A Whirlwind Tour of Programming in Python

P. K. G. Williams (peter@newton.cx)

May 28, 2017

version `68a9cf9-dirty`

Introduction

This handout accompanies a 90-minute introduction into computer programming using the Python language. It is intended for students who have never programmed a computer using a formal language before, although we hope that some of the ideas we present will be interesting to more experienced students as well.

It is *totally impossible* to teach anything but the most superficial elements of the Python language or the principles of computer science in 90 minutes.

Given that reality, for this lesson we have chosen to just *visit* a few topics that are important to the craft of writing software and illustrate them with some *extremely limited* examples in Python. Our treatments of both the "big picture" ideas *and* the basics of writing Python programs will be dissatisfying. We hope, however, that this lesson will plant seeds that will grow over time.

Due to the context in which this lesson was written, it is aimed at *version 2* of Python, not version 3. Students are expected to have a laptop available that is running the interactive Jupyter notebook. The lesson is divided into seven activities, each of which has a "mini-lecture" component and a hands-on component. Each activity is intended to take about 15 minutes to complete.

Hands-on #1: Foundations

In this hands-on activity, we'll just make sure that you can get Jupyter running and show you some basics about how to run commands in a Jupyter notebook. Because we're just getting started, there's a lot more explanatory text than you'll find in the other activities.

If you haven't already, start Jupyter and create a new, empty notebook. We've written this lesson assuming that you've already seen how to do this — ask for help from a neighbor or a helper if not, or if you run into any problems.

In this handout, text styled the following way is code that you should type into your notebook.

```
print 'Hello, world!'
x = 1
```

Type this code now, *exactly* as it is shown here — be careful with punctuation and spaces. When using Jupyter, start new lines of code by just hitting the Enter key as usual.

(By the way, we are *very intentionally* forcing you to type this code yourself! You'll have to type a lot of code yourself eventually; might as well start now.)

To actually *run* the block of code that you have just typed, type Shift-Enter (that is, the Enter key with the Shift key also held down). You should see the message "Hello, world!" appear and a new block for code entry appear. Each of these blocks is a *cell* in your notebook.

The horizontal bars delimit pieces of code that you should type in separate cells, typing Shift-Enter at the end of each cell. For instance:

```
x
```

```
y = 1.6
y
```

This means that you should type these two pieces code in separate cells, running the first one before you start typing the second one. Do so now. You should see the values "1" and "1.6" printed back at you.

From here on out, if you see a code listing as above, always assume that you should type it in and run it in your notebook. Skipping cells might cause later demonstrations to fail.

There is an exception to our request that you type things precisely as written. Sometimes we'll write some explanatory text as Python *comments*. These are delimited with hash signs (#). In these activities you don't need to type the comments. When there's a line with a #, skip it and everything that comes after it.

```
x = 7 #<== don't bother typing this or anything after it
x
```

Every so often we intentionally have you type a command that results in an error. We will indicate this with a comment about "raising" a particular kind of error:

```
assert x == 2 # !!! raises AssertionError
```

Running this line should produce some red text and technical information indicating an error. Eventually you should learn how to make sense of the information, but we'll ignore it for now.

Only lines with these "raise" annotations should result in error messages. These lines will come at the ends of cells, since any code within a cell that comes after an error will not be run. If some other line of code causes an error, something has gone wrong and you should ask a neighbor or helper to take a look.

Because of the limited time available, we're going to have to ask you to do a lot of "monkey see, monkey do" coding: we'll have you type in code without giving you a fair chance to be able to understand it. For instance, type and run this code, *making sure to reproduce the spaces after the* while *keyword*:

```
import pylab as pl
import numpy as np
i = 99
x, y = np.mgrid[-2:2:999j, -2:2:999j]
c = z = x * 1j + y
while i:
    x[np.abs(z) > 2] = i
    z = z * z + c
    i -= 1
pl.show(pl.imshow(x))
```

If you've done this right, you'll see some warning messages get printed in red, and then an image of the Mandlebrot set will appear after a few seconds. You might have to scroll your notebook down to see everything.

The demo above might be neat, but we recognize that it is not very instructive, if not actively frustrating, for most people. Apologies — we have made a choice to try to expose you to several big ideas about programming, at the expense of building a solid foundation for those ideas.

