plt.plot([1, 2, 3, 4], [1, 2.1, 1.8, 4.3],
         linestyle='--', linewidth=5.0,
         marker='*', markersize=20.0,
         markeredgewidth=2.0,
         markerfacecolor='w')
```

Note how `linestyle`, `marker`, and `markerfacecolor` use special string codes to specify the line and marker type and formatting. The `plot` call above uses a dashed line and a white star for the marker. Linewidth, marker size, and marker edge width are in points.

Instead of using keyword input parameters, you can also specify line color and type and marker color and type as a string third argument, e.g.:

```
plt.plot([1, 2, 3, 4], [1, 2.1, 1.8, 4.3], 'r*--')
```

Notice that this third argument contains *all* the codes to specify line color, line type, marker color, and marker type. That is to say, all these codes can be specified in one string. In the above example, the color of the marker and connecting line is set to red, the marker is set to star, and the linestyle is set to dashed. (The marker edge color is still the default, black, however.)

Tables 2.1 and 2.2 list some of the basic linestyles and marker codes.[1]

2.4.3 Annotation and adjusting the font size of labels

We introduced the `xlabel` and `ylabel` functions in Section 2.2.1 to annotate the x- and y-axes, respectively. To place a title at the top of the plot, use the `title` function, whose basic syntax is the same as `xlabel` and `ylabel`. General annotation uses the `text` function, whose syntax is:

plt.text(<*x-location*>, <*y-location*>, <*string to write*>)

The x- and y-locations are, by default, in terms of **data coordinates.** For all four functions (`xlabel`, `ylabel`, `title`, and `text`), font size is controlled by the `size` keyword input parameter. When set to a floating point value, `size` specifies the size of the text in points.

2.4.4 Plotting multiple figures

If you have have multiple independent figures (not multiple curves on one plot), call the `figure` function before you call `plot` to label the figure accordingly. A subsequent call to that figure's number makes that figure current. For instance, consider this code:

```
plt.figure(3)
plt.plot([5, 6, 7, 8], [1, 1.8, -0.4, 4.3], \
         marker='o')
plt.figure(4)
plt.plot([0.1, 0.2, 0.3, 0.4], [8, -2, 5.3, 4.2], \
         linestyle='-.')
plt.figure(3)
plt.title('First Plot')
```

[1] See http://stackoverflow.com/a/13360032 for a way of listing all the linestyle and marker codes. A list of basically all the line and marker properties can be found here: http://matplotlib.org/api/lines_api.html.

Linestyle	**String Code**
Solid line	`-`
Single dashed line	`--`
Single dashed-dot line	`-.`
Dotted line	`:`

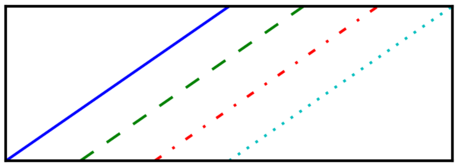


Table 2.1: Some linestyle codes in pyplot and a high-resolution line plot showing the lines generated by the linestyle codes. See http://matplotlib.sourceforge.net/api/pyplot_api.html.

Marker	**String Code**
Circle	`o`
Diamond	`D`
Point	`.`
Plus	`+`
Square	`s`
Star	`*`
Up Triangle	`^`
X	`x`

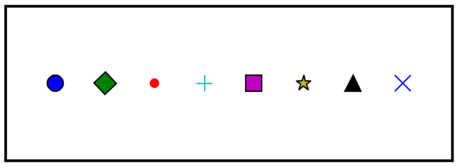


Table 2.2: Some marker codes in pyplot and a high-resolution line plot showing the markers generated by the marker codes. See http://matplotlib.sourceforge.net/api/pyplot_api.html.

Color	**String Code**
Black	`k`
Blue	`b`
Green	`g`
Red	`r`
White	`w`

Table 2.3: Some color codes in pyplot. See http://matplotlib.sourceforge.net/api/colors_api.html for a full list of the built-in colors codes as well as for ways to access other colors.

Line 1 creates a figure and gives it the name "3". Lines 2–3 (which is a single logical line to the interpreter) makes a line plot with a circle as the marker to the figure named "3". Line 4 creates a figure named "4", and lines 5–6 make a line plot with a dash-dot linestyle to that figure. Line 7 makes figure "3" the current plot again, and the final line adds a title to figure "3".

2.4.5 Plotting multiple curves

To plot multiple curves on a single plot, you can make repeated calls to `plot`. The lines for each call will be added to the same figure. Alternately, you can string the set of three arguments (x-locations, y-locations, and line/marker properties) for each plot one right after the other. For instance, consider this code:

```
plt.plot([0, 1, 2, 3], [1, 2, 3, 4], '--o',
         [1, 3, 5, 9], [8, -2, 5.3, 4.2], '-D')
```

The first three arguments specify the x- and y-locations of the first curve, which will be plot using a dashed line and a circle as the marker. The second three arguments specify the x- and y-locations of the second curve, which will be plot with a solid line and a diamond as the marker. Both curves will be on the same figure.

2.4.6 Adding a legend

Adding a legend, with the default settings, is pretty straightforward: Just call the `legend` function (with no arguments) after you've made the plots. When you plot each curve, however, you have to set the `label` keyword input parameter to what the text for that curve should be in the legend. For instance:

```
plt.plot([0, 4, 7, 8], [1, 2, 3, 4], 'r--o', label="Curve 1")
plt.plot([1, 3, 5, 9], [8, -2, 5.3, 4.2], 'b-D', label="Curve 2")
plt.legend()
```

produces the plot in Figure 2.4.6. Note the `"r"` and `"b"` strings in the `plot` calls produce a red and blue line/marker, respectively. For more information on legends, see http://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes.legend (the documentation is pretty verbose, though, so you might find it a more fruitful experience after some more experience with Python).

2.4.7 Adjusting the plot size

One easy way of adjusting the plot size is to set the `figsize` and `dpi` keyword input parameters in the `figure` command.[2] For instance, this call to `figure`:

```
plt.figure(1, figsize=(3,1), dpi=300)
```

before the call to the `plot` command, will make figure "1" three inches wide and one inch high, with a resolution of 300 dots per inch (dpi). Note that the `figsize` keyword is set to a two-element tuple.

[2] http://stackoverflow.com/a/638443 (accessed August 13, 2012).

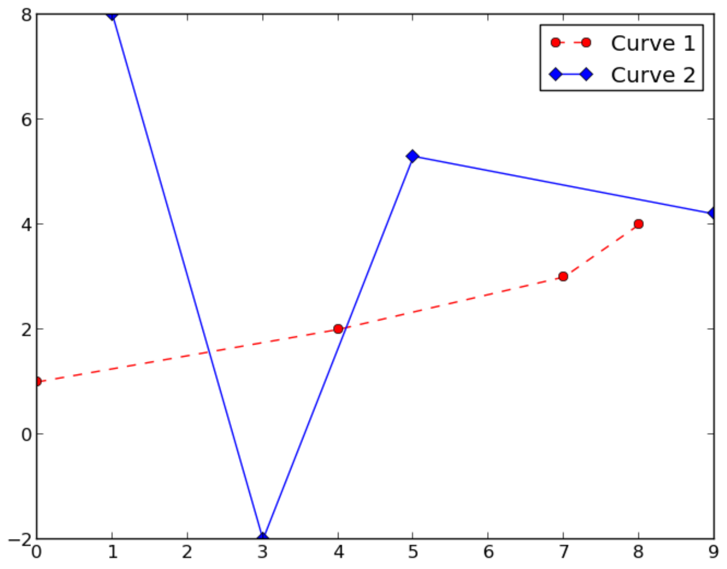


Figure 2.4: Graph created by the code in Section 2.4.6.

2.4.8 Saving figures to a file

To write the plot out to a file, you can use the `savefig` function. For example, to write out the current figure to a PNG file called *testplot.png*, at 300 dpi, type:

```
plt.savefig('testplot.png', dpi=300)
```

Here we specify an output resolution using the optional `dpi` keyword parameter; if left out, the matplotlib default resolution will be used. Note that it is not enough for you to set `dpi` in your `figure` command to get an output file at a specific resolution. The `dpi` setting in `figure` will control what resolution `show` displays at while the `dpi` setting in `savefig` will control the output file's resolution; however, the `figsize` parameter in `figure` controls the figure size for both `show` and `savefig`.

You can also save figures to a file using the graphical user interface (GUI) save button that is part of the plot window displayed on the screen when you execute the `show` function. If you save the plot using the save button, it will save at the default resolution, even if you specify a different resolution in your `figure` command; use `savefig` if you want to write out your file at a specific resolution.

2.5 ⌨ Python Sidebar: Commenting

We're starting to write scripts that are longer and longer. While Python is a very clear language, at some point we'll want to put explanatory notes in our code to help explain why our code is written a certain way, what the code does, and what is the source of our information. Comments in Python are begin with the hash symbol (“#”). Whether the hash is found at the beginning of a line or mid-way through a line, the Python interpreter considers everything after and including the hash as a comment.

Let’s take the code from Section 2.4.4 and add some commenting to it:

```
#- Create figure 3 as a scatter plot:
plt.figure(3)   #+ Go to figure 3
plt.plot([5, 6, 7, 8], [1, 1.8, -0.4, 4.3], \
         marker='o')

#- Create figure 4 as a line plot:
plt.figure(4)   #+ Go to figure 4
plt.plot([0.1, 0.2, 0.3, 0.4], [8, -2, 5.3, 4.2], \
         linestyle='-.')

#- Return to figure 3 and add a title:
plt.figure(3)   #+ Go to figure 3
plt.title('First Plot')
```

These comment lines don’t really say much than is already clear from the code, but they illustrate how the comment symbol works.

2.6 Summary

Matplotlib makes it easy to make x-y plots of various types. We’ll be using this package a lot as we learn more data analysis tools.

Chapter 3

Simple Data Analysis

Chapter Contents

Calculations that are suitable for a handheld calculator, even a programmable one, represent only a smattering of the kinds of calculations we can do today and are interested in with regards to business data. Full-fledged programming language like Python have the capability to bring business insights from large collections of data, whether they be from finance, human resources, manufacturing, sales, operations, or any aspect of a business. In this chapter, we make our first foray in using Python as a data analysis tool.

3.1 My first dataset

Let's look at some real data. Below are the adjusted (for season, holiday, and trading-days) total monthly sales at gasoline stations, for the U.S. as a whole, during calendar year 2015 (in millions of dollars). The first number is for January, the second for February, and so on:[1]

[1] https://www.census.gov/retail/marts/www/adv44700.txt (accessed September 8, 2016).

```
35384
36453
36644
36211
37208
37727
37436
36568
34945
34517
33834
33703
```

Let's first calculate the mean of these values. A function for calculating the mean is called, not surprisingly, `mean`, and is part of the SciPy package. It takes a single list of input values (see Section 2.3 for more on lists) as an input parameter. Thus, to calculate the mean and save the value as the variable `mean_2015` we would type:

```
import scipy
mean_2015 = scipy.mean([35384, 36453, 36644, 36211, 37208, 37727, \
                        37436, 36568, 34945, 34517, 33834, 33703])
```

We could also first save the year's worth of input values as a list variable and pass the variable into the function:

```
import scipy
data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
             37436, 36568, 34945, 34517, 33834, 33703]
mean_2015 = scipy.mean(data_2015)
```

This is all fine and good, but what if we want to do more than just pass the data into a function? What if we want to manipulate or change the data values in some way to help us get the calculations we are interested in? That is to say, how do we go about:

- Choosing which groups of the data to examine and manipulate (e.g., "each item, one at a time," "just the first five items," etc.).
- Asking questions of the data (e.g., "which values are greater than $36,000 million," "what months are sales less than $37,000 million," etc.).
- Changing the data.

Let's go through each of these tasks on this first dataset.

3.2 Choosing which groups of the data to examine and manipulate

In Section 2.3, we saw how to examine a single element (by specifying the index) and how to extract a subset of elements (by a range of indices). When we have a collection of items, however, what we really want to be able to do is not just examine an element or set of elements but to *repeatedly* examine an element or set of elements. That

Let's first consider the case of examining each element in a list of data, one element at a time. Here's the code to print the above `data_2015` list, one element at a time:

```
1 data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
2              37436, 36568, 34945, 34517, 33834, 33703]
3 for idata in data_2015:
4     print(idata)
```

Let's unpack this. The `for` statement in line 2 tells Python to go through each element of `data_2015` and set the variable `idata` to that element, one at a time, in the order they are given in the list `data_2015`. Once that is done, Python executes the contents of the body of the loop, that is, all the lines of code that are indented in four spaces under the `for` line. In this case, there is only one line in the body of the loop (that is, that are indented in four spaces), and so nothing else is done except printing `idata`.

What if you want to go through the list in a different order? You can loop through the list by referencing the indices of the list elements rather than the elements themselves. For instance, if you wanted to print out each element of `data_2015`, backwards (i.e., starting with the December value and ending with the January value), you could do the following:

```
1 data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
2              37436, 36568, 34945, 34517, 33834, 33703]
3 for i in [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]:
4     print(data_2015[i])
```

We could also store the list of indices as a list variable and loop through that list in the `for` loop immediately above. Remember the first element of a list has index 0.

We can use the same strategy of looping through indices to enable us to examine only a subset of the data. For instance, to print out only the data for February and March:

```
1 data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
2              37436, 36568, 34945, 34517, 33834, 33703]
3 for i in [1, 2]:
4     print(data_2015[i])
```

Because going through a list of indices is such a common operation, Python includes a built-in function called `range` which produces such a list: `range(n)` returns the list [0, 1, 2, …, $n-1$]. Thus:

```
data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
             37436, 36568, 34945, 34517, 33834, 33703]
for i in range(3):
    print(data_2015[i])
```

prints out only the data for January, February, and March:

Here, we covered just the basics on loops. See Section 3.3 for some more on looping.

3.3 ⌨ Python Sidebar: More on `for` looping

We went over basic looping in Section 3.2. Here we provide some more of the "computer science" behind basic looping.

The standard loop in Python begins with `for` and has the syntax:

`for` <*index*> `in` <*list*>`:`

followed by the contents of the loop. (Don't forget the colon at the end of the `for` line.) The `for` loop is kind of different compared to the Java `for` loops you might be familiar with. In Java, etc. you specify a beginning value and an ending value (often 1 and an integer n) for an index, and the loop runs through all integers from that beginning value to that ending value, setting the index to that value. In Python, the loop index runs through a list of items, and the index is assigned to each item in that list, one after the other, until the list of items is exhausted.

Recall that elements in a Python list can be of *any* type, and that list elements do not all have to be of the same type. Also remember that Python is dynamically typed, so that a variable will change its type to reflect whatever it is assigned to at any given time. Thus, in a loop, the loop index could, potentially, be changing in type as the loop runs through all the elements in a list. Since the loop index does not have to be an integer, it doesn't really make sense to call it an "index;" in Python, it's called an **iterator**. When we talk about objects and methods, we'll see how the fact the loop index is not a number but an object makes Python loops more powerful.

A lot of the time you will loop through lists. Technically, however, Python loops can loop through any data structure that is **iterable,** i.e., a structure where after you've looked at one element of it, it will move you onto the next element. Arrays (which we'll cover in Ch. 4) are another iterable structure.

3.4 Asking questions of the data

Loops enable us to go through the data, one at a time. We're now in a position to ask questions of the data. Let's consider our gasoline retail data again. Instead of printing every value to screen, let's say we only want to print out amounts that are greater than $35,000 million. The code to do that would be:

```
data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
             37436, 36568, 34945, 34517, 33834, 33703]
for idata in data_2015:
    if idata > 35000:
        print(idata)
```

In line 4, we have a branching statement or `if` statement. It takes the value of `idata`, checked to see if `idata` is greater than 35000, and if so, prints the value of `idata` to the screen.

Note that as with a `for` loop, if the true/false test the `if` statement examines is true, the statement executes the block of lines after the `if` statement, and this block of lines is denoted by four spaces of indentation. In the above code, there is only one line executed when `idata` is greater than 35000; if we had more lines, we'd list them one after the other, all indented four spaces in.

What if we decided we wanted to check to see which values of the data are over 35000 and for those values to print both the value of the data and the month number? To do this, we'll make use of the indices of the list, since the index values are the same as the month number, minus one. We thus loop over the indices instead of the data values themselves:

```
1  data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
2                37436, 36568, 34945, 34517, 33834, 33703]
3  for i in range(len(data_2015)):
4      if data_2015[i] > 35000:
5          print(str(i+1) + " " + str(data_2015[i]))
```

Note that in line 3, the input parameter for `range` is the return value of the call to `len`, using `data_2015` as the input parameter to that call. This way, if the length of `data_2015` changes, we don't have to change the `for` loop. Also remember that the `print` statement is looking for a string as input and you have to use the `str` function to convert the month number (i.e., `i+1` in line 6) and the value from `data_2015` (i.e., `data_2015[i]` in line 6).

Finally, what if we wanted to ask a question, do one thing if the answer is true adn something else if the answer to the question is false? To do so, we use `if` paired with `else`. In the code below, we print the month number and amount if the value of the data is greater than 35000 and print a message `"Too low"` otherwise:

```
1  data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
2                37436, 36568, 34945, 34517, 33834, 33703]
3  for i in range(len(data_2015)):
4      if data_2015[i] > 35000:
5          print(str(i+1) + " " + str(data_2015[i]))
6      else:
7          print("Too low")
```

Note that there is a colon after the `else`, and the lines of code you want to execute if the `if` condition is false is also indented in four spaces. As you've noticed, all blocks of code that are in a distinct grouping (e.g., a function definition, a `for` loop body, an `if` statement body) have their first lines of that body indented in one indentation level of four spaces.

3.5 Python Sidebar: Booleans

In our description of `if` statements in Section 3.4, we just presented the questions we asked (e.g., `idata > 35000`) as if it was obvious they are questions. Here we make our description a little more precise, to give us tools for dealing with more complex questions.

Expressions like `idata > 35000` are called **boolean** expressions because the Python interpreter returns a "true" or "false" value after evaluating the expression. For instance, in the interpreter typing the following will obtain:

```
>>> idata = 40000
>>> idata > 35000
True
>>> idata > 50000
False
```

Notice that the values that are returned from evaluating the true/false expression are the values `True` and `False`. (The capitalized first letter is meaningful here, as in all Python code because Python is case-sensitive.) These are actual values, just like a number or a string. And like actual values, you can set variables to them:

```
>>> idata = 40000
>>> test_result = idata > 35000
>>> print(test_result)
True
>>> type(test_result)
<type 'bool'>
```

Variables whose values are either `True` or `False` are called **boolean variables**. You can set a variable directly to one of those values or, as we did above, set them to the result of a boolean expression.

The operators we use to operate on boolean variables are called logical or boolean operators. The main ones are: `and`, `or`, and `not`.

Example 5 (Operations with boolean variables):

Try this in a Python interpreter:

```
a = True
b = False
print(a and b)
print(a or b)
print(not a)
```

What did you get?

Solution and discussion: The first two lines assign `a` and `b` as boolean variables. The first two `print` statements return `False` and `True`, respectively. Remember that `and` requires both operands to be `True` in order to return `True`, while `or` only requires one of the operands be `True` to return

`True`. The `not` operator is a unary operator, meaning it operates on one token rather than two (like `and` and `or`).

Membership testing: The `in` operator is another useful logical operator, though not as well-known as the traditional logical operators like equality, greater than, etc. The `in` operator can be used in Python lists and strings to check to see if some value is one of the elements (for a list) or is at least a substring (for a string).

Example 6 (Checking list membership):

Assume we have the following list:

```
words = ['hi', 'bye', 'yes', 'no' ]
```

What do the following commands give you when typed into the Python interpreter?:

- `'hi' in words`
- `'okay' in words`
- `'y' in words`
- `'bye' not in words`

Solution and discussion: An expression using the `in` operator returns a boolean value. The first command returns `True` because the string "hi" is in the list `words`. The next two commands return `False` because neither the strings "okay" nor "y" are elements in the list. Note that "y" exists as a substring of "yes", but that doesn't count as far as the `in` operator is concerned in the above commands, since we're using the operator on a list; the value to the left of `in` has to *completely* match at least one of the *elements* of the list to the right of `in`. The final command returns `False`. Since "bye" is in the list, it is false that "bye" is not in `words`. Note that the syntax to say "not in" is `not in` and not a `not` applied to the return of the `in` expression (boy, that's a tongue twister!).

String membership: When `in` is used on strings, Python looks for any occurrence of the *entire* string to the left of `in` *anywhere* in the string to the right of `in`. Thus:

```
>>> 'us' in 'business'
```

will return `True` but:

```
>>> 'using' in 'business'
```

will return `False`.

Note that the syntax of `in` for membership testing is not the same as `in` in a `for`...`in` statement. Even though the same term ("in") is being used, in the latter case the `in` functions to tell Python to iterate through what is to the right of the `in`.

3.6 ⌨ Python Sidebar: Looping an indefinite number of times

With the introduction of boolean variables in Section 3.5, we can discuss another way of looping that enables us to do a task without knowing ahead of time how many times we will do that task. With lists of data, if you want to loop through all elements in that list, you do know ahead of time how many times you will want to loop through that list, so a `for` loop is the best approach. When you don't know that ahead of time, the `while` loop is the best approach.

The Python `while` loop begins with the syntax:

`while` *<condition>*:

The code block (indented four spaces) that follows the `while` line is executed while *<condition>* evaluates as `True`. Here's a simple example:

Example 7 (A `while` loop):

Type in the following into a file (or the interpreter):

```
a = 1
while a < 10:
    print(a)
    a = a + 1
```

What did you get?

Solution and discussion: This will print out the integers one through nine, with each integer on its own line. Prior to executing the code block underneath the `while` statement, the interpreter checks whether the condition (`a < 10`) is true or false. If the condition evaluates as `True`, the code block executes; if the condition evaluates as `False`, the code block is not executed. Thus:

```
a = 10
while a < 10:
    print(a)
    a = a + 1
```

will do nothing. Likewise:

```
a = 10
while False:
    print(a)
    a = a + 1
```

will also do nothing. In this last code snippet, the value of the variable `a` is immaterial; as the condition is always set to `False`, the `while` loop will never execute. (Conversely, a `while True:` statement will never terminate. It is a bad idea to write such a statement ☺.)

Please see your favorite Python reference if you'd like more information about `while` loops.

3.7 Changing the data

There are two main ways we can “change” the values in a list of data, which we saw in Section 2.3. The first way is to loop through the indices of each element in the list of data and change each element’s values via assignment. For instance, the following goes through `data_2015` and changes every value greater than 35000 to 0, by looping through the list element indices:

```
data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
             37436, 36568, 34945, 34517, 33834, 33703]
for i in range(len(data_2015)):
    if data_2015[i] > 35000:
        data_2015[i] = 0
```

The second way is to create a separate, new list, loop through the elements of the old list, make changes to the values of the old list, then save the changed values into the new list. The following does the same as the previous code snippet except the “changed” list of data is in a list called `new_data_2015`. Note that the old list of data `data_2015` is unchanged:

```
1 new_data_2015 = []
2 data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
3              37436, 36568, 34945, 34517, 33834, 33703]
4 for idata in data_2015:
5     if idata > 35000:
6         new_data_2015.append(0)
7     else:
8         new_data_2015.append(idata)
```

In line 1, we create a new list as an empty list. In lines 6 and 8, we use the `append` method to add either 0 or the value of `idata` into the new list `new_data_2015`.

3.8 Analyzing data

We now can look at data and do things with it. Let’s use these tools to analyze data. In Ch. 1, we learned about some functions in the SciPy package. We’ll use some other SciPy functions to do basic statistical analysis on our data from Section 3.1. We’ll use functions provided by Python as well as implement our own calculation of a function. The functions we’ll use are at the SciPy module level (i.e., they can be used by an `import scipy` command).

Recall the moments[2] mean and standard deviation are defined as:[3]

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

[2]https://en.wikipedia.org/wiki/Moment_(mathematics) (accessed September 13, 2016).

[3]https://en.wikipedia.org/wiki/Mean and https://en.wikipedia.org/wiki/Standard_deviation (both accessed September 13, 2016).

and

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

respectively.

We can, of course, use the SciPy `mean` and `std` functions to calculate the mean and standard deviation, respectively:

```
import scipy
data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
             37436, 36568, 34945, 34517, 33834, 33703]
data_mean = scipy.mean(data_2015)
data_std = scipy.std(data_2015)
print(data_mean)
print(data_std)
```

The code above produces the following as output:

```
35885.8333333
1322.68998845
```

which are the annual mean and standard deviation of U.S. gasoline retail sales (adjusted) for 2015.

But, using a pre-written function isn't very instructive. Let's start with the definition of mean given above and calculate the 2015 annual mean directly from the mathematical definition, using what we've learned about looping.

In our previous loops, we hadn't done anything arithmetic with the numbers. To calculating the mean, we see from the presence of a summation operator that we need to create a running sum of the data values as well as figure out how many points there are we're averaging over (N). To do the former, we create a variable called `running_sum`, initialize it to 0, and add to it in the loop. To do the latter, we can use the `len` function on the list of data:

```
1 data_2015 = [35384, 36453, 36644, 36211, 37208, 37727, \
2              37436, 36568, 34945, 34517, 33834, 33703]
3 running_sum = 0
4 for idata in data_2015:
5     running_sum = running_sum + idata
6 data_mean = float(running_sum) / len(data_2015)
7 print(data_mean)
```

which gives 35885.8333333, just as the SciPy function `mean` did. Note that Python supports the +=, etc. operator, so we could rewrite line 5 as:

```
running_sum += idata
```

Of course, SciPy has many more statistical functions. Some are at the SciPy module level like `mean` while others are in the stats submodule of SciPy (and are accessed by an `import scipy.stats` command). To learn more about all that SciPy has to offer, see the SciPy Reference Guide (http://docs.scipy.org/doc/scipy/reference/) and the NumPy Reference Guide (http://docs.scipy.org/doc/numpy/reference/). Many of the NumPy functions (such as `mean`) are available at the SciPy module level, but the documentation is in the NumPy Reference Guide. (We'll talk more about the NumPy package in Ch. 4.)

3.9 ⌨ Python Sidebar: Docstrings

In Section 2.5, we looked at how to write comments in the code. Another way of "commenting" is the use of **docstrings**. Docstrings describe functions. What makes them special is that if you follow the standardized format for docstrings, Python's `help` function will automatically turn the docstring into a manual page, with the proper formatting. Let's look at an example. We'll write a docstring for the function we presented in Figure 1.6:

```
def percent_to_decimal(input):
    """Convert a percent to its decimal equivalent.

    The decimal equivalent is computed by dividing the input by 100.
    The result will be float, regardless of whether or not the input
    is float or integer.  input is assumed to be scalar; no testing
    is done to see whether this is true.

    Positional Input Parameter:
        input : float or int
            Numerical value of a percent.  Scalar.

    Keyword Input Parameters:
        None

    Examples:
    >>> percent_to_decimal(42)
    0.42
    """
    return input / 100.
```

The docstring is a triple-quote string (see Section 1.3.2 regarding triple-quote strings) that immediately follows the `def` line of the function. Note that the docstring is indented in four spaces, so Python knows it's part of the function's definition block.

The first line of the docstring is a single sentence summarizing what the function does. This line is followed by a blank line and then one or more paragraphs describing the algorithm used in the function, general details about dependencies, general details about the return value, etc. (In this case, we wouldn't normally say how the decimal equivalent is calculated since it's a trivial calculation, but we put such a description here for illustrative purposes.) Finally, the docstring has sections that describe the input parameters, provide examples of how to use the function, etc. Information is provided on the type of variables, what those variables correspond to, etc.

Notice how we put returns in the docstring to make the text nicely formatted if the code were printed on a piece of letter-sized paper. (The triple-quotes make this automatic.) I highly recommend this practice! It makes your documentation much easier to read for someone who is looking through your code file.

3.10 Python Sidebar: Writing code a little bit at a time

As we start writing more and more complex programs, I want to encourage you to write your code a little bit at a time. What I mean by that is that you should write one or two lines of code at a time, run the code (or code snippet), and print out whether the variables and results are what you expect them to be. If everything looks good, write two more lines of code. If you do not get what you expect, find what's going wrong and fix it.

This sounds like a lot of work but this "small bits" approach makes it much more likely that your code will work and work correctly. The time you spend testing the one to two lines of code you write will pay off in the time you save by not having so many bugs to hunt down and fix. This "small bits" approach works particularly well with Python. Because the code does not need to be compiled, you can write a snippet, cut-and-paste it into the Python interpreter, and see if it works the way you expect.

3.11 Summary

With looping and branching, we can efficiently examine datasets and manipulate the data. Variables, looping, and branching are the there core constructs of a programming language. With these, you can get the computer to do incredibly complex calculations. How to do so, and to see the additional tools Python gives to make our lives easier in more advanced data analysis, is the topic of the next chapter.

Chapter 4

Manipulating Data and More Complex Data Analysis

Chapter Contents

Taking a list of data and processing it is useful. But most of the time, we're interested in not only a single list of data but a whole table (or bunch of tables) of data. Many times in business scenarios, we are looking to see how many factors (e.g., costs from many suppliers, pricing in different markets, etc.) relate to each other and tell us about how a business is doing and how to grow the business. Clearly, single lists of data won't get us very far.

The key to using Python to wrangle these kinds of business datasets is the Numpy array. NumPy arrays (which we'll just call arrays from now on) are tables of numbers. These tables can be one-dimensional (as in a list of data), two-dimensional (like a spreadsheet), three-dimensional (like a workbook of spreadsheets), or four, five, or more-dimensional. Arrays differ from lists in mainly two ways: every element in a given array has to be of the same type and the syntax for dealing with multi-dimensional arrays is a lot more straightforward and powerful.

In this chapter, we'll look at data analysis that deals with this more complex data. Along the way, we'll describe how we create and manipulate arrays.

4.1 Revisiting the gasoline retail dataset

Of course, the total dataset of monthly gasoline station adjusted retail sales data, which we showed part of in Section 3.1, includes more than just calendar year 2015. Table 4.1 shows all the data from 1992–2016.

YEAR	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
1992	12803	12601	12639	12710	12870	12886	12966	13092	13246	13304	13358	13475
1993	13398	13596	13487	13539	13521	13461	13502	13372	13416	13755	13754	13627
1994	13656	13875	13911	13867	13773	14059	14322	14685	14649	14709	14878	14885
1995	14915	14963	14921	15021	15215	15333	15249	15227	15194	15007	14981	15246
1996	15479	15457	15951	16252	16488	16352	16075	16066	16088	16360	16602	16759
1997	16938	16984	16969	16613	16367	16378	16396	16682	16757	16711	16638	16536
1998	16225	16054	15703	15798	16057	15933	16087	15851	15812	15948	16014	16399
1999	16394	16309	16529	17115	17186	17070	17704	18135	18339	18632	18992	20095
2000	19358	20214	20819	19907	20242	20885	20905	20574	21327	21412	21906	22124
2001	21931	21615	20652	21618	22628	21928	20686	20878	21336	20033	19245	18950
2002	19130	19068	19813	20856	20930	20733	21438	21182	21272	21659	22009	22464
2003	23073	23885	23968	22581	21810	21777	22196	23073	23214	22714	23472	23994
2004	24889	25256	25679	25518	27011	27050	26745	26552	27161	28656	29349	29046
2005	28438	29015	29528	29950	29113	29917	30976	33368	36008	36088	33185	33119
2006	34298	34174	34126	35752	36126	36127	37064	37765	34865	33010	33393	34858
2007	34168	34973	35963	36386	38031	37207	36996	37155	38327	39425	41897	41637
2008	42425	42411	43140	43025	44313	46429	46869	45701	45747	40640	31890	27952
2009	28768	29910	28811	28848	30507	33241	33057	34781	34868	34955	36655	37232
2010	37372	37009	37184	37208	36700	35761	35778	36261	37423	38534	39269	41172
2011	41731	42226	43654	44506	45191	44501	44638	45047	44935	44755	45596	45142
2012	45673	47216	47358	47023	45918	43711	43467	46411	47115	48015	46690	45827
2013	45932	48776	47161	45760	45212	45471	45384	44847	45035	44381	43963	45909
2014	46399	46816	46041	46317	45976	44973	44563	44416	44220	43305	42282	39199
2015	35384	36453	36644	36211	37208	37727	37436	36568	34945	34517	33834	33703
2016	32653	30967	32113	32984	33538	34265	33329					

Table 4.1: Adjusted (for season, holiday, and trading-days) total monthly sales at gasoline stations, for the U.S. as a whole, from calendar years 1992–2016 (in millions of dollars). Source: https://www.census.gov/retail/marts/www/adv44700.txt (accessed September 8, 2016).

Immediately, we see that this table of data is best understood as a two-dimensional grid, rather than a list. How do we hold all this data into a single variable in Python? We could make a list of lists, but the syntax of specifying elements of such a variable sounds like it'd be a little hairy.

Instead, Python has a nice data structure called a **NumPy array** that makes handling two- (and higher) dimensional data very straightforward. They are similar to Java arrays but are much more useful and powerful. Because "NumPy arrays" is a mouthful, we'll often just call them "arrays."

In the next section, we'll take a look at some basic statistical calculations using the gasoline retail data from calendar years 1992–2015 (since the 2016 data ends with July in Table 4.1). Through that example, and later ones, we'll see how to create and manipulate arrays.

4.2 Basic statistics

Let's take a few years from the Table 4.1 data, say 2012–2015, and do the following with that data: (1) calculate the annual mean retail sales for each year in the period, and (2) calculate the monthly mean retail sales for each month in the period.

The first step to doing either calculation is to put the data into a NumPy array variable. In Ch. 5, we'll discuss how to read-in large amounts of data from files and put their contents into an array automatically. Here, we'll just type it in:

```
1  import numpy as np
2  data_as_list = [ \
3      [45673, 47216, 47358, 47023, 45918, 43711, 43467, 46411, 47115, 48015, 46690, 45827], \
4      [45932, 48776, 47161, 45760, 45212, 45471, 45384, 44847, 45035, 44381, 43963, 45909], \
5      [46399, 46816, 46041, 46317, 45976, 44973, 44563, 44416, 44220, 43305, 42282, 39199], \
6      [35384, 36453, 36644, 36211, 37208, 37727, 37436, 36568, 34945, 34517, 33834, 33703] ]
7  data_as_array = np.array(data_as_list)
```

To utilize the NumPy array functions, we need to first import the NumPy package, as shown in line 1. We give an alias to the package, so instead of having to type `numpy` before each NumPy function we use, we only need to type `np`. We enter in the values as list of lists, in lines 2–6. Finally, we create an array version of `data_as_list` in the last line, by using the `array` function.

With our data in array form, we can more easily slice and dice the data to do the calcuations we want. To calculate the annual means for each year in this dataset, we need to go through each row of the array (since each row holds a year of data), and calculate the mean of all the values in that row. The following code will make these calculations, using the `data_as_array` array, and fill the lists `annual_means` and `monthly_means` with the respective means:

```
import numpy as np

annual_means = []
num_rows = np.shape(data_as_array)[0]
for irow in range(num_rows):
    annual_means.append( np.mean(data_as_array[irow,:])

monthly_means = []
num_cols = np.shape(data_as_array)[1]
for icol in range(num_cols):
    monthly_means.append( np.mean(data_as_array[:,icol])
```

You might be looking at the code above and be saying, "Whoa!" It is a bit dense. But based on the names of the variables, we can make some reasoned guesses as to what is going on in the code.

In line 3, we initialize an empty list to hold all the values of the annual means that we will calculate below. Line 4 appears to calculate the number of rows in `data_as_array`, but how does it do that? It makes use of a function `shape` which, when we compare to line 9, apparently returns a list that gives the number of rows then the number of columns. We'll talk more about `shape` and an array's **shape** in Section 4.3.

Once we know how many rows there are, we iterate through the indices of the rows of `data_as_array`, starting with the loop definition in line 5. The loop body, in line 6, needs a little unpacking. In Python, you'll often see these kinds of seemingly complicated, composite statements, because you can use the return value of an operation or function call as input into another operation or function call. The way to read these composite statements is to start in the inner-most parenthesis or operation nesting level and work outwards from there. In line 6, this is the `data_as_array[irow,:]` statement. We'll talk more about this in Section 4.3, but what this statement does is select the row with index `irow` and returns all elements of that row as a subarray. That return value is then fed into the NumPy `mean` function, which takes the mean of that subarray. The return value from the `mean` function call is a scalar that is appended to the `annual_means` list.

Lines 8–11 behave similarly as lines 2–7. The difference here is in line 10 we iterate over all the indices of the columns of `data_as_array` and the subarray we create using `data_as_array[:,icol]` in line 11 is an array consisting of all the elements in the column `icol` (i.e., all the Januarys, all the Februarys, etc.).

As we can see, arrays are like lists but also different, and provide additional functionality that makes our code more compact and powerful. We now turn to Section 4.3 to spend more time talking about making and using NumPy arrays and to do so with a little more structure.

4.3 Python Sidebar: More on creating and using NumPy arrays

An array is like a list except: All elements are of the same type, so operations with arrays are much faster; multi-dimensional arrays are more clearly supported; and array operations are supported. To utilize NumPy's functions and attributes, you import the package `numpy`, often as the alias `np`.

4.3.1 Creating arrays

The most basic way of creating an array is to take an existing list and convert it into an array using the `array` function in NumPy. Here is a basic example:

Example 8 (Using the `array` function on a list):

Assume you have the following list:

```
mylist = [[2, 3, -5],[21, -2, 1]]
```

then you can create an array `a` with:

```
import numpy as np
a = np.array(mylist)
```

The `array` function will match the array type to the contents of the list. Note that the elements of `mylist` have to be convertible to the same type. Thus, if the list elements are all numbers (floating point or integer), the `array` function will work fine. Otherwise, things could get dicey.

Sometimes you will want to make sure your NumPy array elements are of a specific type. To force a certain numerical type for the array, set the `dtype` keyword to a type code:

Example 9 (Using the `dtype` keyword):
Assume you have a list `mylist` already defined. To make an array `a` from that list that is double-precision floating point, you'd type:

```
import numpy as np
a = np.array(mylist, dtype='d')
```

where the string `'d'` is the **typecode** for double-precision floating point. Some common typecodes (which are all strings) include:

- `'d'`: Double precision floating
- `'f'`: Single precision floating
- `'i'`: Short integer
- `'l'`: Long integer

Often you will want to create an array of a given size and shape, but you will not know in advance what the element values will be. To create an array of a given shape filled with zeros, use the `zeros` function, which takes the shape of the array (a tuple) as the single positional input argument (with `dtype` being optional, if you want to specify it):

Example 10 (Using the `zeros` function):
Let's make an array of zeros of shape (3,2), i.e., three rows and two columns in shape. Type in:

```
import numpy as np
a = np.zeros((3,2), dtype='d')
```

Print out the array you made by typing in `print(a)`. Did you get what you expected?

Solution and discussion: You should have gotten:

```
>>> print(a)
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
```

Note that you don't have to type `import numpy as np` prior to every use of a function from NumPy, as long as earlier in your source code file you have done that import. In the examples in this section, I will periodically include this line to remind you that `np` is now an alias for the imported NumPy module. However, in your own code file, if you already have the `import numpy as np` statement near the beginning of your file, you do not have to type it in again as per the example. Likewise, if I do not tell you to type in the `import numpy as np` statement, and I ask you to use a NumPy function, I'm assuming you already have that statement earlier in your code file.

Also note that while the input shape into `zeros` is a tuple, which all array shapes are, if you type in a list, the function call will still work.

Another array you will commonly create is the array that corresponds to the output of `range`, that is, an array that starts at 0 and increments upwards by 1. NumPy provides the `arange` function for this purpose. The syntax is the same as `range`, but it optionally accepts the `dtype` keyword parameter if you want to select a specific type for your array elements:

Example 11 (Using the `arange` function):

Let's make an array of 10 elements, starting from 0, going to 9, and incrementing by 1. Type in:

```
a = np.arange(10)
```

Print out the array you made by typing in `print(a)`. Did you get what you expected?

Solution and discussion: You should have gotten:

```
>>> print(a)
[0 1 2 3 4 5 6 7 8 9]
```

Note that because the argument of `arange` is an integer, the resulting array has integer elements. If, instead, you had typed in `arange(10.0)`, the elements in the resulting array would have been floating point. You can accomplish the same effect by using the `dtype` keyword input parameter, of course, but I mention this because sometimes it can be a gotcha: you intend an integer array but accidentally pass in a floating point value for the number of elements in the array, or vice versa.

4.3.2 Array indexing

Like lists, element addresses start with zero, so the first element of a 1-D array `a` is `a[0]`, the second is `a[1]`, etc. Like lists, you can also reference elements starting from the end, e.g., element `a[-1]` is the last element in a 1-D array `a`.

Array slicing follows rules very similar to list slicing:

- Element addresses in a range are separated by a colon.
- The lower limit is inclusive, and the upper limit is exclusive.
- If one of the limits is left out, the range is extended to the end of the range (e.g., if the lower limit is left out, the range extends to the very beginning of the array).
- Thus, to specify all elements, use a colon by itself.

Here's an example:

Example 12 (Array indexing and slicing):

Type the following in a Python interpreter:

```
a = np.array([2, 3.2, 5.5, -6.4, -2.2, 2.4])
```

What does `a[1]` equal? `a[1:4]`? `a[2:]`? Try to answer these first without using the interpreter. Confirm your answer by using `print`.

Solution and discussion: You should have gotten:

```
>>> print(a[1])
3.2
>>> print a[1:4]
[ 3.2  5.5 -6.4]
>>> print(a[2:])
[ 5.5 -6.4 -2.2  2.4]
```

For multi-dimensional arrays, indexing between different dimensions is separated by commas. Note that the fastest varying dimension is always the last index, the next fastest varying dimension is the next to last index, and so forth (this follows C convention).[1] Thus, a 2-D array is indexed [row, col]. Slicing rules also work as applied for each dimension (e.g., a colon selects all elements in that dimension). Here's an example:

Example 13 (Multidimensional array indexing and slicing):

Consider the following typed into a Python interpreter:

[1]See http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html and the definition of "row-major" in http://docs.scipy.org/doc/numpy/glossary.html (both accessed August 9, 2012).