But a foundation is essential! We recommend a textbook, *How to Think Like a Computer Scientist: Learning with Python*, by Allen Downey, Jeffrey Elkner, and Chris Meyers. Download it here for free:

http://www.greenteapress.com/thinkpython/thinkCSpy/thinkCSpy.pdf

Download it now and save it on your computer. Your homework is to *start reading the textbook*. The end of this handout points you to a variety of other resources aimed at helping people learn to code using Python.

One last thing: we do not have you create much code from scratch in this lesson, but feel free to experiment as you go. Students often come to us with a piece of code and ask, "Will this work?" To which we always reply, "Just try running it and find out!" *The worst that can happen is that it won't work*. Which isn't so bad, is it? Python and Jupyter are great for experimentation — take advantage of it!

Hands-on #2: Data structures

In this activity we will practice creating various kinds of Python data objects and try to demonstrate how the ideas of *indirection* and *mutability* are important in computing.

Foundational data types; printing your data

Assign some values of different types to some variables:

```
a_none_variable = None
negative_int = -100
one_million = 1e6 # "exponential" notation for floats: = 1 * 10**6
a_string = 'Using python'
```

When using the Jupyter notebook, if you just type in a variable's name in a cell and hit Shift-Enter, you will be shown the current value of that variable:

```
one_million
```

It is very important and valuable to examine your variables as you manipulate them. *When in doubt, print it out!* It never hurts.

If you want to look at multiple variables, you have to put each one's name in a different cell:

```
one_million # does not reappear
a_string
```

You can do math with numeric variables by typing in standard mathematical notation. Multiplication is expressed with a single asterisk (`*`). Exponentiation (a^b) is written with a double asterisk (`a**b`).

```
new_number = one_million / negative_int
other_number = (new_number * 0.01) + 5**2
other_number
```

Lots of coders end up typing their math densely for some reason: `(5*i)+3` instead of `(5 * i) + 3`. This is ridiculous. Spaces make things easier to read! And they are free! Hit the spacebar liberally.

Lists

As presented in the mini-lecture, Python's *lists* are ordered sequences of other data. Lists can hold any kind of data.

```
list_of_ints = [1, 2, 5]
mixed_list = ['Hello', one_million, list_of_ints]
mixed_list
```

Unlike the basic data types, lists are mutable — you can modify their contents. We do this by *indexing* the list, using square brackets to identify a specific one of its *elements* (or *items*) that we care about, rather than the whole list at once:

```
list_of_ints[1] = 3
list_of_ints
```

There might be something surprising about what just got printed out. We now present you with one of the first mystical secrets of programming:

The "first" element in a list has an index of zero, not one!

Every single person who starts out programming has trouble getting used to this. Rest assured: there are very good reasons for this convention . . . but not so good that we think they'll be compelling just yet.

```
list_of_ints[0]
```

```
list_of_ints[2]
```

Indirection; mutability

In the mini-lecture, we tried to emphasize that when you assign a variable, you are creating a name that lets you refer to some piece of data. The name and the data are not the same!

The following classic example demonstrates the point. Try hard to understand what's going on here.

```
my_list = [1, 2, 3]
print 'Before:', my_list
other_list = my_list
other_list[1] = 37
print 'After:', my_list
```

The "before" and "after" values of my_list are different, even though it didn't *look* like you did anything to it! But while the variables my_list and other_list might have different names, they both refer to the same piece of data. So altering other_list — that is, the piece of data referenced by the variable other_list — alters my_list as well.

Please note that *this kind of surprise can only happen because lists are mutable.* This is one reason that mutable data should be handled carefully. If you share a mutable piece of data in two different pieces of code, it is all too easy to run into problems because an alteration in one piece wasn't intended to affect the other piece, but did anyway. In complex programs these problems can be *very* difficult to track down.

Tuples

In fact, Python specifically provides immutable versions of some of its data structures. The immutable version of a list is called a *tuple*. They are written down like lists, but with parentheses instead of square brackets. However, in a surprising number of cases the parentheses are actually optional!

```
a_tuple = (0, 3, 4)
equivalent_tuple = 0, 3, 4 # this is fine; parens not actually needed
other_tuple = (a_tuple, 'hello', my_list)
other_tuple
```

You should sense that mutability must be important, if Python bothers to provide two data types that are essentially identical *except* for their mutability.

It gets more complicated, though, because tuples themselves are like named variables: they only refer to their contents *indirectly*. So while you can't change a tuple, if that tuple happens to contain a mutable variable, the tuple can appear to change:

```
print 'Before:', other_tuple
my_list[0] = -1234
print 'After:', other_tuple
```

You might be starting to see why indirection has caused headaches for so many programmers over the years!