```
import numpy as np
a = np.array([[2, 3.2, 5.5, -6.4, -2.2, 2.4],
             [1,  22,   4,  0.1,  5.3,  -9],
             [3,   1, 2.1,   21,  1.1,  -2]])
```

What is `a[1,2]` equal to? `a[:,3]`? `a[1,:]`? `a[1,1:4]`?

Solution and discussion: You should have obtained:

```
>>> print(a[1,2])
4.0
>>> print(a[:,3])
[ -6.4   0.1  21. ]
>>> print(a[1,:])
[  1.   22.    4.    0.1   5.3  -9. ]
>>> print(a[1,1:4])
[ 22.    4.    0.1]
```

Note that when I typed in the array I did not use the line continuation character at the end of each line because I was entering in a list, and by starting another line after I typed in a comma, Python automatically understood that I had not finished entering the list and continued reading the line for me.

4.3.3 Array inquiry

Some information about arrays comes through functions that act on arrays; other information comes through attributes attached to the array object. Let's look at some array inquiry examples:

Example 14 (Array inquiry):

Import NumPy as the alias `np` and create a 2-D array `a`. Below are some array inquiry tasks and the Python code to conduct these tasks. Try these commands out in your interpreter and see if you get what you expect.

- Return the shape of the array: `np.shape(a)`
- Return the number of dimensions of the array: `np.ndim(a)`
- Return the number of elements in the array: `np.size(a)`
- Typecode of the array: `a.dtype.char`

Solution and discussion: Here are some results using the example array from Example 13:

```
>>> print(np.shape(a))
(3, 6)
>>> print(np.ndim(a))
2
>>> print(np.size(a))
18
>>> print(a.dtype.char)
d
```

Note that you should *not* use `len` for returning the number of elements in an array. Also, the `size` function returns the total number of elements in an array. Finally, `a.dtype.char` is an example of an array attribute; notice there are no parentheses at the end of the specification because an attribute variable is a piece of data, not a function that you call.

The neat thing about array inquiry functions (and attributes) is that you can write code to operate on an array in general instead of a specific array of given size, shape, etc. This allows you to write code that can be used on arrays of all types, with the exact array determined at run time.

4.3.4 Array manipulation

In addition to finding things about an array, NumPy includes many functions to manipulate arrays. Some, like `transpose`, come from linear algebra, but NumPy also includes a variety of array manipulation functions that enable you to massage arrays into the form you need to do the calculations you want. One common such function is the `reshape` function which, as you might expect, takes an array and gives it a new shape (but doesn't change its total number of elements). For instance, re-consider this 2-D array:

```
import numpy as np
a = np.array([[2, 3.2, 5.5, -6.4, -2.2, 2.4],
              [1,  22,   4,  0.1,  5.3,  -9],
              [3,   1, 2.1,   21,  1.1,  -2]])
```

If we type `np.shape(a)`, we find this is a `(3, 6)` shape array (i.e., three rows and six columns). That equals 18 elements. We can reshape this into a 2 slice, 3 row, and 3 column array using `reshape`. This reshaped array has the shape `(2, 3, 3)`. Note that the total number of elements of this reshaped array is the same as the original array, i.e., 18 elements. The syntax for using `reshape` is `reshape(`*input_array*`, `*new_shape*`)`. Thus, the example described in this paragraph, when typed into the interpreter, would be:

```
>>> import numpy as np
>>> a = np.array([[2, 3.2, 5.5, -6.4, -2.2, 2.4],
...              [1,  22,   4,  0.1,  5.3,  -9],
...              [3,   1, 2.1,   21,  1.1,  -2]])
>>> b = np.reshape(a, (2, 3, 3))
>>> b
array([[[  2. ,   3.2,   5.5],
        [ -6.4,  -2.2,   2.4],
        [  1. ,  22. ,   4. ]],

       [[  0.1,   5.3,  -9. ],
        [  3. ,   1. ,   2.1],
        [ 21. ,   1.1,  -2. ]]])
```

Note that when you re-shape an array, you do not move the elements around in memory. All reshaping does is change when you decide to stop a row, a slice, etc. Put another way, the 4 and 0.1 in array a above are stored next to each other in the computer's memory, but so too are the 4 and 0.1 in array b, even though in array b the 4 ends a row and slice. In that sense, row, slice, etc. breaks are not "hard-wired" into the way the computer stores the array but are rather "bookmarks" the computer keeps track of to allow us to use array slicing (as we saw in Example 13).

4.3.5 Calculations using array contents

So far we've learned how to make arrays and ask arrays to tell us about themselves. Here we talk about making calculations of the contents of an array. We'll talk about two ways of doing so. The first method uses `for` loops, and we saw this being used in Section 4.2. The second method uses something called **array syntax**, where looping over array elements happens implicitly. The first method is what Java uses. This will probably be the first time you've seen the second method. You'll see that the second method, however, is very powerful.

Calculations using array contents: Method 1 (loops)

The tried-and-true method of doing arithmetic operations on arrays is to use loops to examine each array element one-by-one, do the operation, and then save the result. Depending on what the result is, the result might be saved in a scalar variable (i.e., just a number) or in a results array. Here's an example:

Example 15 (Multiply two arrays, element-by-element, using loops):

Consider this code:

```
1  import numpy as np
2  a = np.array([[2, 3.2, 5.5, -6.4],
3                [3,   1, 2.1,   21]])
4  b = np.array([[4, 1.2,  -4,  9.1],
5                [6,  21, 1.5,  -27]])
6  shape_a = np.shape(a)
7  product_ab = np.zeros(shape_a, dtype='f')
8  for i in range(shape_a[0]):
9      for j in range(shape_a[1]):
10         product_ab[i,j] = a[i,j] * b[i,j]
```

Can you describe what is happening in each line?

Solution and discussion: In the first four lines after the `import` line (lines 2–5), I create arrays `a` and `b`. They are both two row, four column arrays. In the sixth line, I read the shape of array `a` and save it as the variable `shape_a`. Note that `shape_a` is the tuple (2,4). In the seventh line, I create a results array of the same shape of `a` and `b`, of single-precision floating point type, and with each element filled with zeros. In the last three lines (lines 8–10), I loop through all rows (the number of which is given by `shape_a[0]`) and all columns (the number of which is given by `shape_a[1]`), by index. Thus, `i` and `j` are set to the element addresses for rows and columns, respectively, and line 10 does the multiplication operation and sets the product in the results array `product_ab` using the element addresses.

One other note: In this example, I make the assumption that the shape of `a` and the shape of `b` are the same, but I should instead add a check that this is actually the case. While a check using an `if` statement condition such as:

```
np.shape(a) != np.shape(b)
```

will work, because equality between sequences is true if all corresponding elements are equal,[2] things get tricky, fast, if you are interested in more complex logical comparisons and boolean operations for arrays. For instance, the logic that works for `!=` doesn't apply to built-in Python boolean operators such as `and`. We'll see later on in Section 4.6.2 how to do element-wise boolean operations on arrays.

So, why wouldn't you want to use the looping method for general array operations? In three and a half words: Loops are (relatively) s-l-o-w. Thus, if you can at all help it, it's better to use array syntax for general array operations: your code will be faster, more flexible, and easier to read and test.

Calculations using array contents: Method 2 (array syntax)

The basic idea behind array syntax is that, much of the time, arrays interact with each other on a corresponding element basis, and so instead of requiring the user to write out the nested `for` loops

[2] See the "Built-in Types" entry in the online Python documentation at http://docs.python.org/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange (accessed March 26, 2012).

explicitly, the loops and element-wise operations are done implicitly in the operator. That is to say, instead of writing this code (assume arrays a and b are 1-D arrays of the same size):

```
c = np.zeros(np.shape(a), dtype='f')
for i in range(np.size(a)):
    c[i] = a[i] * b[i]
```

array syntax means you can write this code:

```
c = a * b
```

Let's try this with a specific example using actual numbers:

Example 16 (Multiply two arrays, element-by-element, using array syntax):
Type the following in a file and run it using the Python interpreter:

```
import numpy as np
a = np.array([[2, 3.2, 5.5, -6.4],
              [3,   1, 2.1,   21]])
b = np.array([[4, 1.2,  -4,  9.1],
              [6,  21, 1.5,  -27]])
product_ab = a * b
```

What do you get when you print out product_ab?

Solution and discussion: You should get something like this:

```
>>> print(product_ab)
[[   8.      3.84  -22.    -58.24]
 [  18.     21.      3.15 -567.  ]]
```

In this example, we see that arithmetic operators are automatically defined to act element-wise when operands are NumPy arrays or scalars. (Operators do have function equivalents in NumPy, e.g., product, add, etc., for the situations where you want to do the operation using function syntax.) Additionally, the output array c is automatically created on assignment; there is no need to initialize the output array using zeros.

There are three more key benefits of array syntax. First, operand shapes are automatically checked for compatibility, so there is no need to check for that explicitly. Second, you do not need to know the number of dimensions of the arrays ahead of time, so the same line of code works on 1-D, 2-D, 3-D, etc. arrays. Finally, the array syntax formulation runs faster than the equivalent code using loops! Simpler, better, faster: pretty cool, eh? ☺

Let's try another array syntax example:

Example 17 (Another array syntax example):
Type the following in a Python interpreter:

```
import numpy as np
a = np.arange(10)
b = a * 2
c = a + b
d = c * 2.0
```

What results? Predict what you think a, b, and c will be, then print out those arrays to confirm whether you were right.

Solution and discussion: You should get something like this:

```
>>> print(a)
[0 1 2 3 4 5 6 7 8 9]
>>> print(b)
[ 0  2  4  6  8 10 12 14 16 18]
>>> print(c)
[ 0  3  6  9 12 15 18 21 24 27]
>>> print(d)
[  0.   6.  12.  18.  24.  30.  36.  42.  48.  54.]
```

Arrays a, b, and c are all integer arrays because the operands that created those arrays are all integers. Array d, however, is floating point because it was created by multiplying an integer array by a floating point scalar. Python automatically chooses the type of the new array to retain, as much as possible, the information found in the operands.

4.4 Correlation and lag autocorrelation

The correlation coefficient between two sets of data (r_{xy}) is given as:[3]

$$r_{xy} = \frac{\mathrm{cov}(x,y)}{\sigma_x \sigma_y} = \frac{\sum_{i=1}^{n} (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2 \sum_{i=1}^{n} (y_i - \overline{y})^2}}$$

The correlation coefficient tells us the degree to which two variables are linearly related to each other. If the value of the correlation coefficient is 1, the two variables are tightly related and increases in one variable are connected to increases in the other (**"correlated"**). If the value of the correlation coefficient is -1, the two variables are also tightly related but where increase in one

[3]https://en.wikipedia.org/w/index.php?title=Correlation_and_dependence&oldid=739479314 (accessed September 24, 2016)

variable are connected to *decreases* in the other (**“anti-correlated”**). If the value of the correlation coeffience is 0, the two variables are not tightly related to each other (**“uncorrelated”**).

SciPy has a function called `corrcoef` that returns the correlation matrix. In the case of two one-dimensional timeseries (that is, two single sequences of data values), the correlation coefficient is the value of the **“off-diagonal”** elements, of which there are two, and both of which are the same value. Thus, the following code returns the correlation coefficient between two one-dimensional arrays `x` and `y` (both arrays of the same length):

```
import scipy
scipy.corrcoef(x, y)[0,1]
```

You can verify that this result is the same as generated by the equation above.

In this chapter, however, we are dealing with the question of how to do NumPy arrays to help us do more complex data analysis; calling a function isn’t really more “complex.” Well, the reason we introduced the idea of correlation is to provide background for something that is a little more complex, and illustrates another use of slicing. And that something is lag autocorrelation.

The definition of the correlation coefficient above gives us the correlation coefficient between two variables, x and y. But what if x and y weren’t two different variables but the same variable, except shifted in time? That is to say, what if x and y were time series but where y were values of x shifted earlier or later? Then, the correlation coefficient we calculated would tell us the degree to which an earlier value is related to a later value. If we calculated the correlation coefficient for various degrees of shifting, we would have calculated the **lag autocorrelation** for the time series. (When we do this correlation calculation between two *different* shifted time series x and y it is called **lag correlation**. The “auto” in “lag autocorrelation” means “same time series. For our purposes, we’ll focus on lag autocorrelation, but the more interesting and useful measure is lag correlation. I encourage you to learn more about that technique someday on your own.)

Let’s take the subset of gasoline retail data in Section 4.2 and calculate and plot the lag autocorrelation for various number of months of lag. Here is code that would do the trick, using loops:

```
1  import numpy as np
2  import scipy
3  data_flat = np.ravel(data_as_array)
4  lags = np.arange(6)
5  correl = []
6  for ilag in lags:
7      x = np.zeros( (np.size(data_flat)-ilag) )
8      y = np.zeros( (np.size(data_flat)-ilag) )
9      for imonth in range(np.size(x)):
10         x[imonth] = data_flat[imonth]
11         y[imonth] = data_flat[imonth+ilag]
12     correl.append( np.corrcoef(x, y)[0,1] )
```

In line 3, we use the `ravel` function to turn the two-dimensional array into a one-dimensional array. In line 4 we make the lags we’re considering to be 0–5 months. The correlation coefficient will be between $f(t)$ vs. $f(t + \text{lag})$, and we hold the correlation coefficient values in the list `correl`.

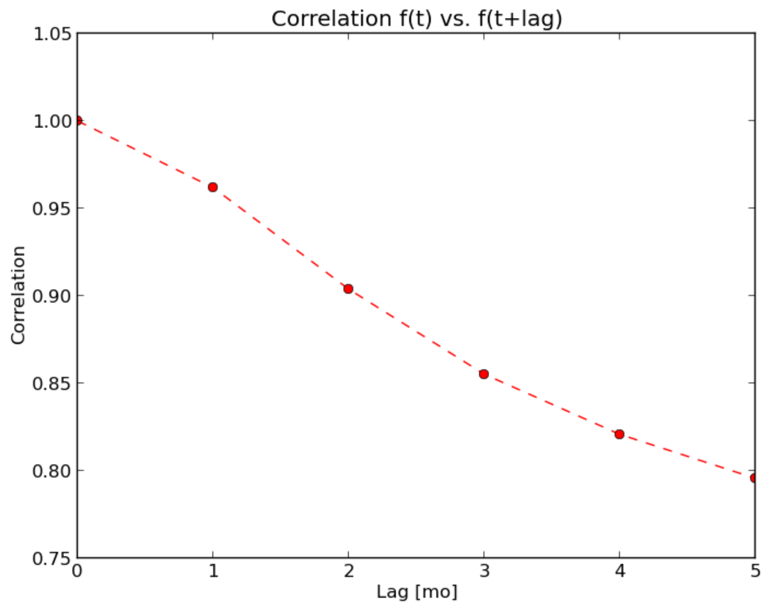


Figure 4.1: Lag autocorrelation plot for the data set in Section 4.4.

In lines 6–11, we loop through each lag value and make the temporary variables `x` and `y` values of `data_flat`, shifted by the lag value. Once we have the `x` and `y` arrays, we calculate the correlation coefficient for that lag.

A graph of the lag autocorrelation coefficient vs. the lag value is given in Figure 4.1. As expected, at no lag, the correlation coefficient is 1, since a time series is perfectly related to itself at the same time. As the lag increases, the correlation drops. This is consistent with what we'd expect, that is, that the relationship between the current month's gasoline retail sales with sales in the future will become weaker the farther out in the future you go. (Note, however, that we can't really say what these coefficient values *mean* without conducting a test of statistical significance. When doing statistical calculations, only statistically significant values compared to some reasonable baseline have meaning, in the statistical sense.)

Here's the solution using array slicing:

```
1   import numpy as np
2   import scipy
3   data_flat = np.ravel(data_as_array)
4   lags = np.arange(6)
5   correl = []
6   for ilag in lags:
7       if ilag == 0:
8           correl.append(1.0)
9       else:
10          x = data_flat[0:-ilag]
11          y = data_flat[ilag:]
12          correl.append( np.corrcoef(x, y)[0,1] )
```

It's similar to the solution using loops except the portion where we select the values from `data_flat` are done using array slicing, with adjustments for the lag value. Note that we pre-calculate the lag 0 case because the lag autocorrelation for no lag is always 1. We have to do this because the slicing syntax in lines 10–11 do not work if `ilag` equals 0. The slicing syntax in lines 10–11, however, are cleaner than the code in the loops solution (lines 7–11 in that solution), which does the same thing.

4.5 More complicated manipulations of the data for customized analyses

We can do a lot of analysis by using NumPy and SciPy functions on arrays and subarrays of data. Often times, however, we'll want to do more than extract the data or do simple operations on the data. Instead, we'll want to something more complicated. How can we use Python arrays to help us do this kind of analysis?

Let's pretend we wanted to take the subset of gasoline retail data in Section 4.2 and extract only the values meeting a certain criteria, say the monthly sales is greater than $40,000 million. The code below does this, and the list `data_gt_40k` contains only those values that are greater than 40,000:

```
1   import numpy as np
2   data_gt_40k = []
3   data_shape = np.shape(data_as_array)
4   for iyear in range(data_shape[0]):
5       for imonth in range(data_shape[1]):
6           if data_as_array[iyear, imonth] > 40000:
7               data_gt_40k.append(data_as_array[iyear, imonth])
```

In this code, we make use of nested loops. That is, the loop defined in line 5 is run for every iteration of the loop defined in line 4. In this way we go through all the data in the two-dimensional array `data_as_array`. The `if` test in line 6 is conducted on every element in `data_as_array`. Note that in Python, the nesting of loops is defined by the indentation. If you do not indent the inner loop, you won't have a nested loop but rather two separate loops.

Note that the above code would also work on the full data in Table 4.1, if that data was set to `data_as_array`, since we make use of array inquiry functions to determine how many times to loop through the array.

We saw in Section 4.3.5 that with array syntax we can implicitly loop through all elements of an array and make calculations with each element of the array, one element at a time, without writing an actual `for` statement. But what about `if` tests? Is there a way we can do testing inside an array while using array syntax? That way, we can get the benefits of simpler code, the flexibility of code that works on arrays of any number of dimensions, and speed. The answer is, yes! NumPy has comparison and boolean operators that act element-wise and array inquiry and selection functions. Here's the code to do the above task without explicit `for` loops:

```
1  import numpy as np
2  is_gt_40k_indices = np.where(data_as_array > 40000)
3  data_gt_40k = data_as_array[is_gt_40k_indices]
```

The expression `data_as_array > 40000` produces an array of the same size and shape as `data_as_array` but whose elements are either `True` or `False`, depending on whether or not the corresponding element in `data_as_array` is greater than 40000. The NumPy `where` function, when called with a single boolean array as a parameter, returns a data structure that specifies the indices where the input parameter is true. That data structure can then be used when specifying indices in an array (line 3) to select those elements elements of the array.

This use of boolean array expressions and `where` only touches on ways we can do tests while using array syntax. Section 4.6.2 goes into more detail on this topic.

4.6 Python Sidebar: Testing inside an array

We've seen there are two ways of doing calculations or manipulations of an array that involves conditionals. The first is to implement this in a loop. A second way is to use array syntax and take advantage of comparison operators and specialized NumPy search functions. In this section we dive more deeply into these methods.

4.6.1 Testing inside an array: Method 1 (loops)

In this method, you apply a standard conditional (e.g., `if` statement) while inside the nested `for` loops running through the array. This is similar to traditional Fortran syntax. Here's is an example:

Example 18 (Using looping to test inside an array):

Say you have a 2-D array `a` and you want to return an array `answer` which is double the value of the corresponding element in `a` when the element is greater than 5 and less than 10, and zero when the value of that element in `a` is not. What's the code for this task?

Solution and discussion: Here's the code:

```
answer = np.zeros(np.shape(a), dtype='f')
for i in range(np.shape(a)[0]):
    for j in range(np.shape(a)[1]):
        if (a[i,j] > 5) and (a[i,j] < 10):
            answer[i,j] = a[i,j] * 2.0
        else:
            pass
```

The `pass` command is used when you have a block statement (e.g., a block `if` statement, etc.) where you want the interpreter to do nothing. In this case, because `answer` is filled with all zeros on initialization, if the `if` test condition returns `False`, we want that element of `answer` to be zero. But, all elements of `answer` start out as zero, so the `else` block has nothing to do; thus, we `pass`.

Again, while this code works, loops are slow, and the `if` statement makes it even slower. The nested `for` loops also mean that this code will only work for a 2-D version of the array `a`.

4.6.2 Testing inside an array: Method 2 (array syntax)

As we've seen, because NumPy has comparison and boolean operators that act element-wise and array inquiry and selection functions, we can write a variety of ways of testing and selecting inside an array while using array syntax. Before we discuss some of those ways, we need some context about using NumPy comparison operators and boolean array functions.

NumPy comparison operators and boolean array functions

NumPy has defined the standard comparison operators in Python (e.g., ==, <) to work element-wise with arrays. Thus, if you run these lines of code:

```
import numpy as np
a = np.arange(6)
print(a > 3)
```

the following array is printed out to the screen:

```
[False False False False True True]
```

Each element of the array a that was greater than 3 has its corresponding element in the output set to `True` while all other elements are set to `False`. You can achieve the same result by using the corresponding NumPy function `greater`. Thus:

```
print(np.greater(a, 3))
```

gives you the same thing. Other comparison functions are similarly defined for the other standard comparison operators; those functions also act element-wise on NumPy arrays.

Once you have arrays of booleans, you can operate on them using boolean operator NumPy functions. You cannot use Python's built-in `and`, `or`, etc. operators; those will not act element-wise. Instead, use the NumPy functions `logical_and`, `logical_or`, etc. Thus, if we have this code:

```
a = np.arange(6)
print(np.logical_and(a>1, a<=3))
```

the following array will be printed to screen:

```
[False False True True False False]
```

The `logical_and` function takes two boolean arrays and does an element-wise boolean "and" operation on them and returns a boolean array of the same size and shape filled with the results.

With this background on comparison operators and boolean functions for NumPy arrays, we can talk about ways of doing testing and selecting in arrays while using array syntax. Here are two methods: using the `where` function and using arithmetic operations on boolean arrays.

The `where` function

The Python version of `where`, can be used in two ways: To directly select corresponding values from another array (or scalar), depending on whether a condition is true, and to return a list of array element indices for which a condition is true (which then can be used to select the corresponding values by selection with indices).

The syntax for using `where` to directly select corresponding values is the following:

`np.where(`*<condition>*`,` *<value if true>*`,` *<value if false>*`)`

If an element of *<condition>* is `True`, the corresponding element of *<value if true>* is used in the array returned by the function, while the corresponding element of *<value if false>* is used if *<condition>* is `False`. The `where` function returns an array of the same size and shape as *<condition>* (which is an array of boolean elements). Here is an example to work through:

Example 19 (Using `where` to directly select corresponding values from another array or scalar):

Consider the following case:

```
import numpy as np
a = np.arange(8)
condition = np.logical_and(a>3, a<6)
answer = np.where(condition, a*2, 0)
```

What is `condition`? `answer`? What does the code do?

Solution and discussion: You should get:

```
>>> print(a)
[0 1 2 3 4 5 6 7]
>>> print(condition)
[False False False False  True  True False False]
>>> print(answer)
[ 0  0  0  0  8 10  0  0]
```

The array `condition` shows which elements of the array `a` are greater than 3 and less than 6. The `where` call takes every element of array `a` where that is true and doubles the corresponding value of `a`; elsewhere, the output element from `where` is set to 0.

The second way of using `where` is to return a tuple of array element indices for which a condition is true, which then can be used to select the corresponding values by selection with indices.[4] For 1-D arrays, the tuple is a one-element tuple whose value is an array listing the indices where the condition is true. For 2-D arrays, the tuple is a two-element tuple whose first value is an array listing the row index where the condition is true and the second value is an array listing the column index where the condition is true. In terms of syntax, you tell `where` to return indices instead of an array of selected values by calling `where` with only a single argument, the *<condition>* array. To select those elements in an array, pass in the tuple as the argument inside the square brackets (i.e., `[]`) when you are selecting elements. Here is an example:

Example 20 (Using `where` to return a list of indices):

Consider the following case:

```
import numpy as np
a = np.arange(8)
condition = np.logical_and(a>3, a<6)
answer_indices = np.where(condition)
answer = (a*2)[answer_indices]
```

What is `condition`? `answer_indices`? `answer`? What does the code do?

Solution and discussion: You should have obtained similar results as Example 19, except the zero elements are absent in `answer` and now you also have a tuple of the indices where `condition` is true:

[4]This is like the behavior of IDL's `WHERE` function.