Hands-on #3: Control Flow

In this activity we will explore some of Python's constructs for controlling program flow and think about ways to keep the flow clear.

To demonstrate these concepts, we are going to step away from Jupyter and instead use an interactive debugger called PuDB. Below, sentences beginning with ↦ are ones that tell you what to type.

Conditionals

Start by opening a new terminal window. ↦ Run the following command in your terminal:

```
$ pudb control-flow.py
```
Launch PuDB on our script.

PuDB uses a semi-graphical interface in your terminal. It's not intuitive to use, but we'll try to walk you through it carefully. Avoid hitting extraneous keys, though, since you might put PuDB in a mode where our pre-written instructions stop working.

When you launch it, PuDB will open a welcome message. You don't need to read it. ↦ Hit Enter to make it go away.

You will then get a window showing preferences you can change. The defaults are fine. ↦ Hit the right arrow key to highlight the OK box, then Enter to close the window.

PuDB will now show you a screen with a lot of sections that are mostly empty. It has loaded up our demo script and paused it on the first line. ↦ Hit the n key to run the program one step, so that it advances to the next line.

You'll see that in the "Variables" display, in the top-right of your screen, a line saying x: 0 has appeared, because our program just assigned the variable x the value 0. In the main window, the indicator has advanced to the next non-blank line, an if statement, showing us where the program is paused.

Take a moment to guess how control will flow over the next few lines. ↦ Once you've done so, hit the n key a few more times *until you get to the line labeled "pause point A."* Stop there. Were you right?

We'll now restart the program, without bothering to let it finish. ↦ Do this by hitting the q key to open the "Finished" window, then hitting Enter.

↦ Hit the n key once to advance to the if statement once again. ↦ Now, type Control-x. The screen will recolor, showing that now the "Command line" section in the lower-left part of the screen is active.

↦ Type x = 7 in the command line, then hit Enter. ↦ Then hit Control-x again, which returns the "focus" of the program to the main program listing. ↦ Finally, hit n to advance to the next line. In the "Variables" display, you should see that x has changed to 7 — we were able to monkey with it interactively, while our program was running! Fun! ↦ Hit n again to get to pause point A.

Loops

We'll now navigate through a few loops. The first *stanza* of code, going up until "pause point B", calculates the greatest common denominator (GCD) of two numbers — although you might have to trust us on this point. The expression `a % b` computes the remainder of a divided by b.

↦ Hit `n` and watch the variables change until you reach "pause point B." The `while` loop doesn't have any complicated internal structure, so the control flow is pretty simple here.

The next stanza has a nonsensical `for` loop that invokes the `break` and `continue` statements. ↦ Hit `n` to step through the loop. At the top of each iteration, consider the values of the variables `z` and `k` and predict how the program's control flow will progress. ↦ Proceed with `n` until you reach "pause point C."

Avoiding rightward drift

The control flow of a program should, ideally, be fairly straightforward, and one should be able to get a sense of it from skimming the code. The syntax of the Python language generally encourages this.

However, when you're writing code for not-so-simple computations, it is easy for the control flow to get hard to follow. For instance, consider the next stanza of code in our demonstration file. Given a span of time measured in seconds, it converts it into an English string describing that span of time, ranging from "seconds" all the way to "years."

To give you a chance to experiment with how the code works, we've implemented this conversion in a loop. Each iteration, the code considers the variable `seconds` and places the English description of the time span in the variable `msg`.

↦ Press `n` to step through one full iteration of the loop. You'll see the control flow jump around as the necessary numbers are computed.

We think that you'll agree that the logic in this code is hard to follow at a glance — the `else` statements are widely separated from the `if` statements, and the indentation gets deeply nested. Each level of indentation brings with it a piece of context that you have to remember when reading the code, which makes it confusing. In real programs, logic is often *not* simple, and it's easy to get this kind of deep nesting — a phenomenon called *rightward drift*.

↦ At the top of the loop, you can use `Control-x` to change the value of the `seconds` variable to try out different settings, using `n` to step through the code. When you've seen enough, use `Control-x` to change the value of `keep_going` to `False`. *The loop won't exit until you do this!* Move ahead to pause point D.

In the next stanza of code, we implement the same logic. But this time, we use `continue` statements to avoid rightward drift. While the two pieces of code have identical functionality, we believe that this version is *much* easier to read.