```
>>> print(a)
[0 1 2 3 4 5 6 7]
>>> print(condition)
[False False False False  True  True False False]
>>> print(answer_indices)
(array([4, 5]),)
>>> print(answer)
[ 8 10]
```

The array `condition` shows which elements of the array `a` are greater than 3 and less than 6. The `where` call returns the indices where `condition` is true, and since `condition` is 1-D, there is only one element in the tuple `answer_indices`. The last line multiplies array `a` by two (which is also an array) and selects the elements from that array with addresses given by `answer_indices`.

Note that selection with `answer_indices` will give you a 1-D array, even if `condition` is not 1-D. Let's turn array `a` into a 3-D array, do everything else the same, and see what happens:

```
import numpy as np
a = np.reshape( np.arange(8), (2,2,2) )
condition = np.logical_and(a>3, a<6)
answer_indices = np.where(condition)
answer = (a*2)[answer_indices]
```

The result now is:

```
>>> print(a)
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
>>> print(condition)
[[[False False]
  [False False]]

 [[ True  True]
  [False False]]]
>>> print(answer_indices)
(array([1, 1]), array([0, 0]), array([0, 1]))
>>> print(answer)
[ 8 10]
```

Note how `condition` is 3-D and the `answer_indices` tuple now has three elements (for the three dimensions of `condition`), but `answer` is again 1-D.

Arithmetic operations using boolean arrays

You can also accomplish much of what the `where` function does in terms of testing and selecting by taking advantage of the fact that arithmetic operations on boolean arrays treat `True` as 1 and `False` as 0. By using multiplication and addition, the boolean values become selectors, because any value multiplied by 1 or added to 0 is that value. Let's see an example of how these properties can be used for selection:

Example 21 (Using arithmetic operators on boolean arrays as selectors):
Consider the following case:

```
import numpy as np
a = np.arange(8)
condition = np.logical_and(a>3, a<6)
answer = ((a*2)*condition) + \
         (0*np.logical_not(condition))
```

Solution and discussion: The solution is the same as Example 19:

```
>>> print(a)
[0 1 2 3 4 5 6 7]
>>> print(condition)
[False False False False  True  True False False]
>>> print(answer)
[ 0  0  0  0  8 10  0  0]
```

But how does this code produce this solution? Let's go through it step-by-step. The `condition` line is the same as in Example 19, so we won't say more about that. But what about the `answer` line? First, we multiply array `a` by two and then multiply that by `condition`. Every element that is `True` in `condition` will then equal double of `a`, but every element that is `False` in `condition` will equal zero. We then add that to zero times the `logical_not` of `condition`, which is `condition` but with all `True`s as `False`s, and vice versa. Again, any value that multiplies by `True` will be that value and any value that multiplies by `False` will be zero. Because `condition` and its "logical not" are mutually exclusive—if one is true the other is false—the sum of the two terms to create `answer` will select either `a*2` or `0`. (Of course, the array generated by `0*np.logical_not(condition)` is an array of zeros, but you can see how multiplying by something besides `0` will give you a different replacement value.)

Also, note the continuation line character is a backslash at the end of the line (as seen in the line that assigns `answer`).

This method of testing inside arrays using arithmetic operations on boolean arrays is also faster than loops.

4.6.3 Additional array functions

The functions SciPy and NumPy provide that act on scalars also act on arrays, element-wise. These include basic mathematical functions (`sin`, `exp`, `interp`, etc.) and basic statistical functions (`correlate`, `histogram`, `hamming`, `fft`, etc.). For more complete lists of array functions, see the SciPy and NumPy manuals: http://docs.scipy.org/doc/. From the Python interpreter, you can also use `help(numpy)` as well as `help(numpy.`*x*`)`, where *x* is the name of a function, to get more information. (The syntax is equivalent for SciPy.)

4.7 ⌨ Python Sidebar: Floating point comparison

It turns out that when testing to see whether two numbers are the same, it it important to know whether the two numbers are integers or floating point numbers. Integers are exactly represented by a binary computer and so a test of equality between two integers will also work correctly. Floating point numbers on any binary computer, however, in general are not represented exactly.[5] For instance, if you type in:

```
>>> 3.5 * (-2.1)
-7.3500000000000005
```

The answer is not -7.35, as you'd expect. The inexactness of floating point representation in a computer can cause real problems when doing tests for equality. Consider these three tests:

```
>>> 3.0 == 3.0
True
>>> 3.0 == 3.00001
False
>>> 3.0 == 3.00000000000000000001
True
```

The computer may think that two floating point numbers, if they're close enough to each other, are the same. And, this behavior is not consistent between different computers and operating systems. What then should we do?

The answer is to never do logical equality comparisons between floating point numbers, but instead, we should compare whether two floating point numbers are "close to" each other. The NumPy array package has a function `allclose` that does this. The `allclose` function allows you to set what constitutes two numbers being "close," so you can do the closeness testing the same way everytime, regardless of what computer you're using. Here's an example:

[5]See Bruce Bush's article "The Perils of Floating Point," http://www.lahey.com/float.htm (accessed March 17, 2012).

```
>>> import numpy as np
>>> np.allclose(3.0, 3.0)
True
>>> np.allclose(3.0, 3.00001)
True
>>> np.allclose(3.0, 3.00001, atol=1e-6, rtol=1e-6)
False
```

When the two input values to `allclose` are arrays, the comparison is done element-wise between corresponding elements of the two arrays. If all such comparisons yield `True`, the function returns `True`. If even one of the element pairs are not close to each other, the function returns `False`:

```
>>> import numpy as np
>>> a = np.array([3., 6., -4.])
>>> b = np.array([3., 6., -4.001])
>>> np.allclose(a, b)
False
>>> b = np.array([3., 6., -4.000001])
>>> np.allclose(a, b)
True
```

To find out which element(s) is not "close" between the two arrays, use the NumPy function `isclose`:

```
>>> import numpy as np
>>> a = np.array([3., 6., -4.])
>>> b = np.array([3., 6., -4.001])
>>> np.isclose(a, b)
array([ True,  True, False], dtype=bool)
>>> b = np.array([3., 6., -4.000001])
>>> np.isclose(a, b)
array([ True,  True,  True], dtype=bool)
```

The resulting boolean array tells us which corresponding elements in the two arrays are close to each other. It takes the same `atol` and `rtol` keyword input parameters to change what counts as "close." See the documentation for the two routines for details as to how these parameters work.[6]

4.8 Python Sidebar: String indexing

In Section 4.3.2, we found that we can select individual elements and slices of arrays using the list/tuple indexing notation. It turns out, we can do the same with strings. That is, we can think of a string as a little 1-D array where each element is a character.

[6] See https://docs.scipy.org/doc/numpy/reference/generated/numpy.allclose.html and https://docs.scipy.org/doc/numpy/reference/generated/numpy.isclose.html.

Example 22 (String indexing):
Say you have the following string `mysaying`:

```
mysaying = "Buy low, sell high, and you'll do fine."
```

What is `mysaying[2]`? `mysaying[0:5]`? `mysaying[-5:]`? How would you extract the word "you'll"?

Solution and discussion: `mysaying[2]` returns a "y". `mysaying[0:5]` returns "Buy l". And `mysaying[-5:]` returns "fine.". To extract the word "you'll", `mysaying[24:30]` will work. The indexing syntax is the same for substrings as for 1-D arrays, including the meaning of ranges (the colon) and negative indices. Remember blank spaces and the apostrophe are all each characters in the string `mysaying`.

4.9 ⌨ Python Sidebar: A simple way of seeing how fast your code runs

The time module has a function `time` that returns the current system time relative to the Epoch (a date that is operating system dependent). If you save the current time as a variable before and after you execute your function/code, the difference is the time it took to run your function/code.

Example 23 (Using `time` to do timings):
Type in the following and run it in a Python interpreter:

```
import time
begin_time = time.time()
for i in range(1000000L):
    a = 2*3
print(time.time() - begin_time)
```

What does the number that is printed out represent?

Solution and discussion: The code prints out the amount of time (in seconds) it takes to multiply two times three and assign the product to the variable `a` one million times. (Of course, it also includes the time to do the looping, which in this simple case probably is a substantial fraction of the total time of execution.)

4.10 Summary

The NumPy array is a really powerful tool. It enables us to slice-and-dice data in any number of ways, enabling us to analyze data quickly. Especially for datasets that are more complicated than a single sequence of values, NumPy arrays enable us to get a handle on that data.

Chapter 5

Reading In and Writing out Text Data

Chapter Contents

We're almost done with the foundations of doing business data analysis using the tools in Python. In the previous chapters, the datasets we've looked at have been pretty small. Partly, this is because it's easier to get a handle on small datasets, and when you're learning a new tool, this helps out a lot. However, what makes using a programming language (as opposed to Excel) to analyze data is the ease with which a program written in a programming language can be scaled up to handle a large dataset. In Excel, it's not so easy to go from, say 500 rows of data to 500 million rows of data.

The key to getting a computer to operate on large amounts of data is in storing the data in a file and reading the data into an array (or similar variable). After you've put the data into an array, you can manipulate and do calculations on that data. For really large datasets, the data will be stored in a database in a format that helps you manage the data. SQL is perhaps the best-known general use database language, but there are plenty of others, customized for use in other use-cases.

In this chapter, we'll look at reading data that is stored in a text file. Text files are handy for storing files on the order of tens to hundreds of thousands of lines. Once you start hitting millions of lines, text files will still work, but they start becoming cumbersome. The ideas behind handling data in text files, however, are similar to other file formats. In addition, this gives us an opportunity to revisit string handling in Python, which in and of itself is one of Python's "superpowers."

5.1 File objects

A file object is a "variable" that represents the file to Python. (In Section 5.3, we'll talk in more detail about what an object is in a programming language.)

File objects are created like any other object in Python, that is, by assignment. For text files, you **instantiate** a file object with the built-in `open` statement:

```
fileobj = open('foo.txt', 'r')
```

The first argument in the `open` statement gives the filename. The second argument sets the mode for the file: `'r'` for reading-only from the file, `'w'` for writing a file, and `'a'` for appending to the file.

Once you've created the text file object, you can use various methods attached to the file object to interact with the file.

When you are done reading from or writing to a file, you can close it using the `close` method. Thus, to close a file object `fileobj`, execute:

```
fileobj.close()
```

5.2 Reading a file

To read one line from the file, use the `readline` method:

```
aline = fileobj.readline()
```

Because the file object is connected to a text file, `aline` will be a string. Note that `aline` contains the newline character, because each line in a file is terminated by the newline character.

To read the rest of a file that you already started reading, or to read an entire file you haven't started reading, and then put the read contents into a list, use the `readlines` method:

```
contents = fileobj.readlines()
```

Here, `contents` is a list of strings, and each element in `contents` is a line in the `fileobj` file. Each element also contains the newline character, from the end of each line in the file.

Note that the variable names `aline` and `contents` are not special; use whatever variable name you would like to hold the strings you are reading in from the text file.

5.3 ⌨ Python Sidebar: An introduction to objects

One good way of describing something new is to compare it with something old. Everyone who has taken a first term programming course has had experience with procedural programming, so we'll start there. Procedural programs look at the world in terms of two entities, "data" and "functions." In a procedural context, the two entities are separate from each other. A function takes data as input and returns data as output. Additionally, there's nothing customizable about a function with respect to data. As a result, there are no barriers to using a function on various types of data, even inappropriately.

In the real world, however, we don't think of things or objects as having these two features (data and functions) as separate entities. That is, real world objects are not (usually) merely data nor merely functions. Real world objects instead have both "state" and "behaviors." For instance, people have state (tall, short, etc.) and behavior (playing basketball, running, etc.), often both at the same time, and, of course, in the same person.

The aim of object-oriented programming is to imitate this in terms of software, so that "objects" in software have two entities attached to them, states and behavior. This makes the conceptual leap from real-world to programs (hopefully) less of a leap and more of a step. As a result, we can more easily implement ideas into instructions a computer can understand.

5.3.1 The nuts and bolts of objects

What do objects consist of? An object in programming is an entity or "variable" that has two entities attached to it: data and things that act on that data. The data are called attributes (also called **instance variables**) of the object, and the functions attached to the object that can act on the data are called methods of the object. Importantly, you *design* these methods to act on the attributes; they aren't random functions someone has attached to the object. In contrast, in procedural programming, variables have only one set of data, the value of the variable, with no functions attached to the variable.

How are objects defined? In the real world, objects are usually examples or specific realizations of some class or type. For instance, individual people are specific realizations of the class of human beings. The specific realizations, or instances, differ from one another in details but have the same pattern. For people, we all have the same general shape, organ structure, etc. In **object-oriented programming (OOP)**, the specific realizations are called object **instances,** while the common pattern is called a **class.** In Python, this common pattern or template is defined by the `class` statement. (We describe the `class` statement in Section 9.2.)

So, in summary, objects are made up of attributes and methods, the structure of a common pattern for a set of objects is called its class, and specific realizations of that pattern are called "instances of that class."

All the Python "variables" we introduced earlier are actually objects. (In fact, basically everything in Python is an object.) Let's look at a number of different Python objects to illustrate how objects work.

5.3.2 Example of how objects work: Strings

In Section 1.3.2 we were introduced to strings. What we didn't mention then is that Python strings (like nearly everything else in Python) are objects. Thus, built into Python, there (implicitly) is a class definition of the string class, and every time you create a string, you are using that definition as your template. That template defines both attributes and methods for all string objects, so whatever string you've created, you have that set of data and functions attached to your string which you can use. Let's look at a specific case:

Example 24 (Viewing attributes and methods attached to strings and trying out a few methods):

In the Python interpreter, type in:

```
a = "hello"
```

Now type: `dir(a)`. What do you see? Type `a.title()` and `a.upper()` and see what you get.

Solution and discussion: The `dir(a)` command gives a list of (nearly) all the attributes and methods attached to the object `a`, which is the string `"hello"`. (An Internet search for "python string attributes methods" would also probably give you a similar list.) Note that there is more data attached to the object than just the word "hello", e.g., the attributes `a.__doc__` and `a.__class__` also show up in the `dir` listing.

Methods can act on the data in the object. Thus, `a.title()` applies the `title` method to the data of `a` and returns the string `"hello"` in title case (i.e., the first letter of the word capitalized); `a.upper()` applies the `upper` method to the data of `a` and returns the string all in uppercase. Notice these methods do not require additional input arguments between the parenthesis, because all the data needed is already in the object (i.e., `"hello"`).

The syntax for objects looks very similar to the syntax for data and functions in modules. First, to refer to attributes or methods of an instance, you add a period after the object name and then put the attribute or method name. To set an attribute, the reference should be on the lefthand side of the equal sign; the opposite is the case to read an attribute. Method calls require you to have parentheses after the name, with or without arguments, just like a function call. Finally, methods can produce a return value (like a function), act on attributes of the object in-place, or both.

5.3.3 Example of how objects work: Arrays

We've already been introduced to arrays, but most of our discussion of arrays has focused on functions that create and act on arrays. Arrays, however, are objects like any other object and have attributes and methods built-in to them; arrays are more than just a sequence of numbers. Let's look at an example list of all the attributes and methods of an array object:

Example 25 (Examining array object attributes and methods):

In the Python interpreter, type in:

```
a = np.reshape(np.arange(12), (4,3))
```

Now type: `dir(a)`. What do you see? Based on their names, and your understanding of what arrays are, what do you think some of these attributes and methods do?

Solution and discussion: The `dir` command should give you a list of a lot of stuff. I'm not going to list all the output here but instead will discuss the output in general terms.

We first notice that there are two types of attribute and method names: those with double-underscores in front and in back of the name and those without any pre- or post-pended double-underscores. We consider each type of name in turn.

A very few double-underscore names sound like data. The `a.__doc__` variable is one such attribute and refers to documentation of the object. Most of the double-underscore names suggest operations on or with arrays (e.g., add, div, etc.), which is what they are: Those names are of the methods of the array object that *define* what Python will do to your data when the interpreter sees a "+", "/", etc. Thus, if you want to redefine how operators operate on arrays, *you can do so.* It is just a matter of redefining that method of the object.

That being said, I do not, in general, recommend you do so. In Python, the double-underscore in front means that attribute or method is "very private." (A variable with a single underscore in front is private, but not as private as a double-underscore variable.) That is to say, it is an attribute or method that normal users should not access, let alone redefine. Python does not, however, do much to prevent you from doing so, so advanced users who need to access or redefine those attributes and methods can do so.

The non-double-underscore names are names of "public" attributes and methods, i.e., attributes and methods normal users are expected to access and (possibly) redefine. A number of the methods and attributes of `a` are duplicates of functions (or the output of functions) that act on arrays (e.g., `transpose`, `T`), so you can use either the method version or the function version.

And now let's look at some examples of accessing and using array object attributes and methods:

Example 26 (Using array attributes and methods):

In the Python interpreter, type in:

```
a = np.reshape(np.arange(12), (4,3))
print(a.astype('c'))
print(a.shape)
print(a.cumsum())
print(a.T)
```

What do each of the `print` lines do? Are you accessing an attribute or method of the array?:

Solution and discussion: The giveaway as to whether we are accessing attributes or calling methods is whether there are parenthesis after the name; if not, it's an attribute, otherwise, it's a method. Of course, you could type the name of the method without parentheses following, but then the interpreter would just say you specified the method itself, as you did not *call* the method:

```
>>> print(a.astype)
<built-in method astype of numpy.ndarray object at 0x20d5100>
```

That is to say, the above syntax prints the method itself; since you can't meaningfully print the method itself, Python's `print` command just says "this is a method."

The `astype` call produces a version of array `a` that converts the values of `a` into single-character strings. The `shape` attribute gives the shape of the array. The `cumsum` method returns a flattened version of the array where each element is the cumulative sum of all the elements before. Finally, the attribute `T` is the transpose of the array `a`.

While it's nice to have a bunch of array attributes and methods attached to the array object, in practice, I find I seldom access array attributes and find it easier to use NumPy functions instead of the corresponding array methods. One exception with regards to attributes is the `dtype.char` attribute, which we've already seen; that's very useful since it tells you the type of the elements of the array.

5.4 Writing a file

To write a string to the file that is defined by the file object `fileobj`, use the `write` method attached to the file object:

```
fileobj.write(astr)
```

Here, `astr` is the string you want to write to the file. Note that a newline character is *not* automatically written to the file after the string is written. If you want a newline character to be added, you have to append it to the string prior to writing (e.g., `astr+'\n'`).

To write a list of strings to the file, use the `writelines` method:

```
fileobj.writelines(contents)
```

Here, `contents` is a list of strings, and, again, a newline character is *not* automatically written after the string (so you have to explicitly add it if you want it written to the file).

5.5 Processing file contents

Let's say you've read-in the contents of a file from the file and now have the file contents as a list of strings. How do you do things with them? In particular, how do you turn them into numbers (or arrays of numbers) that you can analyze? Python has a host of string manipulation methods, built-in to string variables (a.k.a., objects), which are ideal for dealing with contents from text files. We will mention only a few of these methods.

The `split` method of a string object takes a string and breaks it into a list using a separator. For instance:

```
a = '3.4 2.1 -2.6'
print(a.split(' '))
['3.4', '2.1', '-2.6']
```

will take the string `a`, look for a blank space (which is passed in as the argument to `split`, and use that blank space as the delimiter or separator with which one can split up the string. If `split` has no parameters, the method removes all whitespace of any kind between the non-whitespace characters of the string.

The `join` method takes a separator string and puts it between items of a list (or an array) of strings. For instance:

```
a = ['hello', 'there', 'everyone']
'\t'.join(a)
'hello\tthere\teveryone'
```

will take the list of strings `a` and concatenate these elements together, using the tab string (`'\t'`) to separate two elements from each other.

Finally, once we have the strings we desire, we can convert them to numerical types in order to make calculations. Here are two ways of doing so:

- If you loop through a list of strings, you can use the `float` and `int` functions on the string to get a number. For instance:

  ```
  import numpy as np
  a = ['3.4', '2.1', '-2.6']
  anum = np.zeros(len(a), 'd')
  for i in range(len(a)):
      anum[i] = float(a[i])
  ```

  takes a list of strings `a` and turns it into a NumPy array of double-precision floating point numbers `anum`.[1]

- If you make the list of strings a NumPy array of strings, you can use the `astype` method for type conversion to floating point or integer. For instance:

  ```
  anum = np.array(a).astype('d')
  ```

  takes a list of strings `a`, converts it from a list to an array of strings using the `array` function, and turns that array of strings into an array of double-precision floating point numbers `anum` using the `astype` method of the array of strings.

[1]Note that you can specify the array `dtype` without actually writing the `dtype` keyword; NumPy array constructors like `zeros` will understand a typecode given as the second positional input parameter.

A gotcha: Different operating systems may set the end-of-line character to something besides `'\n'`. Make sure you know what your text file uses. (For instance, MS-DOS uses `'\r\n'`, which is a carriage return followed by a line feed.) By the way, Python has a platform independent way of referring to the end-of-line character: the attribute `linesep` in the module os. If you write your program using that variable, instead of hard-coding in `'\n'`, your program will write out the specific end-of-line character for the system you're running on.

Example 27 (Writing and reading a single column file):

Take the following list of prices `prices`:

```
prices = [1.99, 24.95, 4.99, 17.95]
```

write it to a file *one-col_prices.txt*, then read the file back in.

Solution and discussion: This code will do the trick (note I use comment lines to help guide the reader):

```
import numpy as np

outputstr = ['\n']*len(prices)    #- Convert to string
for i in range(len(prices)):      #  and add newlines
    outputstr[i] = \
        str(prices[i]) + outputstr[i]

fileout = open('one-col_prices.txt', 'w')  #- Write out
fileout.writelines(outputstr)              #  to the
fileout.close()                            #  file

filein = open('one-col_prices.txt', 'r')   #- Read in
inputstr = filein.readlines()              #  from the
filein.close()                             #  file

prices_new = np.zeros(len(inputstr), 'f')  #- Convert
for i in range(len(inputstr)):             #  string to
    prices_new[i] = float(inputstr[i])     #  numbers
```

Note you don't have to strip off the newline character before converting the number to floating point using `float`.

While a multi-column text file of data seems more formidable than a single column text file, with the `split` method, it's not all that more complex, as we see in the following example.

Example 28 (Reading in a multi-column text file):

Let's look again at the gasoline sales dataset given in Table 4.1. That dataset is available as a file here: http://www.johnny-lin.com/infosys/adv44700_sales_only.txt. How would we read the file in and put the data into an two-dimensional array? To keep things simple, we'll discard the 2016 data, since the file does not cover that year completely.

Solution and discussion: In preparing to write our code, we first note that the first two lines contain header information rather than data. Second, we note that columns are separated by variable amounts of whitespace. That won't pose a problem because the `split` method will intelligently eliminate the whitespace between the columns.

With this as background, here's a solution:

```
1  import numpy as np
2
3  fileobj = open('adv44700_sales_only.txt', 'r')
4  data_str = fileobj.readlines()
5  fileobj.close()
6
7  data = np.zeros((len(data_str)-3, 13), dtype='l')
8  for i in range(len(data_str)-3):
9      split_istr = data_str[i+2].split()
10     data[i,:] = split_istr[:]
```

The array `data` holds all the numerical data in the file from 1992–2015 and has the same structure as Table 4.1, without the headers. The first column of `data` are the years, and the subsequent columns are the data for each month over all the years.

A few items: In lines 7 and 8, the number of rows in the `data` array and, subsequently, the number of rows we loop through are three less than the number of lines in the file. This is because we discard the first two lines as headers and the last line as an incomplete year. In line 9, the row index we are examining is not `i` but `i+2` because we are skipping the first two elements (i.e., rows) in `data_str`.

5.6 Catching file opening errors

What happens if you try to open a file and that file doesn't exist? Python sends you a message telling you what's wrong and stops the normal execution of the program. This process is called throwing an **exception**. For instance, if we tried to execute this line of code:

```
fileobj = open('not\_a\_file.txt', 'r')
```

and the file *not_a_file.txt* does not exist, the following will be returned by the interpreter:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'not_a_file.txt'
```

and the program will stop.