↦ Step through the loop a few more times, using `Control-x` to try out different values of `seconds` and eventually exiting by setting `keep_going` to `False`. When you're all done, type `q` *twice* to exit PuDB.

Sophisticated programmers use statements like `continue` to keep their control flow tidy and avoid rightward drift.

Hands-on #4: Input and Output ("IO")

We will now try out some basic IO in Python and explore the "streaming" model for data sets.

For this hands-on activity, return to your Jupyter notebook.

The print statement

We've already encountered one form of IO: the print statement. It turns your Python data into strings and prints them to your Jupyter notebook or, if you're writing a command-line program, to your terminal.

Here's an *important disclaimer:* in Python 3, print() is a function, not a special statement. This is a much, much better way to do things, but it is not compatible with Python 2, and this fact makes it very annoying to port code from version 2 to version 3. *TODO: a good tutorial on the print_function future import!*. For annoying technical reasons, that's not a good solution for today, so we stick with the old-fashioned statement form of print.

The print statement is usually very straightforward:

```
x = 4
print 'x = ', x
```

It takes a comma-separated list of values, converts them into text, and prints them all on a line, separated by spaces. The fact that it only separates things with spaces can get a little annoying if you want to print out several variables at once. Here's our recommended pattern for doing so:

```
y = 17
z = 3.14
print '(x, y, z) =', (x, y, z)
```

This approach doesn't require you to type that many quotes or parentheses. It is tempting to just write statements like print x, y, z, but trust us: printing numbers without hints as to their context gets confusing quickly.

IO with files

The built-in open() function opens regular files on your computer's hard disk for input or output. We refer to the file's *open mode* as either being *read* or *write* depending on what you're going to do with it. The mode is denoted with a special string argument that is passed to open().

```
f = open('mydata.txt', 'w') # the "w" means to open for writing
```

The open() function returns a new form of Python data, an *object* — a *file object*, to be specific. These objects are their own type, different than floats, lists, etc.

File objects, and almost all other Python objects, come with functions attached to them. You can access and call these functions using "dot syntax" as shown below. In the first line, write() is a special function — a *method* — that operates on the variable f. Here the '\n' bit of the string is a special "escape code" that indicates that a new line of output should start.

```
f.write('Here is some text.\n')
another_name_for_f = f
another_name_for_f.write('Here is more text.\n')
```

There is a special way to invoke the print statement that prints data to a file instead of to the terminal:

```
print >>f, 'The value of x is', x
```

When you're done with a file that you've opened, close it:

```
f.close()
```

Once you've closed a file, the f variable is in a bit of a funny state: the variable still exists, but it's illegal to do anything with it:

```
print >>f, 'This will fail' # !!! raises ValueError
```

There's no need to assign the result of the open() function to a variable if we're not going to keep it around for long. Therefore, to print the contents of the file we just wrote, we can just do this:

```
print open('mydata.txt', 'r').read()
```

Here we are being naughty and not calling close() on the file object; that's forgivable if you're just reading the file in one go.

Network IO

Thanks to the infrastructure that Python provides, reading and writing over the network is almost as easy as reading and writing files from the local hard disk. For instance, saving a file from the internet to disk takes just a few lines of code:

```
from urllib2 import urlopen
from shutil import copyfileobj
source = urlopen('http://tinyurl.com/l66kod5')
dest = open('datatable.txt', 'w')
copyfileobj(source, dest)
source.close()
dest.close()
```

Above, the first two lines load standard Python *modules* that provide pre-packaged routines that allow us to avoid a lot of work that we'd otherwise have to do ourselves.

This cell probably took a half-second to run, since your notebook actually connected to the internet and fetched a file. While we didn't see any reassuring output, fear not: if no error message popped up, your download succeeded!

High-Level IO

The print statement is useful but it is not a good solution for anything other than ad-hoc diagnostics and messages intended for users.

For more important data, it is better to choose a preexisting, standard data format that is suited to the data. There is a Python module to read or write essentially any data format you can think of.

For instance, the file we just downloaded contains data in the textual table format developed by the Centre de Données astronomiques de Strasbourg (CDS). The popular Python module AstroPy can read these files into customized table data objects.

```
from astropy.io import ascii
table = ascii.read('datatable.txt')
table
```

Here, the AstroPy table object has special integration with Jupyter so that when we display its value in the notebook, you get a fancy representation of the table contents. Pretty fancy results for just 10 lines of code!