For many purposes, this works fine. If you run a program and get this message, you probably will go in and change the filename in the code and re-run the program. But what if you *expected* this error to occur and had a plan to deal with the error? For instance, what if you knew that a data file you wanted to open was named either *data.txt* or *DATA.TXT*, but you didn't know beforehand which it would be? This might occur because the program that generated the data named the file differently depending on what computer it was run on (historically, some programming languages were in all-caps). In such a situation, in Python you can use the `try`...`except` structure to gracefully handle the exception. This code:

```
try:
    fileobj = open('data.txt', 'r')
except IOError:
    fileobj = open('DATA.TXT', 'r')
```

will try to open *data.txt* and if it fails with an `IOError`, will execute the code in the `except` block, which tries to open the file *DATA.TXT*.

The `try`...`except` structure is a useful way of gracefully handling errors of many kinds, not just file errors. In Section 5.7 below, we look a little more closely at Python's exception handling mechanism.

5.7 ⌨ Python Sidebar: More on exception handling

Let's say you're writing a program and want the execution of the program to stop when a certain condition occurs. How can you manually raise an error? In Python, you do this with a `raise` statement. Here's an example:

Example 29 (Using `raise`):

Consider a function to calculate the area of a circle given a user-inputted radius. How would we put in a test to ensure the user would not pass in a negative radius? One answer: We could put in an `if` test for a negative radius and if true, execute a `raise` statement:

```
def area(radius, pi=3.14):
    if radius < 0:
        raise ValueError('radius negative')
    area = pi * (radius**2)
    return area
```

The syntax for `raise` is the command `raise` followed by an exception class (in this case I used the built-in exception class `ValueError`, which is commonly used to denote errors that have to do with bad variable values), then a comma and a string that will be output by the interpreter when the `raise` is thrown.

So far, I've been loosely saying that an exception "stops" the execution of a program. Actually, raising an exception is not exactly the same as stopping the execution of a program. In a traditional "stop," execution of the program terminates and you are returned to the operating system level. The program can't do anything more. When an exception is raised, execution stops and sends the interpreter up one level to see if there is some code that will properly handle the error. This means that in using `raise`, you can gracefully handle expected errors without terminating the entire program.

In Section 5.6, we saw how to handle a file opening exception using the `try`...`except` handler. In that handler, you execute the block under the `try`, then execute the `excepts` if an exception is raised. Here's another non-file opening example:

Example 30 (Handling an exception):

Assume we have the function `area` as defined in Example 29 (i.e., with the test for a negative radius). Here is an example of calling the function `area` using `try`...`except` that will gracefully recognize a negative radius and call `area` again with the absolute value of the radius instead as input:

```
rad = -2.5
try:
    a = area(rad)
except ValueError:
    a = area(abs(rad))
```

When the interpreter enters the `try` block, it executes all the statements in the block one by one. If one of the statements returns an exception (as the first `area` call will because `rad` is negative), the interpreter looks for an `except` statement at the calling level (one level up from the first `area` call, which is the level of calling) that recognizes the exception class (in this case `ValueError`). If the interpreter finds such an `except` statement, the interpreter executes the block under that `except`. In this example, that block repeats the `area` call but with the absolute value of `rad` instead of `rad` itself. If the interpreter does not find such an `except` statement, it looks another level up for a statement that will handle the exception; this occurs all the way up to the main level, and if no handler is found there, execution of the entire program stops.

In the examples in this chapter, I used the exception classes `IOError` and `ValueError`. These are examples of built-in exception classes; you can find these and other built-in exception classes listed in a good Python reference (e.g., `TypeError`, `ZeroDivisionError`, etc.) and which you can use to handle the specific type of error you are protecting against.[2] I should note, however, the better and more advanced approach is to define your own exception classes to customize handling, but this topic is beyond the scope of this book.

5.8 Better ways of reading a file

Everything we've seen so far about reading in and writing out a text file works fine. And, just as important, we've seen how powerful Python's string objects are. But, in the real-world, instead of writing our own string parsers to access a file, we make use of pre-written routines to read a data file into an array. In this section, we'll mention the SciPy function `genfromtxt` and the csv module, which work well (respectively) for text files that only (mostly) have numbers and text files made from comma-separated values. (For a discussion of many other pre-written ways of reading in input, see the SciPy Cookbook's "Input and output" page: http://scipy-cookbook.readthedocs.io/items/InputOutput.html.)

Files with only (mostly) numbers: The `genfromtxt` function works well for files that are mostly filled with numbers. How would we use `genfromtxt` to read the data file from Example 28? The code below will produce the array `data`, just as in Example 28:

```
from scipy import genfromtxt
data = genfromtxt('adv44700_sales_only.txt',
                  skip_header=2, skip_footer=1, dtype='l')
```

The `skip_header` keyword input parameter sets how many lines at the beginning of the file to ignore and the `skip_footer` keyword input parameter does the same for the bottom of the file. The `dtype` keyword input parameter tells me to convert the file's values to long integer; if I had left out that keyword input parameter, the function would have converted the values to double-precision floating point type. Details about using `genfromtxt` are found here: http://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html. You might also find the function's reference manual entry to be useful: http://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html.

Files with comma-separated values: We might think, at first glance, that to read in a comma-separated values (CSV) file, all we need to do is apply the string method `split` using the comma (`','`) as the delimiter. And this would work fine, if CSV files contained only numbers. In fact, often times you'll find CSV files that have fields that themselves have newline characters or commas inside them. For instance, consider the following lines that might be the first four lines from a CSV file:

```
Date Sold,Item and Details,Purchase Price,Retail Price
01/01/2015,"Party Mix: Nuts, chips, and salsa",2.14,8.19
02/04/2016,"Game: Rock, scissors, and paper",3.43,9.99
01/22/2013,"Stack-Of-Cars: Chevy\nHonda\nKia\nJeep",2.33,6.87
```

The first line is a header that gives the names for each column. Lines 2–4 are data lines. The quotation marks tell us that the second element in that list is a string `"Party Mix: Nuts, chips, and salsa"`

[2]See http://docs.python.org/library/exceptions.html for a listing of built-in exception classes (accessed August 17, 2012).

and not three separate items, "Party Mix: Nuts", "chips", "and salsa". Unfortunately, a plain-old split(',') call won't figure that out. Note that in a CSV file, there is no whitespace between commas. If there were, the whitespace would be considered part of the values for the column after the comma.

We could, using string variables and Python's programming constructs, write a parser that would properly deal with these and similar special cases. It would, however, be a lot of work. Fortunately, there is a module called csv that handles these special cases for us.

Below is code that will read the contents of a CSV text file *myfile.csv* and put the contents into the list data:

```
import csv
fileobj = open('myfile.csv', 'r')
readerobj = csv.reader(fileobj)
data = []
for row in readerobj:
    data.append(row)
fileobj.close()
```

Note that the list data is not a list of strings but a list of lists. That is, each element in data is a single row in *myfile.csv*, parsed so that each column is a separate element.

Underneath the hood, what's happening in the above code is that the reader function creates the **iterable** readerobj. Iterables are objects that you can loop through using a for loop (as in line 5 in the code above). The syntax of looping through an iterable is the same as looping through a list (which itself is an iterable).

If the CSV text file *myfile.csv* was the data lines earlier on p. 83, printing each row would result in:

```
['Date Sold', 'Item and Details', 'Purchase Price', 'Retail Price']
['01/01/2015', 'Party Mix: Nuts, chips, and salsa', '2.14', '8.19']
['02/04/2016', 'Game: Rock, scissors, and paper', '3.43', '9.99']
['01/22/2013', 'Stack-Of-Cars: Chevy\\nHonda\\nKia\\nJeep', '2.33', '6.87']
```

Note that every element in these lists are strings. If we want to do arithmetic on the prices, we have to convert the prices into floating point values. But, we were able to handle nicely the string elements in the file that had commas and newline characters in them. Details on the csv module are available here: https://docs.python.org/3.5/library/csv.html.[3]

5.9 Summary

In this chapter we saw that Python conceptualizes files as objects, with attributes and methods attached to them. To manipulate and access those files, you use the file object's methods. For the contents of text files, we found string methods to be useful. For most daily work, however, we'll

[3]The discussion in this section also owes credit to http://stackoverflow.com/a/13472940 (accessed October 13, 2016).

want to use special packages designed for reading in data, such as found in NumPy (which we discussed above).

As a postscript: Though we won't talk about it in this course, if you decide you really want to use Python to do data analysis, the package you actually want to use is pandas (http://pandas.pydata.org/). It handles Excel workbook files really well. (The openpyxl module also enables you access Excel workbooks.[4]) Pandas is amazingly powerful, but its learning curve is a little steep. For the purposes of this course, we won't need to use the power it offers, but I wanted to give you a heads-up as to something to learn in the future.

[4]http://openpyxl.readthedocs.io/en/default/ (accessed October 7, 2016).

Part II

Automating and Managing Information Systems

Chapter 6

Managing Files

Chapter Contents

You've done a bunch of data analysis and have generated a host of files. Some are data files, some are plots, and maybe some others are text. Or, maybe the various parts of your company are generating data—point of sale logs, invoices and expenses, customer surveys databases, etc.—and all those other units are sending files your way (perhaps automatically) to manage. How do we handle all these files?

Today's graphically-based operating systems (Windows, Mac OS X) are easy to use because they make sense, visually and kinesthetically. If you want to move a file from one folder to another folder, you click the file's icon with your mouse and drag the icon to the destination folder. Drag means "move." But, this operation no longer is easy when you have thousands of files to move into hundreds of different folders.

In this chapter, we exam some common file management tasks and describe how Python can help us automate these tasks. We won't cover tasks dealing with directories (e.g., getting a list of files in a directory, directory paths, creating and deleting directories, etc.), instead waiting until Chapter 7 to cover those operations; in the present chapter, we'll assume the directories we're interested in already exist. We'll find both in the present chapter and in future chapters that with very few lines of code, Python can tackle the management of five or five million files, and can do so in an operating system independent way. That is, the Python code we write to manage files on a Windows system can also work on a Mac OS X system, with no change. Very cool!

A note of warning: Before you begin using Python's file and directory manipulation tools, I strongly encourage you to create a scratch directory and do all your work in there, until you've gotten the hang of using these functions and commands. Change your **current working directory**

to this scratch directory. If you're running Python from a terminal, just change to this newly created scratch directory and start Python from in there. If you're using Canopy, Figure 7.2 shows where the switch is to change the working directory. This will help decrease the potential of inadvertently messing up your file system.

6.1 Segue: Creating a lot of files

In Chapter 5, we saw how to write out data to a single file. Before we talk about how to manage files, let's first describe how to write out a bunch of files, so we have some files to manage.

Whenever we think about making a lot of something, we should think about doing something over-and-over. After all, doing one task over and over is easier than doing a million different kinds of tasks. When we think of repetition, loops come to mind, so one way of creating a bunch of files is to do so in a loop.

Consider the following made-up example.[1] Let's say you're the owner of an ice cream cart in a local park. You open from 11 am to 7 pm and find your number of customers every hour follows the central part of the first half of a sine function, with the peak number of customers coming between 3–4 pm. That is:

```
import numpy as np
hours = np.array([11, 12, 13, 14, 15, 16, 17, 18, 19], dtype='d')
norm_customers = np.sin( (hours-hours[0]+1)/(hours[-1]-hours[0]+2)*np.pi )
```

where `hours` is the first time in an hour range, based on a 24-hour clock (thus, hour "13" represents the hour from 1–2 pm) and `norm_customers` is the normalized number of customers, which varies from 0 to 1, with 1 being the maximum number of customers in any given day.

Through trial-and-error, you also discover that the number of customers per hour scales with the day's high temperature in the following way:

Temperature	**Scaling Factor**
50°F	6
60°F	10
70°F	25
80°F	41
90°F	55
100°F	64

For the temperatures listed above, you can obtain the number of actual customers for each hour in `hours` by multiplying `norm_customers` by the scaling factor above. Thus, when it is 50°F, the actual number of customers per hour is:

```
>>> norm_customers * 6
array([ 1.85410197,  3.52671151,  4.85410197,  5.7063391 ,  6.        ,
        5.7063391 ,  4.85410197,  3.52671151,  1.85410197])
```

[1]This example is entirely ficticious. Don't try to create a business based upon these numbers.

Give this model of customers at the ice cream cart, what would be the code to write out files listing the hour in the day and the actual customers for each hour (i.e., each file has two columns), for each temperature value we're given above? We'll want to give the files names that are related to the temperature they describe, such as *hrly_customers-50F.txt*. The code to do so is below, using tabs as delimiters:

```
import numpy as np
temps = [50, 60, 70, 80, 90, 100]
scaling_factors = [6, 10, 25, 41, 55, 64]

hours = np.array([11, 12, 13, 14, 15, 16, 17, 18, 19], dtype='d')
norm_customers = np.sin( (hours-hours[0]+1)/(hours[-1]-hours[0]+2)*np.pi )
for i in range(len(temps)):
    fileobj = open("hrly_customers-" + str(temps[i]) + "F.txt", 'w')
    actual_customers = norm_customers * scaling_factors[i]
    for j in range(len(hours)):
        fileobj.write(str(hours[j]) + '\t' + str(actual_customers[j]) + "\n")
    fileobj.close()
```

And instantly we have six files of data! The same idea of using loops to generate multiple files can be used to create any number of files. And, now we have a bunch of files that we can manage in the rest of this chapter ☺.

6.2 Moving and filing lots of files

Now that we've created a lot of files, how do we go about moving them and putting them into sub-directories? Again, let's assume that the sub-directories already exist (we'll create the directories in Chapter 7).

Let's consider the files we created in Section 6.1. Let's say I have two sub-directories, *mild* and *warm*, and I want all files for temperatures less than 65°F to be in *mild* and all files hotter than 65°F to be in *warm*. The following code would accomplish this, given the names of the files is in the list `filenames` (assuming the files are in the same directory as the Python script):

```
1   import shutil
2
3   filenames = [ 'hrly_customers-50F.txt',
4                 'hrly_customers-60F.txt',
5                 'hrly_customers-70F.txt',
6                 'hrly_customers-80F.txt',
7                 'hrly_customers-90F.txt',
8                 'hrly_customers-100F.txt']
9
10  for ifile in filenames:
11      iafter_hyphen = ifile.split('-')[1]
12      itemp = int(iafter_hyphen.split('.')[0][:-1])
13      if itemp <= 65:
14          shutil.move(ifile, 'mild')
15      else:
16          shutil.move(ifile, 'warm')
```

The shutil module (imported in line 1) has a number of functions that simulate **shell** commands. A shell is a text-based interface to an operating system, the software that manages your files, directories, etc.[2] Shells contain commands to enable you to copy, move, delete, etc. your files. In shutil, the `move` function (lines 14 and 16) moves files. The function takes two arguments, the source and its destination (both strings). If the destination is a directory, the source file is moved into the destination directory. If the destination is not a directory, the file is renamed.[3]

Lines 3–8 define the names of the files we will be managing. In this program, we write the names of the files by hand. But what if there were a way to get a list of the files in the current directory without typing them in? We'll talk about such ways in Chapter 7.

In lines 11–12, we use Python's string manipulation methods to extract the temperature value corresponding to the file. The `if` statement in lines 13–16 enables us to move the file to the correct location.

What general lessons can we take from this example? First, file management commands use the string representations of filenames to operate on the files. This means we can use Python's powerful string manipulation tools to work with files. Second, because Python is a full-fledged programming language, we can make use of constructs like loops and `if` statements to help us intelligently manage our files.[4] Finally, we see in this short program that Python has packages that give us the ability to interface with the operating system. This means we aren't stuck with managing files by hand nor with writing a separate program (called a shell script) to do this managing. Instead, we can do this in one environment!

[2]A common shell for Windows is PowerShell. For Linux, bash (or the Bourne Again SHell) is a widely used. Mac OS X, as a Unix-based operating system, also has access to the common shells available to Linux.

[3]Reference for parts of this paragraph: https://docs.python.org/2/library/shutil.html (accessed November 7, 2016).

[4]Shells also have this ability, but the syntax of many shells are much less readable than Python's.

6.3 Renaming lots of files

We noted in Section 6.2 that the `move` function can also be used for renaming files. In the example below, we take the same files as in our Section 6.2 example and rename them so they only have the temperature, then *.txt* (e.g., *50F.txt*). The script assumes the source files are in the same directory as the script:

```
1  import shutil
2
3  filenames = [ 'hrly_customers-50F.txt',
4                'hrly_customers-60F.txt',
5                'hrly_customers-70F.txt',
6                'hrly_customers-80F.txt',
7                'hrly_customers-90F.txt',
8                'hrly_customers-100F.txt']
9
10 for ifile in filenames:
11     iafter_hyphen = ifile.split('-')[1]
12     shutil.move(ifile, iafter_hyphen)
```

In line 11, the `split` method separates the filename based upon the presence of the hyphen. Because the filenames have only one hyphen, the filenames are separated into two parts. These two parts are returned as elements in a list (which has two elements). Everything to the left of the hyphen is the zeroth element of the list and everything to the right of the hyphen is the oneth element of the list. The hyphen itself does not appear in the list. Thus, when we use the list indexing syntax to select the oneth element of the list generated from the split, we get only that portion with the temperature and the ".txt" extension.

6.4 Copying lots of files

To copy a file, we can use shutil's `copy2` function. Like `move`, `copy2` also takes two arguments, the name of the source and then the name of the destination. To take our Section 6.2 example, the following would make copies of all the files in `filenames` with the prefix *copy-* appended to the front of each filename:

```
import shutil

filenames = [ 'hrly_customers-50F.txt',
              'hrly_customers-60F.txt',
              'hrly_customers-70F.txt',
              'hrly_customers-80F.txt',
              'hrly_customers-90F.txt',
              'hrly_customers-100F.txt']

for ifile in filenames:
    shutil.copy2(ifile, 'copy-' + ifile)
```

In the above example, the copies will be in the same directory as the original files.

However, as powerful as `copy2` is, some metadata attached to the file will not transfer. This can include, for instance, ACLs (Access Control Lists) on Mac OS X.[5]

6.5 Deleting lots of files

The os module contains a `remove` function that will do the trick. Note, however, this function does not work on directories. We'll talk about remove directories (empty or not) in Chapter 7. In the example below, we take the same files as in our Section 6.2 example and delete them. Again, the script assumes the files are in the same directory as the script:

```
1  import os
2
3  filenames = [ 'hrly_customers-50F.txt',
4                'hrly_customers-60F.txt',
5                'hrly_customers-70F.txt',
6                'hrly_customers-80F.txt',
7                'hrly_customers-90F.txt',
8                'hrly_customers-100F.txt']
9
10 for ifile in filenames:
11     os.remove(ifile)
```

Note how we import a different module in line 1, and use a different function in the `for` loop.

6.6 Another segue: Testing to see what kind of file something is

I've made reference to Chapter 7 on directories so many times you're probably wondering when will we get there ☺. That chapter's coming right up, but before we get there, let me introduce one final set of functions: those that test what kind of file something is.