Streaming

When doing IO with data sets, it is good to try to deal with them in a *streaming* fashion when possible. That is, to process data as you work through a file, rather than reading in the whole file at once. Streaming is more efficient and flexible, and often just as easy to implement.

For instance, to loop through the lines in a text file, some people will write code like this:

```
f = open('mydata.txt') # if you don't specify the mode, it defaults to 'r'
data = f.read()
lines = data.splitlines()
f.close()

for line in lines:
    print 'A line from the file:', line
```

But you can use a file object in a for loop to stream the lines as they're read from the file:

```
f = open('mydata.txt')

for line in f:
    print 'A line from the file:', line.rstrip()

f.close()
```

Note that string data, like file objects, have methods attached — rstrip() in this case. This method returns a string that has had the newline code ('\n' from before) removed, which is needed for this style of reading lines from a file.

Now, to be honest: modern machines have so much memory that it is almost always fine to read in whole files at once. But if your file is a terabyte large, or your "file" is a *never-ending* stream of data, streaming is an important technique.

You can do some surprisingly sophisticated computations in the streaming paradigm. As a relatively simple example, let's say that we want to compute the mean $\langle M \rangle$ and variance σ_M^2 of the "Mass" column of the data table we read in, which is a catalog of stars. If there are N masses labeled M_i, these are defined as:

$$\langle M \rangle = \frac{1}{N} \sum_{i=1}^{N} M_i$$

and

$$\sigma_M^2 = \frac{1}{N} \sum_{i=1}^{N} (M_i - \langle M \rangle)^2,$$

which can be implemented in Python thusly:

```
mass_col = table['Mass']
mass_col = mass_col[~mass_col.mask] # drop rows without mass data
x = 0.

for i in range(len(mass_col)):
    x += mass_col[i] # same as: "x = x + mass_col[i]"

mean_mass = x / len(mass_col)

x = 0.

for i in range(len(mass_col)):
    x += (mass_col[i] - mean_mass)**2

mass_var = x / len(mass_col)
mean_mass, mass_var
```

But you may recall this statistical identity:

$$\sigma_M^2 = \langle M^2 \rangle - \langle M \rangle^2,$$

which makes it possible to do the same computation like so:

```
m_sum = 0.
m_sq_sum = 0.
n = 0

for mass in mass_col:
    m_sum += mass
    m_sq_sum += mass**2
    n += 1

mean_mass = m_sum / n
mass_var = m_sq_sum / n - mean_mass**2
mean_mass, mass_var
```

Now in this particular example, the full table of data has been loaded into memory, so we're not actually gaining anything. But our second, "streaming" approach *could* work with arbitrarily large tables, or ones whose size you don't know in advance, while the first approach cannot.

We should also mention that the first example shows a coding style that you should almost always avoid in Python: when working with lists of numbers, you should use "vectorized" routines provided by the "Numpy" module. We're tell you *much* more about this in the next session. However, while Numpy gives much better tools for calculating the mean and variance of a vector of data, it doesn't change the fundamental character of the approach, namely that it requires the full vector to be stored in memory.

Hands-on #5: Modules and Functions

In this hands-on activity we'll install and use a new Python package and create your own modules and functions.

Installing a "progress bar" package

Our goal is to create a module that provides a function that prints out a "progress bar," showing how far some time-consuming task has progressed — say, a download of a large file.

Someone has surely already published a package that prints out a progress bar. So, our first task is to survey the options. In a new web browser window, navigate to the website `https://pypi.org/`. The "PyPI" is a centralized database of a huge number of public Python packages. Run a search for "progress bar" to see what's available.

Yikes! There are literally *dozens* of options. This reflects a general truth of software libraries: the easier it is to implement something, the more different libraries there are to do it. Progress bars are very easy to implement.

Our decision-making is made easier because we prefer to use the conda package manager, which offers a much narrower selection of packages. Run the following command in your terminal to survey conda's offerings:

```
$ conda search progress
```

Search for Conda packages relating to "progress".

It's not important to understand the output of this program in detail. One thing that it is telling us, though, is that there is a package available named progressbar2. That sounds promising.

Return to your web browser opened to `https://pypi.org/` and search specifically for progressbar2. Click on the result for it — being careful because the search feature is a bit rough, and it doesn't put that one package front-and-center.

There's a lot of information here. Scroll down to the "Links" section of the page, which is about halfway down. Read over the list of links, noting that one of them is to "Documentation." Good: we don't want to waste our time with software that doesn't come with a manual. But, to save time, we won't look at the documentation right now.

Since this package meets our (very minimal) standards, let's go ahead and install it. Go back to your terminal and run this command, hitting Enter at the prompt to confirm the action:

```
$ conda install progressbar2
```

Install the "progressbar2" package.

Now, return to your Jupyter notebook. We can load up the module right away. Annoyingly, however, we must be careful because while the name of the *package* we just installed is progressbar2, the name of the *Python module* provided by that package is just progressbar. This sort of thing crops up often.

```
import progressbar
```

If no error message occurred, the import succeeded.

Writing your own module

We'll now create our own module that will demonstrate the features of progressbar2. Because we don't have time to develop a large piece of code, our module will necessarily be small and silly, but it's good to practice the process.

You can't use Jupyter to create modules, so we'll need to go back to the terminal and nano. First, use cd to navigate to the directory where your Jupyter notebook is being stored. Use ls to confirm that you're in the right directory. It is *essential* for this exercise that you create the module file in the same directory as your notebook.

Once you're sure that you're in the right directory, edit a new file:

```
$ nano myfirstmodule.py
```
Edit a new file.

Fill in the following code — again, this is to be typed in nano rather than in your Jupyter notebook:

```
from __future__ import division # we don't have time to explain this
from progressbar import ProgressBar
import time

string_variable = 'Hello from my module!'

def demo_progressbar(n_steps, total_time_seconds):
    time_per_step = total_time_seconds / n_steps
    pb = ProgressBar(maxval=n_steps).start() # "keyword" function argument

    for i in range(n_steps):
        time.sleep(time_per_step)
        pb.update(i + 1)

    pb.finish()
```

Save your file to disk. The way the progressbar module works, you create a special object that is told how many "steps" some task will take, and you call a method update() on it to report how many steps have been completed. As you do so, the object will print and update a textual progress report.

Back in your Jupyter notebook, you should be able to import your new module. Again, we're switching back from nano to Jupyter.

```
import myfirstmodule
```

If there's no error message, everything worked. If there is an error, ask a helper for assistance if you're at all unsure how to fix the problem.

The dir() function will return a list of the variables defined in your module.

```
dir(myfirstmodule)
```

Note that this list includes not just your string variable, but also your function, the other modules that your module imported, *and* some special variables with names like `__builtins__` — these are automatically created by Python within every module.

Finally, let's use the code and data provided by our module!

```
print myfirstmodule.string_variable
myfirstmodule.demo_progressbar(10, 5)
```

You should see a progress bar get filled up over the course of five seconds.

When doing real work, if you're making a module you probably want to reuse it in multiple projects. The approach that we just demonstrated is *not* desirable in those situations. Instead, you would create a `git` repository to track your module's development and write a special "`setup.py`" script allowing you to install it into your Python system's list of libraries. We don't have time to explain that whole process here, but there are many tutorials available online.

Hands-on #6: Classes

In this activity we'll implement our own new class.

"Stubbing out" our own ProgressBar

In particular, we're going to implement our own progress bar like the one provided by the progressbar2 package. If you review the code that we wrote for the last activity, you may guess that the ProgressBar name provided by the progressbar module is a class.

Jupyter notebooks are a bit unwieldy for writing your own classes, but the Jupyter editor is enough better than nano that we'll stick with Jupyter.

The ProgressBar class from progressbar2 had a constructor that took a "keyword" argument named maxval. We'll start by emulating that:

```
class ProgressBar(object):
    _maxval = None

    def __init__(self, maxval=None):
        self._maxval = maxval
```

Here we store the maxval argument in an instance-specific *property* called _maxval. The leading underscore (_) is a common encapsulation convention: it signals that that particular property is internal to the class, and you shouldn't access it from outside the class. (That is, you should avoid writing code that involves constructions like progress_bar_instance._maxval.)

But, for the sake of testing our work-in-progress, we're going to violate that convention! Let's see if things work so far.

```
pb = ProgressBar(maxval=10)
pb._maxval
```

This should run without errors and you should get back the number 10 that you specified as the maxval argument to the constructor.

Implementing start()

The ProgressBar class from progressbar2 also has a method called start(). Our start method will record the current time, for reasons that will become clear later.

Here's where Jupyter gets awkward. *Go back to the cell where you defined the ProgressBar class and edit it*, expanding the definition to look like the following. We have marked the new lines of code; don't bother to type in those comments.

```
import time # new

class ProgressBar(object):
    _maxval = None
    _start_time = None # new

    def __init__(self, maxval=None):
        self._maxval = maxval

    def start(self): # new
        self._start_time = time.time() # new
        return self # new
```

After editing, *re-run the cell* with Shift-Enter to override your previous class definition.