These functions are in the os.path submodule. All of these functions take the filename, as a string, as input:[6]

- `isdir`: Returns `True` if the file is a directory.
- `isfile`: Returns `True` if the file is a regular file.
- `islink`: Returns `True` if the file is a symbolic link (i.e., an alias).

For both `isfile` and `isdir`, if the file is a link, the link is followed until its termination. (You can, for instance, make an alias of an alias of an alias, etc. Following a link means following that chain until you reach the original file or directory.) If there is a file or directory at the end of the chain, the function will return `True` or `False` as appropriate.

[5] Reference for parts of this paragraph: https://docs.python.org/2/library/shutil.html (accessed November 7, 2016).
[6] Reference for parts of this paragraph: https://docs.python.org/2/library/os.path.html (accessed November 7, 2016).

Here are the results from using these functions in the Python interpreter on the *hrly_customers-50F.txt* file and the *mild* directory, as from our Section 6.2 example. Remember, the script assumes the files are in the same directory as the script:

```
>>> import os.path
>>> os.path.isfile('hrly_customers-50F.txt')
True
>>> os.path.isfile('mild')
False
>>> os.path.isdir('mild')
True
>>> os.path.islink('mild')
False
>>> os.path.islink('hrly_customers-50F.txt')
False
```

Both the file and directory are regular items, not links (i.e., not aliases), and so the `islink` call returns `False`.

Why do we care about testing what a file is? Now, as we go into Chapter 7, we can test to see whether a file is a directory or not and choose the right file/directory handling function to use.

6.7 Summary

We've seen that functions in the shutil, os, and os.path modules can be used to manage files, and that the use of looping gives us a powerful way of managing many files in very few lines of code. In addition, the commands we've seen are the same regardless of the operating system we're on. File management using Python, as a result, has the potential of enabling us to write a single program that can manage our files whether the computer we're on is running Windows, Mac OS X, Linux, or any one of a number of other operating systems. While a pure and complete "write once, run anywhere" ability is hard to come by, the file management features in Python help move us in that direction.

(For more details on these modules, please see the Python documentation: https://docs.python.org/2/library/shutil.html, https://docs.python.org/2/library/os.html, and https://docs.python.org/2/library/os.path.html.)

One final note: As powerful as automation is, it's not always worth doing. Here are two xkcd comics to help us think more clearly about when to automate and when not to ☺: https://xkcd.com/1205/ and https://xkcd.com/1319/.

Chapter 7

Managing Collections of Files: Directories

Chapter Contents

In Chapter 6, we took a look at some of the functions Python has that we can use to move, rename, copy, and delete files. These functions are particularly useful because we can embedded calls to them in a loop and automatically operate on many files.

No one today (or at least no one should ☺), however, stores all of their files on their desktop. Instead, we put collections of files into directories or folders and place groups of folders into other folders, and so on. Using directories, we're able to group related files with each other and move entire collections of files around at one time.

But, what if you have a large collection of directories? We then have the same problem as in Chapter 6, except in this case instead of needing to manage many files we need to manage many directories (which may in turn contain many files). Can Python help us with this task?

In this chapter, we look at some of the tools Python provides to help us manage directories and their contents. There are two classes of tools we'll look at: those used in creating, naming, moving, and copying directories and those used in accessing the contents of the directory. We start, however, with something of a prelude: how can we use Python to help us specify the path to a directory?

Before we do, I want to reiterate what I said at the beginning of Chapter 6: Before you begin using Python's file and directory manipulation tools, I strongly encourage you to create a scratch directory and do all your work in there, until you've gotten the hang of using these functions and commands. Change your current working directory to this scratch directory. This will help decrease the potential of inadvertently messing up your file system.

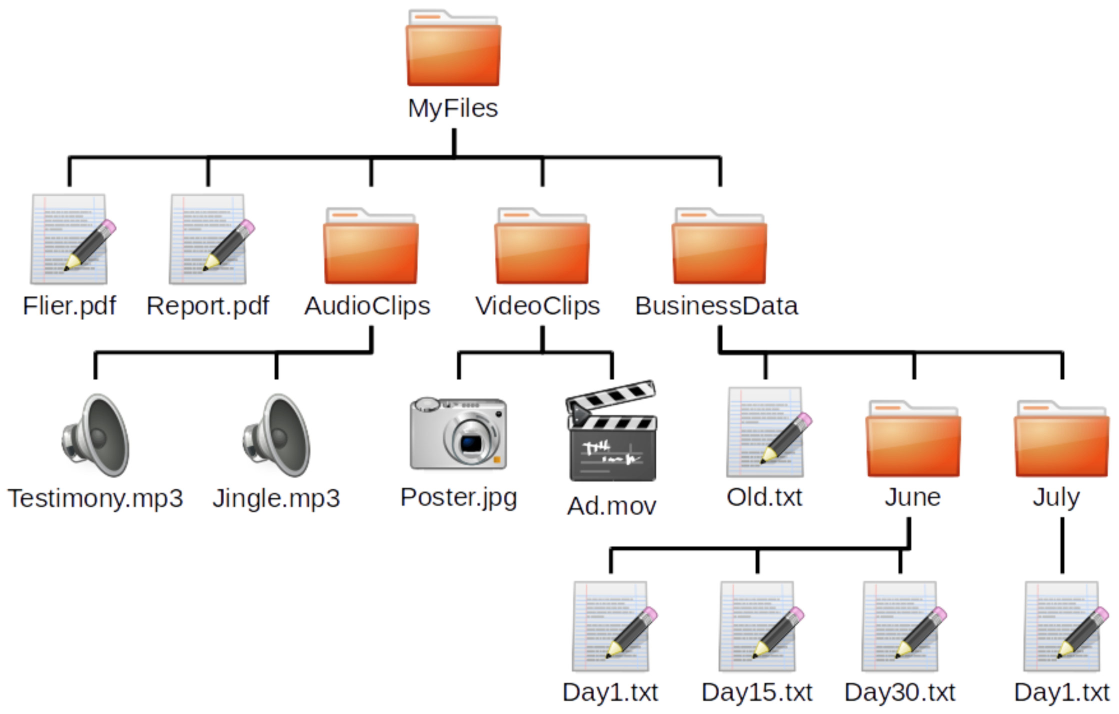


Figure 7.1: A directory tree.

7.1 Prelude: Specifying directory paths

The **path** to a directory is the list or sequence of directories you need to go through to reach file or directory at the end of that list/sequence.[1] Figure 7.1 shows a tree of files starting from the directory *MyFiles*. To reach the file *Day15.txt*, the path would start with *MyFiles*, go into *BusinessData*, and then into *June*, ending with the file of interest, *Day15.txt*. On a Linux system, this path would be written as:

MyFiles/BusinessData/June/Day15.txt

while on a Windows system, this path would be written as:

MyFiles\BusinessData\June\Day15.txt

Immediately, we see that depending on the operating system we use, the character that separates different directories in the path may differ. On Windows, the character is a backslash while in Linux the character is a forward slash. (Note that a backslash, when represented in a string, is a "backslash backslash", i.e., "\\". When the string is printed out, only a single backslash occurs:

```
>>> a = "\\"
>>> print(a)
\
```

[1]Here "list" does not mean a Python list but a sequence or listing of directories.

In the next paragraph we'll see how the os.path module helps us to avoid having to think about this.)

Python's os.path module (the path submodule of the os module) provides tools to enable us to construct and manipulate paths that account for the difference between the directory separation characters used in different operating systems. The `join` function of os.path takes a set of string arguments and joins them together into a single string with the correct character separating directories for the operating system being used. For instance, for our *Day15.txt* path example above, `join` would give:

```
>>> import os.path
>>> os.path.join("MyFiles", "BusinessData", "June", "Day15.txt")
'MyFiles/BusinessData/June/Day15.txt'
```

The above code was executed on a Linux system, so the path separation character used is a forward slash.

In addition to having functions to create a path, the os.path module also has functions to manipulate paths (quite a few, in fact). Some functions include: `abspath`, which gives the **absolute path** (a.k.a. full path) for a given path:

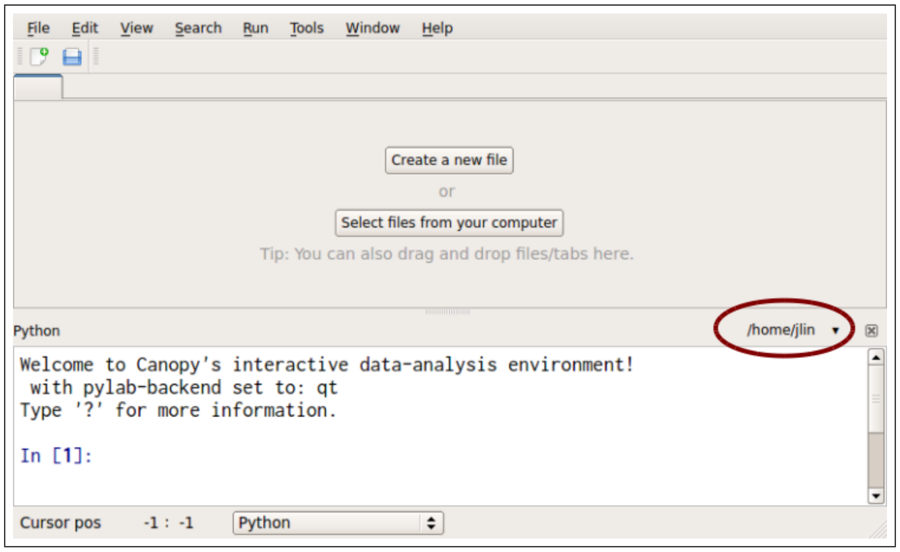
```
>>> import os.path
>>> mypath = os.path.join("MyFiles", "BusinessData", "June", "Day15.txt")
>>> os.path.abspath(mypath)
'/home/jlin/MyFiles/BusinessData/June/Day15.txt'
```

In this example, the path given by `mypath` is a **relative path**, that is, a path specification relative to the present location (called the current working directory). By default, when you start Python from the command line, the current working directory is the location you were in when you started Python. In this example, I started Python while I was in the */home/jlin* directory, so the full path to `mypath` is */home/jlin/MyFiles/BusinessData/June/Day15.txt*, which is what `abspath` gives. In Canopy, you can set the working directory via the Python interpreter window. Figure 7.2 shows the Canopy Python interpreter of Figure 1.2, except with the working directory status bar and edit menu switch circled by a red circle. The status bar shows what the current working directory in Canopy is set to and the edit menu (accessed by a right-click) gives you options for changing that working directory. You can, however, get and change the current working directory from within Python, while a Python program is running; we discuss how in Section 7.5.

The function `basename` returns the file or directory that is at the end of the path. For the above example:

```
>>> import os.path
>>> mypath = os.path.join("MyFiles", "BusinessData", "June", "Day15.txt")
>>> os.path.basename(mypath)
'Day15.txt'
```

If you are looking for a straightforward way to figure out what kind of file is specified by path, via the file's extension, you can use the `splitext` function to split off the file extension from the rest of the path. For example:



```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [1]:
```

Figure 7.2: The Canopy Python interpreter, with the working directory status bar and edit menu switch shown. Right-click the switch to get menu options that include changing the working directory.

```
>>> import os.path
>>> mypath = os.path.join("MyFiles", "BusinessData", "June", "Day15.txt")
>>> os.path.splitext(mypath)
('MyFiles/BusinessData/June/Day15', '.txt')
```

`splitext` returns a two-element tuple with the extension in the second element and the rest of the path in the first. If the path does not have a file extension, the second element of the tuple is an empty string. You can run tests on what file extension is returned and have your program respond accordingly.

The Python documentation on os.path is a must read if you want to see all the different tools you have at your disposal for handling paths: https://docs.python.org/2/library/os.path.html.

7.2 Creating and removing empty directories

To create a new, empty directory, you can use the `mkdir` command from the os module. In this example, I create the *August* directory in the *BusinessData* directory in the directory tree of Figure 7.1 (assuming *MyFiles* is relative to the current working directory):[2]

```
>>> import os
>>> import os.path
>>> mypath = os.path.join("MyFiles", "BusinessData", "August")
>>> os.mkdir(mypath)
```

[2]Reference for parts of this paragraph: https://docs.python.org/2/library/os.html (accessed November 9, 2016).

The `mkdir` command, however, assumes that all intermediate directories to the directory you are creating already exist. In the above example, if the *MyFiles* and *BusinessData* directories do not already exist, the call to `mkdir` will fail and return an error.

If you want to create all the intermediate directories in addition to the final (or **leaf directory**), you should use `makedirs`:[3]

```
>>> import os
>>> import os.path
>>> mypath = os.path.join("MyFiles", "Car", "Manuals")
>>> os.makedirs(mypath)
```

In this example above, the `makedirs` command creates not only the leaf directory, *Manuals*, but also the intermediate directory *Car*, which does not exist in the Figure 7.1 directory tree. `makedirs` does all this in one fell swoop.

Notice for both the `mkdir` and `makedirs` calls above, no message is printed in the interpreter. The commands create the directories on you file system and that's that. No news means the command executed "successfully." Of course, you have to make sure that you typed everything in correctly, in order for "successful" to be the same as "desired outcome" ☺.

Of course, for both `mkdir` and `makedirs`, we're creating empty directories that have nothing inside them (except directories that are part of the path). We'll address the case of copying and deleting directory trees with files in them in Section 7.4.

Given `makedirs`, why would you use ever want to use `mkdir`? Admittedly, `makedirs` is powerful, but with such power comes the potential to mess things up. Unless I know that I want to create all the intermediate directories in a path, I'd use `mkdir` to limit any inadvertent mistakes.

7.3 Renaming and moving directories

In Section 6.3, we saw how the shutil module's `move` function could be used to move and rename files. It also works for renaming directories. In this example, I rename the *June* directories in *BusinessData* to *PastJune*:[4]

```
>>> import os.path
>>> import shutil
>>> june_path = os.path.join("MyFiles", "BusinessData", "June")
>>> new_june_path = os.path.join("MyFiles", "BusinessData", "PastJune")
>>> shutil.move(june_path, new_june_path)
```

If I'm in the *BusinessData* directory, that is, if my current working directory is the *BusinessData* directory in Figure 7.1, the rename is even easier (with the imports removed):

```
>>> shutil.move("June", "PastJune")
```

[3]Reference for parts of this paragraph: https://docs.python.org/2/library/os.html (accessed November 9, 2016).
[4]Reference for parts of this paragraph: https://docs.python.org/2/library/shutil.html (accessed November 9, 2016).

There's no need to bother with `os.path.join` since I'm only referring to paths made up of a single directory name and level.

In addition to renaming directories, we can also use shutil's `move` to move directories (and everything in that directory) into another directory. Thus, if we wanted to create a directory called *Media* inside *MyFiles* (in the Figure 7.1 tree) and move *AudioClips* and *VideoClips* into *Media*, we'd execute the following code (I'm assuming our current working directory is *MyFiles*):

```
>>> import os
>>> import shutil
>>> os.mkdir("Media")
>>> shutil.move("AudioClips", "Media")
>>> shutil.move("VideoClips", "Media")
```

7.4 Copying and deleting directories

Copying and deleting directories is a more complicated task than creating empty directories or moving directories around because copying and deleting need to operate on the entire directory tree in question. That is, if we wanted to copy *BusinessData* in the Figure 7.1 tree, we need to copy *everything* contained in *BusinessData*, going down into the *June* and *July* directories too. This is called a **recursive** copy. (We'll talk more about **recursion** in Chapter 8.)

To make the copy, we use the shutil function `copytree`. Like `move`, it takes two arguments, strings that specify the source and the destination of the copy. The following will make a copy of *BusinessData* called *BusinessDataCopy*. Both the original and copy will be in the same directory (the current working directory):[5]

```
>>> import shutil
>>> shutil.copytree("BusinessData", "BusinessDataCopy")
```

Note that `copytree` does not copy over all the metadata. (If you care about copying *all* the metadata, you'll probably have to write a shell script.)

To remove a directory and all sub-directories and files, use the shutil function `rmtree`. The following will remove *BusinessDataCopy* and everything in it from the current working directory:[6]

```
>>> import shutil
>>> shutil.rmtree("BusinessDataCopy")
```

7.5 Returning and changing the working directory

Earlier in Section 7.1, we talked about relative paths and how they are relative to where you're currently at in the directory tree, i.e., the current working directory. From that discussion, it may have sounded like the current working directory is set in stone for any given Python session. In actuality, you can change the current working directory at any time during a Python session.

Before you change the current working directory, however, you might be interested in knowing what it's currently set to. The `getcwd` function in the os module does this:

[5]Reference for parts of this paragraph: https://docs.python.org/2/library/shutil.html (accessed November 9, 2016).
[6]Reference for parts of this paragraph: https://docs.python.org/2/library/shutil.html (accessed November 9, 2016).

```
>>> import os
>>> os.getcwd()
'/home/jlin'
```

Remember that `getcwd` is a function so you have to put the empty set of parenthesis in order to call the function. In the example above, I had started Python in a terminal window while I was in my home directory, */home/jlin*, so this is the value of my current working directory.

To change my current working directory, I use the `chdir` function in the os module. The following will change my current working directory to the `temp` directory in my home directory:

```
>>> import os
>>> os.chdir("/home/jlin/temp")
```

In the above example, I specified the full or absolute path for my own computer, which runs Linux. Since I had started my Python session from my home directory, and I'm trying to change my current working directory to a sub-directory of my home directory, the following would have also worked:

```
>>> import os
>>> os.chdir("temp")
```

So, I can specify a directory in `chdir` using a relative path instead of an absolute path.

7.6 Listing the contents of a directory

Thus far, we've talked about how to create, move, and delete directories, as well as changing our current working directory. Once we're at or know how to get to a directory we're interested in, how do we list the contents of that directory? The os module's `listdir` function will do the trick. If I'm in the *MyFiles* directory of the directory tree in Figure 7.1, I'll get the following:[7]

```
>>> import os
>>> os.listdir('.')
['AudioClips', 'Report.pdf', 'Flier.pdf', 'VideoClips', 'BusinessData']
>>> os.listdir(os.getcwd())
['AudioClips', 'Report.pdf', 'Flier.pdf', 'VideoClips', 'BusinessData']
```

In the first example of `listdir`, I pass in the current directory symbol in Linux (a period) and the function returns me the list of the contents of the directory I am currently in (which is *MyFiles*). In the second example, I pass in the return value of a call to the `getcwd` function, which returns the full path of the directory I am currently in.

Sometimes, I'm interested in listing only a subset of the contents of a directory. The glob module has a function called `glob` that will do basic pattern matching when doing a directory listing. If I'm in the *MyFiles* directory of the directory tree in Figure 7.1, I'll get the following:[8]

[7]Reference for parts of this paragraph: https://docs.python.org/2/library/os.html (accessed November 10, 2016).
[8]Reference for parts of this paragraph: https://docs.python.org/2/library/glob.html (accessed November 10, 2016).

```
1  >>> import glob
2  >>> glob.glob('*')
3  ['AudioClips', 'Report.pdf', 'Flier.pdf', 'VideoClips', 'BusinessData']
4  >>> glob.glob('*.pdf')
5  ['Report.pdf', 'Flier.pdf']
```

In both calls to `glob`, the asterisk is a wildcard character meaning "any character or set of characters." Thus, in the line 2 `glob` call, the function looks for all contents of *MyFiles*, i.e., items of any name. In the line 4 call, the function looks for all contents whose name ends in *.pdf*. More complex pattern matching is possible; see the glob module documentation for details (https://docs.python.org/2/library/glob.html).

7.7 Python Sidebar: Dictionaries

So far, when we have needed to store more than one piece of information, we've put them into lists/tuples or arrays. Once in those list-like structures, we pick out one or more values by specifying an index or range of indices (via slicing). But when you think about, an index doesn't tell you very much about what an element holds. All it says is that this is the position of the element.

It turns out that many times, it's more useful for us to reference values not by a positional index but by something more meaningful. For instance, say you had a variable `files` that is the list of the files in the *June* subdirectory in the Figure 7.1 directory tree:

```
files = ['Day1.txt', 'Day15.txt', 'Day30.txt']
```

How would you store the number of lines in these files? You could construct a list (let's call it `num_lines` and store the values in that list, and arrange the items so values in a given element correspond to the element in `files` at the same location. That is, if:

```
num_lines = [12, 35, 92]
```

then this means *Day1.txt* has 12 lines, *Day15.txt* has 35 lines, and *Day30.txt* has 92 lines. To print out the filenames and the number of lines corresponding to them, we could write the following:

```
files = ['Day1.txt', 'Day15.txt', 'Day30.txt']
num_lines = [12, 35, 92]
for i in range(len(files)):
    print( files[i] + ": " + str(num_lines[i]) )
```

But isn't this a little cumbersome? If we make any changes to one variable we have to be *very* careful to synchronize the change with the other variable. If `files` changes but `num_lines` does not, there's no way for us to connect the two together again. Wouldn't it be nice if there was a way to directly associate the filename with the number of lines value? The dictionary can solve this problem. We'll first look at what are dictionaries, how to create them, and what kinds of tools they offer us. Then we'll come back to our `files` and `num_lines` example above and store the information in a dictionary.

Like lists and tuples, dictionaries are also collections of elements, but dictionaries, instead of being ordered, are *unordered* lists whose elements are referenced by *keys*, not by position. Keys can be anything that can be uniquely named and sorted. In practice, keys are usually integers or strings. Values can be anything. (And when I say "anything," I mean *anything*, just like lists and tuples.)

Curly braces ("{}") delimit a dictionary. The elements of a dictionary are "key:value" pairs, separated by a colon. Dictionary elements are referenced like lists, except the key is given in place of the element address. The example below will make this all clearer:

Example 31 (A dictionary):

Type the following in the Python interpreter:

```
a = {'a':2, 'b':3.2, 'c':[-1.2, 'there', 5.5]}
```

For the dictionary `a`:

- What does `a['b']` return?
- What does `a['c'][1]` return?