We now import the time module, which (among other things) provides a function named time() that returns the current time measured in seconds since January 1, 1970. This function is used in the start() method to record the time at which it is called. The start() method returns self, a convention that enables a convenient pattern called *method chaining* — Google it if you have time.

To validate our work so far, we'll violate the encapsulation convention again. Return to our usual practice and type the following code in a *new* notebook cell:

```
pb = ProgressBar(maxval=10).start()
pb._start_time
```

Note that we could save a line of code because start() returned self. The value stored in _start_time should be quite large, since there have been a lot of seconds since January 1, 1970.

Implementing update()

The next task is to implement the update() method, which takes as an argument the number of steps that have been completed. When this method is called on our simple progress bar, it will print out the progress toward the goal. Our progress bar will be simpler than the one provided by progressbar2.

Once again, return to your class definition cell and *edit it to add the following method definition inside the class block*, taking care to indent properly.

```
    # this code must be added to the ProgressBar class definition!

    def update(self, n_completed):
        percentage = 100.0 * n_completed / self._maxval
        print 'Task is', percentage, 'percent complete.'
        return self
```

Once again, rerun the cell that defines the class, then return to a new cell and run a quick test. Fortunately, we now have enough features that we can stop violating the encapsulation convention.

```
pb = ProgressBar(maxval=10).start()
pb.update(1)
```

If all has gone well, you should see a message that our task is 10 percent complete.

Implementing finish()

Finally, we'll implement a finish() method, since the progressbar2 version of the class had one. We'll use this opportunity to print out how much time has elapsed since the call to start().

Once again, go back and edit the cell that defines the ProgressBar class, appending the following method definition:

```
    # this code must be added to the ProgressBar class definition!

    def finish(self):
        elapsed = time.time() - self._start_time
        print 'Task took', elapsed, 'seconds to complete.'
        return self
```

Once again, rerun the cell defining the class.

Let's tie it all together by running the same code that we used in our module in the previous activity:

```
n_steps = 10
total_time_seconds = 5.0
time_per_step = total_time_seconds / n_steps
pb = ProgressBar(maxval=n_steps).start()

for i in range(n_steps):
    time.sleep(time_per_step)
    pb.update(i + 1)

pb.finish()
```

Our progress bar doesn't have the fancy output of the one provided by the progressbar2 class. But, if we wanted to, we could figure out how to make the fancy output and update the update() method to print out something more sophisticated. Because the ProgressBar class has a very simple *interface* to the rest of our code, it is easy to think about how to make such a change and to be confident that it won't break anything in a surprising way.

In the mini-lecture, we mentioned the idea of *statefulness* in objects. Note that our ProgressBar is essentially stateless: its internal data variables are never modified once they're set. Once again, avoiding *mutability* leads to code that is easy to reason about.

Hands-on #7: Error Handling

In this last activity, we'll explore Python exceptions and error handling.

Tracebacks

To *raise* your own exception in Python, you use the raise keyword with an instance of the Exception class or one of its subclasses.

```
raise Exception('demonstration error') # !!! raises Exception
```

The string argument to the Exception constructor is a message that will be printed if the exception is not handled.

You will see that if one of your cells raises an unhandled exception, Jupyter prints out a lot more information than just the message. In particular, there is a diagnostic pointing out which line in your cell caused the problem. In this particular case that was no big mystery, but in the real world it often is.

The unhandled-exception diagnostics include a key piece of information known as the *traceback*. The traceback shows not just the line of code where the exception occurred, but also the set of function calls that brought the program to that spot. An example might be helpful:

```
def inner():
    print 'Inner before'
    raise Exception('inner error')
    print 'Inner after'

def outer():
    print 'Outer before'
    inner()
    print 'Outer after'

outer() # !!! raises Exception
```

You should see that the traceback is now longer, showing the "stack" of function calls that led to the raise line. This contextual information can be very helpful.

It can be tempting to write code that "handles" exceptions by turning them into new exceptions. Such code might look like this:

```
try:
    outer() # !!! raises Exception
except Exception as e:
    print 'Got an error:', e
    raise Exception('error when calling outer()') # this is bad form
```

But this is not a good habit to get into. Note that our new exception loses the traceback information that told us exactly where the error occurred. That's bad.

At the same time, code analogous to the above can be very useful for providing context. A traceback might show you that there was an error opening a file, but it might not tell you *which* file caused the error. A better pattern in these situations is:

```
try:
    outer() # !!! raise exception
except Exception as e:
    print 'error when calling outer()'
    raise # this is better form
```

This provides the context but preserves the full traceback. This approach is not ideal since the extra context does not get stored with the Exception instance, though. Python 3 provides much better tools for this kind of context tracking.

Exceptions are OK

It is also tempting to write code that avoids *causing* environmental errors. This is often actually a bad idea!

For instance, say that you are writing a program that reads a file, and the user gives you the file's name. They might type the name wrong, giving you the name of a nonexistent file. This will lead to an IOError:

```
import os.path

file_name = '/not/a/file.txt'

open(file_name) # !!! raises IOError
```

To avoid this, you might want to use the os.path.exists() function to validate that the file exists before trying to open it:

```
if os.path.exists(file_name):
    f = open(file_name)
else:
    print 'error: file', file_name, 'does not exist'
```

This is actually *less correct* than the prior code. Why? Because a tiny slice of time passes between when you check that the file exists, and when you try to open it. During that time, another program could delete the file, and then your open() would fail in the very way you were hoping not to have to deal with! That's a very improbable turn of events, of course, but it is not impossible.

This problem is a classic example of a sort of bug called a *race condition*. It is yet another example of mutability causing problems. Your computer's filesystem is basically a mutable, globally shared data structure, and this race condition arises because *other programs* can mutate it behind our backs at any time.

In these cases the *only* truly robust approach is to just go ahead and open the file. Unfortunately, to be truly careful we have to add one more wrinkle: an IOError from open() could signify a variety of problems, so we have to check that its error code truly means "file not found." Checking this takes some gymnastics in Python 2.

```
try:
    f = open(file_name)
except IOError as e:
    import errno
    if e.errno != errno.ENOENT: # is e *not* a file-not-found error?
        raise # e is some other error that we can't handle, so: reraise
    print 'error: file', file_name, 'does not exist'
```

In yet another improvement, Python 3 adds a special FileNotFoundError type that makes this particular logic much easier to implement.

Other Resources

Many, many people have written many, many books aimed at beginning programmers. Many of them are about Python and will be absolutely relevant to you. Here are a few that you might want to check out.

How to Think Like a Computer Scientist: Learning with Python, by Allen Downey, Jeffrey Elkner, and Chris Meyers. A 280-page textbook freely downloadable from: `http://www.greenteapress.com/thinkpython/thinkCSpy/thinkCSpy.pdf`. The book's Appendix C contains its own list of recommended readings and resources.

A Gentle Introduction to Programming Using Python. A freely-available four-week online course from MIT, patterned after *How to Think Like a Computer Scientist*. The course homepage is at this shortened Web address: `http://bit.ly/2d9V62A`.

A lot of people fail to appreciate the value of just sitting down and *reading the detailed specification* of a language. A good spec is like a good math textbook — dense and dry, but brimming with insights for the engaged reader. The Python 2 spec is here: `https://docs.python.org/2/reference/`.

The other thing that people fail to appreciate is the value of *reading code*. In a course teaching a human language like French, you'll probably read *at least* ten times as much as you write. Learning a programming language should be the same! Read the code behind the modules you use, especially large, mature ones like AstroPy: `https://github.com/astropy/astropy/`.

The for-profit company Codecademy offers a popular, free Python course at `https://www.codecademy.com/learn/python`. It has you type Python code live into your browser, like the Jupyter notebooks that we use.

Google has a Python class for "people with a little bit of programming experience" that may be found online here: `https://developers.google.com/edu/python/`. It includes video lectures and exercises.

The Software Carpentry project aims to teach programming to scientists who might not have a strong computing background. Their lessons are intended to be conducted live, but the materials are available online at `http://swcarpentry.github.io/python-novice-inflammation/`. That particular lesson is intended to take about five hours to finish. Please note that the lesson targets *version 3* of Python, while we are targeting version 2, which is similar but incompatible.

The website StackOverflow probably already contains the answer to any Python question you might ever ask. Its section of Python-related questions may be found at `https://stackoverflow.com/questions/tagged/python`.

Finally, if you go ahead and do a web search for "how to learn Python," you will get an enormous list of resources. Many of them are from companies trying to make a few bucks off of the many people who want to learn how to code, and not all of their offerings will be brilliant. But there are surely gems out there that we haven't listed above.