Solution and discussion: `a['b']` returns the floating point number `3.2`. `a['c']` returns the list `[-1.2, 'there', 5.5]`, so `a['c'][1]` returns the oneth element of that list, the string `'there'`.

Like lists, dictionaries come with methods that enable you to find out all the keys in the dictionary, find out all the values in the dictionary, etc. Here are some useful dictionary methods:

Example 32 (A few dictionary methods):

Assume we have the dictionary from Example 31 already defined in the Python interpreter:

```
a = {'a':2, 'b':3.2, 'c':[-1.2, 'there', 5.5]}
```

If you typed the following into the Python interpreter, what would you get for each line?:

- `d = a.keys()`
- `d = a.values()`
- `a.has\_key('c')`

Solution and discussion: The first line executes the command `keys`, which returns a list of all the keys of `a`, and sets that list to the variable `d`. The second command does this same thing as the first command, except `d` is a list of the values in the dictionary `a`. The third command tests if dictionary `a` has an element with the key `'c'`, returning `True` if true and `False` if not. For the dictionary `a`, the first line returns the list `['a', 'c', 'b']` and sets that to the variable `d` while the second line returns `True`.

Note that the `keys` and `values` methods do *not* return a sorted list of items. Because dictionaries are *unordered* collections, you *must not* assume the key:value pairs are stored in the dictionary in any particular order. If you want to access the dictionary values in a specific order, you should first order the dictionary's keys (or, in some cases, values) in the desired order using a sorting function like `sorted`.

Once a dictionary is created, you can add key:value pairs to the dictionary by assignment. Thus, if we wanted to add the value `'goodbye'` with the key `'d'` to the dictionary in Example 31 (reproduced below as a reminder), we would type:

```
>>> a = {'a':2, 'b':3.2, 'c':[-1.2, 'there', 5.5]}
>>> a['d'] = 'goodbye'
>>> a
{'a': 2, 'c': [-1.2, 'there', 5.5], 'b': 3.2, 'd': 'goodbye'}
```

To replace the value of an existing dictionary entry, we use the same assignment syntax. To delete an entry, we use the `del` command. On the above dictionary:

```
>>> del(a['c'])
>>> a
{'a': 2, 'b': 3.2, 'd': 'goodbye'}
```

Let's return to the file system example we were looking at at the beginning of this section. How might we use dictionaries to connect the number of lines in each file with the file's name? By using the filename as the key and the number of lines as the value. That is, we can define this dictionary `num_lines_dict`:

```
num_lines_dict = {'Day1.txt':12, 'Day15.txt':35, 'Day30.txt':92}
```

We obtain the number of lines values by referencing a filename, which is a key, such as:

```
>>> num_lines_dict = {'Day1.txt':12, 'Day15.txt':35, 'Day30.txt':92}
>>> num_lines_dict['Day15.txt']
35
```

and we can add new elements by assignment. For instance, let's say a new file *Day10.txt* was created in this directory and that the file was 103 lines long. We add the entry to the dictionary by typing in:

```
num_lines_dict['Day10.txt'] = 103
```

If we want to loop through all the files and print out the number of lines for each file, the following code will do the trick:

```
for ikey in num_lines_dict.keys():
    print( ikey + ": " + str(num_lines_dict[ikey]) )
```

Note, however, the files will *not* be printed in any (humanly discernable) order. As dictionaries are unordered collections, we must think of the keys method as returning the dictionary's keys in any way it wants to. If we want to go through the keys in a particular order, we have to sort the keys first:

```
for ikey in sorted(num_lines_dict.keys()):
    print( ikey + ": " + str(num_lines_dict[ikey]) )
```

The above code uses the built-in sorted function to sort the keys using sorted's default ordering algorithm.

What does this use of dictionaries buy us? Some thoughts regarding the above example:

- We can now refer to elements associated with a filename by the filename itself. There's no ambiguity, for instance, as to what the "threeth" element refers to.
- Instead of dealing with two lists and making sure we keep them synchronized with each other, all the information is now stored in a single variable.
- We can add and remove items from this collection of data without having to worry about messing up the order of the collection because the data is stored unordered and referenced without respect to order.

One final note: If we want to store more data about a file than a single number, we can make the value part of the dictionary's key:value pairs to be a collection of some sort, say a list or another dictionary. Dictionaries are very flexible! If the structure of the data we want to store is even more complex, it's probably a good idea to starting thinking about defining our own classes of objects to store the data. We'll talk some about this topic in Chapter 9.

7.8 Summary

Through the os, shutil, and glob modules, Python gives us many of the file and directory management tools that a shell gives us. However, because Python is a full-fledged programming language with many scientific, mathematical, and business packages, we can write much more complex programs for managing and interacting with our files and directories than are possible in a shell language. The dictionary data structure is an example of a tool that makes storing information related to files (whose names are straightforwardly represented by strings) a piece of cake (well, maybe two pieces ☺). One common application of these tools is searching for files on a computer, and it's to that topic we turn to in Chapter 8.

Chapter 8

Managing Collections of Files: Searching

Chapter Contents

We've talked about creating, copying, and moving files. We have one more foundational MIS file and directory management activity to discuss: searching. File systems can contain hundreds of thousands to millions of files and directories. How do we find the file or files we want? It could be that set of sales data files or the brochures from that ad campaign ten years ago.

In this chapter, we describe some of the tools Python provides to help us find files on our computer, automatically. We'll start out by reviewing searching tools we've already seen in Python and how they can be used to search for files (among other things). Next, we'll look at Python tools specifically designed for searching directory trees. Finally, we'll examine another way of searching directory trees using the concept of recursion.

8.1 Review of Python routines for searching

While we haven't specifically considered how do you find something in a collection of items in Python, it turns out we've already seen quite a few ways of doing just that. Here's a quick review.

Testing in a loop: You can loop through a list of items and use an `if` statement to find what you're looking for. Here we take some numbers and store the index in the list where the number six is found in the variable `loc`:

```
data = [3, 6, 1, -8, 9]
for i in range(len(data)):
    if data[i] == 6:
        loc = i
```

Note that in the above code, `loc` will be set to the index of the last occurrence of the number six.

The `index` method of a list: As we saw in Example 4, the `index` method of a list will return the index where the first occurrence of the search target is found. Below, we search the `data` list for the index where the first occurrence of the number six is located and store that index in `loc`:

```
data = [3, 6, 1, -8, 9]
loc = data.index(6)
```

The `count` method of a list: As we also saw in Example 4, the `count` method of a list will count the number of occurrences in the list of a search target. Below, we search the `data` list to find how many times the number six occurs and store that number in `count_six`:

```
data = [3, 6, 1, -8, 9]
count_six = data.count(6)
```

Array syntax and testing for equality: Array syntax automatically applies logical operators like equality element-wise across an array. Below we look in `data` for where it equals the number six:

```
import numpy as np
data = np.array([3, 6, 1, -8, 9])
mask = data == 6
```

The variable `mask` is a boolean array showing which elements of `data` equal the number six and which do not:

```
>>> mask
array([False,  True, False, False, False], dtype=bool)
```

This only works with NumPy arrays and not lists.

The `where` function on arrays: We saw this searching method in Section 4.6.2. Below we search `data` for the number six and return an array where elements equal to six are set to its index and -1 otherwise:

```
import numpy as np
data = np.array([3, 6, 1, -8, 9])
mask = np.where(data == 6, np.arange(np.size(data)), -1)
```

The result of the `where` call is stored in `mask` which is:

```
>>> mask
array([-1,  1, -1, -1, -1])
```

The `isclose` function on arrays: We saw how to use `allclose` and `isclose` in Section 4.7 to do "equality" tests on floating point numbers. Below we look in `data` for where it is close to the number six:

```
import numpy as np
data = np.array([3., 6., 1., -8., 9.])
mask = np.isclose(data, 6.0)
```

and, as earlier, the variable `mask` is a boolean array showing which elements of `data` are close to the number six and which are not:

```
>>> mask
array([False,  True, False, False, False], dtype=bool)
```

Membership testing: Starting on p. 39, we discussed how the `in` operator can be used for membership testing in lists and strings. This doesn't tell us where a search target is in the list or string but it does tell us if the target is present in the list or string. Below we test whether the number six is in the list `data`:

```
>>> data = [3, 6, 1, -8, 9]
>>> 6 in data
True
```

If a dictionary has a specific key: The `has_key` method, as we saw in Example 32, let's us see if the dictionary has a given key. Below we construct a dictionary `data` and ask whether the dictionary has a key named `'six'`:

```
>>> data = {'three':3, 'six':6, 'one':1, 'neg_eight':-8, 'nine':9}
>>> data.has_key('six')
True
```

You can also use the dictionary method `values` to obtain all the values in `data` as a list and use the list `index` method to get the location of the search target. The values are not in any human-understood order, but the index will work to get the value of interest. Here, we look for the number six:

```
>>> data = {'three':3, 'six':6, 'one':1, 'neg_eight':-8, 'nine':9}
>>> data_values = data.values()
>>> loc = data_values.index(6)
>>> data_values[loc]
6
```

The `glob` function for wildcard searching in a directory: In Section 7.6, we saw that the glob module function `glob` can search a directory for filenames matching a given pattern. The following searches the current working directory for all files that have the number six somewhere in their name and returns the list of those files as `files6`:

```
import glob
files6 = glob.glob("*6*")
```

8.2 Searching in a directory tree

In Section 7.6, we saw how to get a list of all the contents of a directory. Using some of the methods we've just reviewed, we can search through that listing to find the file we are interested in.

But, often times, what we're interested in is not whether a file is in a single given directory but all sub-directories of that given directory, and their sub-directories, and so on until there are no more sub-directories to search. That is, we are interested in finding a file in a directory tree (an example is shown in Figure 7.1). At the most extreme, if our current directory is the top-most directory of our **filesystem**, a directory tree search means searching through every file and directory on our computer. The os module's `listdir` function won't do that kind of search for us.

Instead, if you want to descend into a directory tree to search for a file, you can use the os module's `walk` function.[1] The `walk` function, however, is a little different than most functions. Most functions take an input and produce an output. The output is usually something you can save to a variable, such as a number, string, list, or, array. What `walk` produces, however, is something called a generator. For our purposes, what a generator exactly is isn't that important.[2] Instead, we're going to focus on how to use a generator. In short, a generator is an object whose purpose is to give you something to iterate through and return you something as you do the iteration. Thus, generators are often used in loops.

Let's look at an example. Let's pretend the current working directory holds the folder *MyFiles*, which is the topmost directory of the Figure 7.1 directory tree. The following code:

```
import os
for dirpath, dirnames, filenames in os.walk("MyFiles"):
    print( str(dirpath) + ":\n    " + str(dirnames) + \
           "\n    " + str(filenames) )
```

will produce this output:

[1]Reference for parts of this section: https://docs.python.org/2/library/os.html. See also http://stackoverflow.com/a/1724723. (Both accessed November 11, 2016.)

[2]See https://wiki.python.org/moin/Generators if you're interested in learning about generators.

```
1  MyFiles:
2      ['AudioClips', 'VideoClips', 'BusinessData']
3      ['Report.pdf', 'Flier.pdf']
4  MyFiles/AudioClips:
5      []
6      ['Jingle.mp3', 'Testimony.mp3']
7  MyFiles/VideoClips:
8      []
9      ['Ad.mov', 'Poster.jpg']
10 MyFiles/BusinessData:
11     ['July', 'June']
12     ['Old.txt']
13 MyFiles/BusinessData/July:
14     []
15     ['Day1.txt']
16 MyFiles/BusinessData/June:
17     []
18     ['Day15.txt', 'Day1.txt', 'Day30.txt']
```

Let's interpret this. Each iteration of the `for` loop, the variables `dirpath`, `dirnames`, and `filenames` are filled. On the first iteration of the loop, `dirpath` is set to `'MyFiles'`, `dirnames` is set to the list `['AudioClips', 'VideoClips', 'BusinessData']`, and `filenames` is set to the list `['Report.pdf', 'Flier.pdf']`. (These contents are shown in lines 1–3 above.) That is to say, on the first iteration, `dirpath` is set to the topmost directory in the tree (*MyFiles*) and `dirnames` and `filenames` are set to the names of the sub-directories and files that are in *MyFiles*.

On the second iteration, `dirpath` is set to *MyFiles/AudioClips*, that is, the *AudioClips* sub-directory in *MyFiles*. The `dirnames` variable is set to an empty list, and the `filenames` variable is set to the list `['Jingle.mp3', 'Testimony.mp3']`. (These contents are shown in lines 4–6 above.) That is, because there are no sub-directories in *AudioClips*, the `dirnames` list has nothing in it and only the files in *AudioClips* show up (in the list `filenames`).

The third iteration sets `dirpath`, `dirnames`, `filenames` to what's shown in lines 7–9 above, and so on until all sub-directories in the *MyFiles* directory tree are visited. As a side note, from the output, we see that when we reach a "leaf" directory, i.e., a directory that only contains files, the `dirnames` list will be empty (and thus have a length of zero). If we're interested in looking only at leaf directories, we can use `len(dirnames)` as a test to see whether we're in a leaf directory.

Now, let's use `walk` to search for a file. Let's say we want to find the location of all occurrences of a file named *Day1.txt*. From Figure 7.1, we know there are two files in this tree by that name. The following code will store the directory paths to all occurrences of *Day1.txt*:

```
import os
file_occurrence_locations = []
for dirpath, dirnames, filenames in os.walk("MyFiles"):
    if 'Day1.txt' in filenames:
        file_occurrence_locations.append(dirpath)
```

and the `file_occurrence_locations` list will have the following values:

```
>>> file_occurrence_locations
['MyFiles/BusinessData/July', 'MyFiles/BusinessData/June']
```

which are the paths of the sub-directories where there are files named *Day1.txt*.

One final note about finding files: It is an unfortunate truth that different operating systems not only have different directory separation characters but also different rules about case-sensitivity. On some operating systems (like Linux), filenames can be made from upper- or lower-case characters. On other operating systems (like Mac OS X), there is no real distinguishing between a file named *Foo.txt* and one named *foo.txt*.[3] This issue with case makes it possible that a plain string equality or membership test will fail, if you're working across operating systems. Python provides the fnmatch module to enable pattern matching for filenames that takes into account the case handling features of the operating system one is working on. See the fnmatch documentation for details (https://docs.python.org/2/library/fnmatch.html).

8.3 Searching using recursion

Using the `walk` generator in a loop is a great way of searching through a directory tree. It's concise, flexible, and robust. By all means, use it when you want to access the contents of a directory tree!

However, while we could end the topic of walking through a directory tree right here and go on to the next chapter without any loss in our ability to handle directory trees, it turns out the directory tree searching problem very naturally lends itself to being solved by a programming concept called recursion. While `walk` means you don't have to use recursion to traverse a directory tree, it's worth using the tree traversal problem to learn about how recursion works, because recursion is a really useful concept that can help you write more concisely what would otherwise be really complex functions or programs. In my own work, I seldom use recursion—in fact, I can only think of one time—but in that one time, there really was no other way to write the program in a clear and simple way. In the rest of this section, we'll discuss what is recursion and use recursion to implement a simple file search in a directory tree.

8.3.1 What is recursion?

So, what is this mysterious programming technique called recursion? In a nutshell, recursion, or more specifically recursive functions, are functions whose definitions include calls to itself. That is to say, if I have a function `calculate`, within the definition of `calculate` there is at least one call of `calculate`.

I don't know about you, but that seems weird to me. How can you use a function while you're defining it? It sounds like an infinite regress: use a function you're defining using a function you're defining using a function you're defining, and so on without end. At least, that's how I felt the first time (and second, and third, and more) I learned about it.

[3]The real situation is a little more complex. The issue is with the filesystem format you choose and not with the Mac version of Unix itself. In addition, case-insensitive here does not mean it is not case-preserving. See http://apple.stackexchange.com/a/22304 for a nice discussion of the issue (accessed November 11, 2016).

A mathematical example might be helpful to describe both what recursion is and why it doesn't have to lead to infinite regress. Let's say you want to write a function that takes a list of numbers and adds the numbers up. Using a `for` loop, the function `add_up` would be:

```
def add_up(input_list):
    total = 0.0
    for i in input_list:
        total = total + i
    return total
```

When we go through the logic of this code in our head, we probably hear something like, "take one number, add it to `total`, take another number, add it to `total`, and so on until we're out of numbers."

But, we could think of the process of adding up these numbers in another way: "the sum of a list of numbers is the sum of the list of numbers *not including the last number* plus the last number." But how do we get the "sum of the list of numbers *not including the last number*?" Using the same logic, that sum is also, "the sum of the list of numbers *not including the last number* plus the last number," but the "list of numbers" isn't the original list of numbers but the original list without the last item on the list. The logic would continue until "the sum of the list of numbers *not including the last number*" is zero, because there are no numbers left in such a list.

In Python, this would look like:

```
1  def add_up(input_list):
2      if len(input_list) == 0:
3          return 0.0
4      else:
5          return add_up(input_list[:-1]) + input_list[-1]
```

Some things to point out in this code. First, we see that our recursive call, i.e., our call to the `add_up` function inside the `add_up` definition, occurs in line 5. It's important to note, however, that while we are calling `add_up` in line 5, the input parameter is *not* the same as the original, full list of numbers. (If it were, we would have to get infinite regress.) Instead, the input argument we're passing into the line 5 call to `add_up` is the current `input_list` without the last element. This is how we implement the "not including the last number" phrase in our above word description of our logic.

Second, our worries about infinite regress are solved by lines 2–3. The `if` statement checks to see whether the list of numbers `input_list` actually has any numbers in it and if not, the function returns zero. This makes sense because the sum of nothing is zero. But from the standpoint of infinite regress, this means that there comes a point when we no longer will make calls to `add_up`. The situation tested for in lines 2–3 and called the **stopping case** or **base case**. As the name suggests, this condition terminates recursive calls to the function preventing infinite regress. Every recursive function has to have at least one stopping case. The lack of one (or enough) stopping cases will result in a recursive function eventually failing and causing a "stack overflow." We're not going to go into what a stack overflow is beyond saying if it happens, your program will terminate. So you'll know you made a mistake ☺.

In this example, the recursive function doesn't seem more concise than the one using a loop. It's not. For simple problems, this is typical: the looping solution is more straightforward and shorter than the recursive version. For more complex problems, however, this is not the case. In the next section, Section 8.3.2, we look at a more complex problem and compare a recursive solution with pieces of an iterative solution that doesn't have the benefit of the os module's `walk` generator.

Recursion is, at least to me, a tough idea to grasp. I'd encourage you to read more resources on recursion and, most importantly, to practice writing recursive functions. Here are a few resources I'd recommend:

- Python Course: A reasonably accessible introduction, but its even more mathematical examples may not be motivating to MIS professionals: http://www.python-course.eu/recursive_functions.php.
- CodeStepByStep: Scroll down for two recursion practice problems: http://www.codestepbystep.com/problem/list.
- Python Data Structures and Algorithms: Recursion — Exercises, Practice, Solution: A bunch of exercises with solutions: http://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-recursion.php.
- Recursion Examples (Python): A bunch of small problems with solutions: https://frankanya.wordpress.com/2013/04/18/recursion-examples-python/.

8.3.2 Implementing a search using recursion

Let's say we want to return a list of all files in the Figure 7.1 directory tree that ends in *.txt*. We'll do this first recursively then (try to) iteratively without the benefit of the os module's `walk` generator.

A recursive solution

How might we do this recursively? In terms of logic, we might say, "look into the directory, return a list of all *.txt* files and if you find a directory, look into the directory, etc." In terms of code, we could write the following `find_txt` function which would implement that logic:

```python
import os
import os.path

def find_txt(top_dir):
    list_items = os.listdir(top_dir)
    list_txt = []
    for item in list_items:
        item_path = os.path.join(top_dir, item)
        if os.path.isdir(item_path):
            list_txt = list_txt + find_txt(item_path)
        else:
            if os.path.splitext(item)[-1].lower() == '.txt':
                list_txt = list_txt + [item_path]
    return list_txt
```

and a call to `find_txt("MyFiles")` yields the following list:

```
['MyFiles/BusinessData/Old.txt',
 'MyFiles/BusinessData/July/Day1.txt',
 'MyFiles/BusinessData/June/Day15.txt',
 'MyFiles/BusinessData/June/Day1.txt',
 'MyFiles/BusinessData/June/Day30.txt']
```

(I hand-formatted the output above so it'd look readable but the values are what a `print` statement produces on the return value from the call. Also, my solution assumes there are no symbolic links in the directory tree.)

Let's go through each line of `find_txt` and describe what's happening. In lines 1–2 we import the modules we'll need. Line 4 defines the function. Note that the function takes a single input string, `top_dir`, which is the name of the directory in which we will look for all *.txt* files. Line 5 saves a listing of all the items (files and directories) in `top_dir`. Note that the `listdir` method only gives the names (i.e., the basename) of the items; it does not provide the absolute path to the items. In line 6 I initialize an empty list of all the *.txt* files in `top_dir`. For a given `top_dir`, this list will be initially empty since we haven't processed the listing of the items in `top_dir` yet. (You may ask how we'll account for *.txt* files in sub-directories of `top_dir`; I'll discuss that in just a sec.)

In line 7, we loop through each item and for each item that ends in *.txt* (line 12), we add that to `list_txt` (line 13). Note that in the test in line 12, we use the `lower` string method to ensure our comparison with the *.txt* suffix will work, even if the operating system is case-sensitive (e.g., we're considering *.txt* and *.TXT* files as both *.txt* text files). Remember that the `splitext` function breaks any path into the file extension portion and the non-extension portion, with the extension portion being the last item of the list.

If an item in `top_dir` is a directory, we make a call to `find_txt` (this is the recursive call) to obtain the list of *.txt* files from that directory. Note that the path you pass into the recursive `find_txt` call (as well as in the line 9 check with `isdir` has to start with the initial `top_dir`. That is, it has to start with the `top_dir` you passed in on the original call to `find_txt`; in the example above, that was `find_txt("MyFiles")`. The reason the path needs to begin with *MyFiles* is because the function doesn't change the current working directory. As a result, the current working directory is still the parent directory of *MyFiles*, and any item inside the directory tree will remain hidden if the path doesn't go through *MyFiles*.

Note also that line 8 will automatically keep on adding whatever the current item is to the directory path that began with *MyFiles* and continues to whatever sub-directory contains the item refered to in `item_path`. This is because the line 10 `find_txt` call's argument becomes the new `top_dir` in that recursive `find_txt` call, and `join` adds on new sub-directories to that `top_dir` in line 8.

In lines 10 and 13, we don't use `append` to extend the `list_txt` list because the list returned from the line 10 `find_txt` call in line 10 may be more than one element. The addition sign concatenates two lists, which is why we use that in lines 10 and 13. (We could use `append` in line 13, but we use the addition operator for the sake of symmetry.) Here's a simpler example of list concatenation versus appending:

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

from which we see that appending a list to a list does not concatenate the two lists together.

Finally, in line 14, we return the output from `find_txt`, which is the list of all *.txt* files in the directory tree given by `top_dir`.

An attempt at an iterative solution

We've seen that recursion will solve our problem in 14 lines of code. What if we took an iterative approach (i.e., used loops instead of recursion)? Now, in asking this, we have to caveat it with the constraint that we aren't allowed to use the os module's `walk` function. With the `walk` function, we can write a very simple loop that will solve the problem.

The first issue we encounter when trying to write this using only loops is how to deal with sub-directories. We could, for instance, write the following:

```
import os
import os.path

def find_txt(top_dir):
    list_items = os.listdir(top_dir)
    list_txt = []
    for item in list_items:
        item_path = os.path.join(top_dir, item)
        if os.path.isdir(item_path):
            list_items_2 = os.listdir(item_path)
            for item_2 in list_items_2:
                item_2_path = os.path.join(top_dir, item_2)
                [... more code ...]
```

In lines 9–12, when we hit a sub-directory, we create another `for` loop to go through those items. If that list of items includes another directory, we would add another nested `for` loop to go through that sub-directory, and so on.

The problem with this approach is that we don't know, ahead of time, how many levels of directories there are. With two directory levels we need two nested loops, with three directory levels we need three nested loops, and so on. But there could be one level of directories, or two levels, or twenty levels. Do we write a twenty-level nested `for` loop structure in case there are that many directory levels? And how does one write a twenty-level nested `for` loop structure? I can't imagine any way of doing so that is remotely readable.

So, what are the pluses and minuses for recursion versus loops? Loops are generally more readable and understandable and the number of iterations you can make with loops is more or less unlimited. Recursive solutions, in contrast, can be confusing and there is a limit to the number of recursion levels available (on my computer it's 1000).[4] However, we've seen there are some problems where the loops-only solution is so complex that recursion is, by far, the best way to go. For the walk-through-a-directory tree problem, Python provides a generator function `walk` that makes recursion unnecessary for most such uses of directory trees, but there are other problems besides directory traversal ones where recursion is the only way to solve the problem in a reasonable number of lines of code. If it only makes sense to state the problem logic in such a way that there is an element where an action is defined in terms of itself, it's worth trying a recursive solution.

8.4 Summary

Python provides a number of tools to help us find things, whether in a list of items or a tree of items. With directory trees, the os module's `walk` function makes it possible to loop through the entire tree using a single `for` loop. Directory trees, however, can also be accessed using a recursive solution. The recursive technique, while somewhat tricky to understand, can enable us to solve certain kinds of problems with concise and elegant code instead of a spaghetti-like mess a loops-only solution would force on us.

[4]The sys module has a function `getrecursionlimit` that tells you what that limit is. See https://docs.python.org/2/library/sys.html (accessed November 14, 2016).

Chapter 9

Designing Power Programs

Chapter Contents

With what we've learned so far—using Python to make calculations, plot graphs, access data in files, manage files and file processes—we can write powerful Python functions and programs to provide the kinds of data analysis, business intelligence, and information system management that can contribute towards all aspects of a business. However, for highly complex tasks, our current toolkit, while very powerful, could use a little boost. It's like building a house with a hammer and hand saw. You can do it, and human beings did exactly that for thousands of years, but it really makes a difference if you have a nail gun and power saw with you.

In this chapter, we will be given these power tools through learning about and how to use OOP to help us create more complex programs. In Chapter 5, we learned about objects and how to use them. Here, we'll learn how to write our own objects in Python and, more importantly, see how to use these objects to manage more involved information system-related tasks.

9.1 What is object-oriented programming

Object-oriented programming (OOP), deservedly or not, has something of a reputation as an obtuse and mysterious way of programming. You may have heard that it is a powerful way of writing programs, but you probably haven't heard a clear and concise description of how it works to help you write better business programs. Unfortunately, I also cannot give you a clear and concise description of how OOP works to help you program.

The problem is not that I cannot describe to you what an object is or give you a definition of OOP, but rather that any description of the mechanics and use of OOP does not really capture how OOP makes your life easier as a business analytics programmer. It's like thinking that a description of oil pigments and poplar surfaces will somehow enable you to "get" how the Mona Lisa works. For both OOP and art, you can't describe the forest in terms of the trees. Instead, what we'll do is look at OOP in action; from that description, hopefully some sense of how OOP makes programs more organized, readable, maintainable, and usable will come through. Through these examples, I hope to describe both how to write object-oriented programs as well as why object-oriented programs work the way they do.

9.2 Case Study: Managing a bookstore's inventory

In Section 5.3.1, we said that objects are instances of a class. That is to say, they are specific realizations of a general pattern and their bulk characteristics are defined by a template. Each individual person is an instance of the class "human being" and somewhere there's a template that says each person has one head, two eyes, etc. Each specific Honda, Chevy, Kia, etc. is an instance of the class "car," and somewhere there's a template that says each car has four wheels, a steering wheel, at least one door, etc. And in the string and array examples in Section 5.3, each string is an instance of the class "string" and each array is an instance of the class "array," and somewhere there are templates that tell us strings have a `split` method and arrays have an `astype` method.

In the present section, we will create objects that represent the inventory of a bookstore. The inventory of a bookstore (or any other store) does not consist of items that are each completely different from every other item in the store. Rather, there are commonalities to groups of those items, and those groups we can call a "class" of items of objects. (If this weren't the case, there would be no way to organize items on selves or aisles.) Here we define two classes of objects in this bookstore (there are, of course, many others): books and magazines. The Python name for these classes of objects will be called (no surprise): `Book` and `Magazine`. (In Python, class names follow the CapWords convention, where each word in the name is capitalized.)

Before we start writing code, let's think a little bit about what characteristics these classes of objects would have. The class of the object is the template that defines the characteristics of the specific realizations or instances of that object. Those characteristics are either data or information (attributes) or functions that act on that data (methods). Both attributes and methods are bound to an object; they are elements or characteristics of the object.

What might be some characteristics of a book? Attributes of a book might include:

- Title
- Author
- Publisher
- Year
- Price
- Sale discounts (if any)
- Number of pages
- Linear dimensions
- Weight

and some methods that might act on those and related attributes might include:

- Return a summary description of the book
- Return the retail price
- Return the discounted price
- Calculates the volume the book occupies

How about for a magazine? Attributes might include information similar to a book but will probably also include:

- Volume number
- Issue
- Publication date

In terms of methods, besides those associated with a book, we might also have methods that:

- Return the per year subscription rate
- Return the address of the publisher's subscription department

With that as background, let's write our class definitions. Here is a definition of the `Book` class that about a book. The class definition provides a single method (besides the initialization method) that returns a formatted summary description for `Book` objects. In addition, the code below creates two instances of the class (note line continuations are added to fit the code on the page):

```
1  class Book(object):
2      def __init__(self, authorlast, authorfirst, \
3                   title, place, publisher, year):
4          self.authorlast = authorlast
5          self.authorfirst = authorfirst
6          self.title = title
7          self.place = place
8          self.publisher = publisher
9          self.year = year
10
11     def summary_descrip(self):
12         return self.authorlast \
13             + ', ' + self.authorfirst \
14             + ', ' + self.title \
15             + ', ' + self.place \
16             + ': ' + self.publisher \
17             + ', ' + self.year + '.'
18
19 beauty = Book( "Dubay", "Thomas" \
20              , "The Evidential Power of Beauty" \
21              , "San Francisco" \
22              , "Ignatius Press", "1999" )
23 pynut = Book( "Martelli", "Alex" \
24             , "Python in a Nutshell" \
25             , "Sebastopol, CA" \
26             , "O'Reilly Media, Inc.", "2003" )
```

What does each line do? Line 1 begins the class definition. Class definitions start with `class` statement. The block following the `class` line is the class definition.

The argument in the class statement is a special class called `object`. This has to do with the OOP idea of inheritance, which is a topic beyond the scope of this book. Suffice it to say that classes you create can inherit or incorporate attributes and methods from other classes. Base classes (class that do not depend on other classes) inherit from `object`, which is a built-in class in Python that provides the foundational tools for all other classes.

Notice how attributes and methods are defined, set, and used in the class definition. Within the class definition, you refer to the instance of the class as `self`. So, for example, the instance attribute year is called `self.year` in the class definition. If I decided to make use of the `summary_descrip` method elsewhere in the class, I'd refer to it as `self.summary_descrip` in the class definition (and it would be called by `self.summary_descrip()`).

When you actually create an instance, the instance name is the name of the object (e.g., `beauty`, `pynut`), so the instance attribute `title` of the instance `beauty` is referred to as `beauty.title`, and every instance attribute is separate from every other instance attribute (e.g., `beauty.title` and `pynut.title` are separate variables, not aliases for one another).

Attributes are created and set by assignment. If you want to create and set the `title` attribute, type in:

```
self.title =
```

and then put what you want it to equal to on the right-hand side of the equal sign. Methods are defined using the `def` statement. The first parameter in any method definition is `self`; this syntax is how Python tells a method "make use of all the previously defined attributes and methods in this instance." However, you never type `self` in the parameter list when you call the method. So, the `summary_descrip` method definition in lines 11–17 has `self` in the parameter list but if we call that method, the parameter list will be empty.

Usually, the first method you define will be the `__init__` method. This method is called whenever you create an instance of the class, and so you usually put code that handles the arguments present when you create (or instantiate) an instance of a class and conduct any kind of initialization for the object instance. The arguments list of `__init__` is the list of arguments passed in to the constructor of the class, which is called when you use the class name with calling syntax.

Thus, in lines 4–9, I assign each of the positional input parameters in the `def __init__` line to an instance attribute of the same name. (The attributes and input parameters don't have to be of the same name, but it's convenient here to do so.) Once assigned, these attributes can be used anywhere in the class definition by reference to `self`, as in the definition of the `summary_descrip` method.

In lines 19–22, I create an instance `beauty` of the `Book` class. Note how the arguments that are passed in are the same arguments as in the `def __init__` argument list. In the last four lines, I create another instance of the `Book` class, the object `pynut`. Both `beauty` and `pynut` are instances or specific realizations of the class (or template) `Book`.

Now that we've seen an example of defining a class, let's look at an example of using instances of the `Book` class:

Example 33 (Using instances of `Book`):

Consider the `Book` definition given above. Here are some questions to test your understanding of what it does:

1. How would you print out the `author` attribute of the `pynut` instance (at the interpreter, after running the file)?

2. If you type `print(beauty.summary_descrip())` at the interpreter (after running the file), what will happen?

3. How would you change the publication year for the `beauty` book to `"2010"`?

Solution and discussion: My answers:

1. Type: `print(pynut.author)`. Remember that once an instance of `Book` is created, the attributes are attached to the actual instance of the class, not to `self`. The only time `self` exists is in the class definition.

2. You will print out the the bibliography formatted version of the information in `beauty`.

3. Type: `beauty.year = "2010"`. Remember that you can change instance attributes of classes you have designed just like you can change instance attributes of any class; just use assignment.

Note that in Python, you don't generally have to write getters and setters (special methods to return and set the values of attributes) like you do in languages like Java, where privacy is typically strongly controlled.

Now let's define the `Magazine` class:

```
class Magazine(object):
    def __init__(self, journaltitle, \
                 volume, monthday, year):
        self.journaltitle = journaltitle
        self.volume = volume
        self.monthday = monthday
        self.year = year

    def summary_descrip(self):
        return self.journaltitle \
            + ', ' + self.volume \
            + ', ' + self.monthday \
            + ', ' + self.year + '.'
```

This code is similar to that for the `Book` class, with these exceptions: some attributes differ between the two classes (books, for instance, do not have months and days) and the method `summary_descrip` is different between the two classes (to accommodate the different formatting between book and magazine entries).

So, using OOP, we now have objects that store elements of our bookstore's inventory. That's handy, in and of itself. While lists of numbers lend themselves nicely to arrays, most objects in the real world are described by a motley assortment of characteristics. Python objects allow us to store all these characteristics and operate on them in a clear and tidy way.

But let's not just let the objects sit there; let's do something with these objects. Since we have a method to return a summary description of each objects, let's write some code to print out that summary description. In the code below, I first define some `Book` objects and `Magazine` objects, the print out summary descriptions of the objects (I didn't duplicate the `Book` and `Magazine` definitions):

```
1  beauty = Book( "Dubay", "Thomas" \
2               , "The Evidential Power of Beauty" \
3               , "San Francisco" \
4               , "Ignatius Press", "1999" )
5  pynut = Book( "Martelli", "Alex" \
6              , "Python in a Nutshell" \
7              , "Sebastopol, CA" \
8              , "O'Reilly Media, Inc.", "2003" )
9  good = Magazine( "Really Good Magazine", "39" \
10                , "April 1", "2001" )
11 world = Magazine( "The World Today Magazine", "11" \
12                 , "September 19", "2016" )
13
14 inventory = [beauty, pynut, good, world]
15 for item in inventory:
16     print(item.summary_descrip())
```

What are we to make of this bit of code? What does using objects buy us? To answer this, let’s ask how would we have written a program without objects that did the exact task of writing out the summary descriptions of these items in our bookstore inventory that we do in lines 14–16. Let’s say the information for each book or magazine, instead of being stored in an object, was stored in a list. For instance, pynut and good might be given as:

```
pynut = [ "Martelli", "Alex", "Python in a Nutshell" \
        , "Sebastopol, CA", "O'Reilly Media, Inc.", "2003"]
good = [ "Really Good Magazine", "39", "April 1", "2001" ]
```

But in order to properly write out the summary description for each item, we need to know what kind of item it is. Then, when we loop through all the items to write out the summary descriptions, we can use an `if` test to see what kind of item it is and choose the correct format to write out the descriptions. Thus, our code might look like this:

```
beauty = [ "Book", "Dubay", "Thomas" \
         , "The Evidential Power of Beauty" \
         , "San Francisco", "Ignatius Press", "1999" ]
pynut = [ "Book", "Martelli", "Alex", "Python in a Nutshell" \
        , "Sebastopol, CA", "O'Reilly Media, Inc.", "2003"]
good = [ "Magazine", "Really Good Magazine", "39", "April 1" \
       , "2001" ]
world = [ "Magazine", "The World Today Magazine", "11" \
        , "September 19", "2016" ]

inventory = [beauty, pynut, good, world]
for item in inventory:
    if item[0] == "Book":
        print( item[1] \
            + ', ' + item[2] \
            + ', ' + item[3] \
            + ', ' + item[4] \
            + ': ' + item[5] \
            + ', ' + item[6] + '.' )
    elif item[0] == "Magazine":
        print( item[1] \
            + ', ' + item[2] \
            + ', ' + item[3] \
            + ', ' + item[4] + '.'
    else:
        raise ValueError("item type does not exist")
```

How does this code compare to our previous solution using objects? For starters, the code is much less readable. Who knows what `item[2]` corresponds to? But more importantly, let's look at the `if` statement. If we don't use OOP to break-down and organize our data and methods, we have to use an `if` statement to enable us to properly treat different kinds of items in our bookstore's inventory. That's okay if our inventory has just two kinds of items, but what if there are hundreds or thousands? If we don't use OOP, we'll have to write an `if` statement with hundreds or thousands of branches. And we have to do that *every time* we want to do something with those inventory items, whether that's printing out a summary list, calculating the shipping weight of a collection of items, etc. This scenario is a nightmare!

But with OOP, this crazy `if` statement goes away! It doesn't matter if we have two kinds of inventory items or two million. As long as the class definitions for each of those two million inventory items has a `summary_descrip` method, the code to print out summary descriptions of all items in the bookstore's inventory remains unchanged:

```
for item in inventory:
    print(item.summary_descrip())
```

Wow! That saves a lot of work and makes the program so much clearer, less buggy, and extendable ☺!

9.3 ⌨ Python Sidebar: The `NoneType` data type

This is a data type you probably have not seen before. A variable of NoneType can have only a single value, the value `None`. (Yes, the word "None," capitalized as shown, is defined as an actual value in Python, just like `True` and `False`.)

Example 34 (Operations with NoneType):
Try this in a Python interpreter:

```
a = None
print(a is None)
print(a == 4)
```

What did you get?

Solution and discussion: The first `print` statement will return `True` while the second `print` statement will return `False`.

The `is` operator compares "equality" not in the sense of value (like `==` does) but in the sense of memory location. You can type in "`a == None`", the better syntax for comparing to `None` is "`a is None`".[1] The `a == 4` test is false because the number 4 is not equal to `None`.

So what is the use of a variable of NoneType? I use it to "safely" initialize a keyword input parameter or attribute. That is to say, I initialize a variable to `None`, and if later on my program tries to do an operation with the variable before the variable has been reassigned to a non-NoneType variable, Python will give an error. This is a simple way to make sure I did not forget to set the variable to a real value. Remember variables are dynamically typed, so replacing a NoneType variable with some other value later on is no problem!

9.4 Summary

You could, I think, fairly summarize this chapter as addressing one big question: Why should an MIS student bother with object-oriented programming? In short, code written using OOP is less prone to error. OOP enables you to mostly eliminate lengthy argument lists, and it is much more difficult for a function to accidentally process data it should not process. Additionally, OOP deals with long series of conditional tests much more compactly; there is no need to duplicate `if` tests in multiple places. Finally, objects enable you to test smaller pieces of your program (e.g., individual attributes and methods), which makes your tests more productive and effective.

Second, programs written using OOP are more easily extended. New cases are easily added by creating new classes that have the interface methods defined for them. Additional functionality is

[1]The reason is a little esoteric; see the web page http://jaredgrubb.blogspot.com/2009/04/python-is-none-vs-none.html if you're interested in the details (accessed August 16, 2012).

also easily added by just adding new methods/attributes. Finally, any changes to class definitions automatically propagate to all instances of the class.

For short, quick-and-dirty programs, procedural programming is still the better option; there is no reason to spend the time coding the additional OOP infrastructure. But for more involved business applications, things can very quickly become complex. As soon as that happens, the object decomposition can really help. Here's the rule-of-thumb I use: For a one-off, short program, I write it procedurally, but for any program I may extend someday (even if it is a tentative "may"), I write it using objects.

Glossary

absolute path the path to a directory or file starting from the root directory (on Linux) or the drive letter.

argument an item passed into a function as input; there is a subtle distinction from a parameter, but the two have similar meanings.

attribute data bound to an object that are designed to be acted on by methods also bound to that object; sometimes attributes are called "instance variables".

calling execute or run a function.

class the template or "pattern" all instances of that class follow.

current working directory the directory you are currently in and that Python will base all relative file and directory references from.

data coordinates a coordinate system for a plot where locations are specified by the values of the x- and y-axes data ranges.

delimit show where a sequence or collection begins and ends.

docstring a triple-quote delimited string that goes right after the `def` statement (or similar construct) and which provides a "help"-like description of the function.

dynamically typed variables take on the type of whatever value they are set to when they are assigned.

exception an error state in the program that cannot be processed by the current scope.

filesystem the collection and arrangement of all the files on a computer.

immutable a variable/object that cannot be changed.

inherit incorporate the attribute and method definitions of another class into a definition of a new class of objects.

inheritance dealing with inheriting attribute and method definitions of another class into a definition of a new class of objects.

instance an object that is the specific realization of a class of objects.

instantiate create an instance of a class.

interpreter the execution environment for Python commands.

iterable a data structure that one can go through, one element at a time; in such a structure, after you've looked at one element of it, it will move you on to the next element.

keyword input parameter a parameter set by reference to a name or keyword rather than by position in a list.

leaf directory the final directory in a directory path.

method functions bound to an object that are designed to act on the data also bound to that object.

module an importable Python source code file that typically contains function, class, and variable object definitions.

multi-paradigm language a computer language that supports multiple programming methodologies, for instance, object-oriented programming and procedural programming.

mutable a variable/object that can be changed.

newline character a special text code that specifies a new line; the specific code is operating system dependent.

object a "variable" that has attached to it both data (attributes) and functions designed to act on that data (methods).

package a directory of importable Python source code files (and, potentially, subpackages) that typically contains function, class, and variable object definitions.

parameter an item passed into a function as input.

parameter list an list of items passed into a function as input.

path the listing of what directories you need to go through to reach the file or directory at the end of the path.

recursion a realization of something recursive.

recursive related to the idea that we can define some tasks in terms of themselves; this is expressed in terms of a function partially being defined by calls to itself.

relative path the path to a directory or file starting from the current directory.

script a file of Python commands or instructions.

scripting writing a script.

shape a tuple whose elements are the number of elements in each dimension of an array; in Python, the elements are arranged so the fastest varying dimension is the last element in the tuple and the slowest varying dimension is the first element in the tuple.

shell a text-based interface to an operating system, the software that manages your files, directories, etc..

terminal window a text window in which you can directly type in operating system and other commands.

typecode a single character string that specifies the type of the elements of a NumPy array.

Acronyms

CSV comma-separated values.

dpi dots per inch.

GUI graphical user interface.

IDE Interactive Development Environment.

MIS Management Information Systems.

OOP object-oriented programming.

Bibliography

Lin, J. W.-B. (2012). *A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences*. Chicago, IL.