Part 1: Introduction to Computer Programming and Python

Andrew D. Wickert

February 1, 2017

1 Why do we program computers?

In order to analyze data, run simulations, check the logical consistency of our ideas, and compare models to the real world, we need numbers. Working with numbers requires quite a lot of repetitive calculation. Doing these calculations ourselves is boring, miserable, and saps our very humanity. Fortunately, computers are excellent at doing repeated tasks. Automating a computer to do our "dirty work" is what programming is all about. In order to send instructions to the computer, we need to write in a language that obeys rules of unambiguous formal logic. This is a computer programming language.

Over this course, there will be many struggles in making this communication between you and the computer work. Therefore, it is important to remember why we program computers: these reasons include to make our lives easier and more pleasant, to accomplish more than we could without the assistance of a computer, to have a perfectly logical companion who can inform us about the internal consistency of our ideas, and to build a numerical laboratory in which we can develop virtual experiments. If it seems that computers are making our lives harder rather than easier, it is time to take a step back and a deep breath, and evaluate what we are doing and why we are doing it.

2 How we program computers

Computer programming languages are a set of commands that we write in human-readable text that eventually are turned into machine-readable binary. This can happen via one of two means:

- In a **compiled language**, we use a program called a "compiler" to translate what we have written into machine language. In the past, this was done by hand – so you can thank your lucky stars for all of the dedicated engineers, mathematicians, programmers, and technicians who have made these compilers

that literally tell a computer how to program itself based on your commands to it!

- In an **interpreted language** (or "scripting language"), your commands are not compiled into byte code (machine language) *en masse*. Rather, they all reference pre-compiled functions. Because of this, you don't have to re-compile the program every time you change it and your coding can be very flexible, but these programs often run somewhat slower than compiled programs. This is the trade-off: ease of programming for you costs computational efficiency in this case.

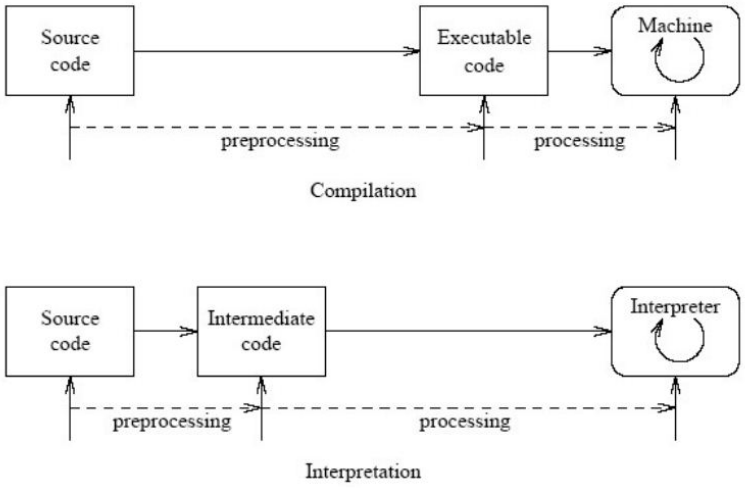

Figure 1: Compiltered languages are pre-processed (i.e. compiled) for efficiency for the computer (but not the human!). Interpreted languages are processed at runtime – one less step for developers (i.e. humans), but more effort for the computer. Image from Aniket Thakur at http://opensourceforgeeks.blogspot.com/2013/03/difference-between-compiler-interpreter.html.

3 Python and the major programming languages

In this course, we will learn Python. It is an interpreted, multi-paradigm, multi-purpose, and easy-to-read open-source language. These mean that:

- **Interpreted:** You don't have to compile the code every time you make a change. This helps speed code development and your learning (critical in this course!) at the cost of some computational efficiency (*not* critical in this course).

	Compiled	**Interpreted**
Open-source	Fortran C/C++ Java	Python R Octave (Matlab clone)
Proprietary	C#	Matlab IDL

Figure 2: Open-source and proprietary, interpreted and compiled, programming languages. For open-access and ease-of-use, we will be using Python, which is interpreted and open-source.

- **Multi-paradigm:** This means that the language can be written as an **imperative** language, with a sequence of command that are executed in some defined order, or as an **object-oriented language**, in which "objects"—clusters of related variables and functions: think perhaps of a volume of soil that has a temperature and thermal conductivity (variables) and can conduct heat (a function)—are manipulated.
- **Multi-purpose:** Python cam be used for numerical modeling, data analysis, web development, interfacing with peripheral devices, automating tasks in GIS, and much more.
- **Easy-to-read:** As programming languages go, Python can be "read" fairly well by someone who has no experience with it.
- **Open-source:** Python is freely available and is updated and expanded by a dedicated community of volunteers. Therefore, learning how to use it will not tie your skills to any software package that must be purchased.

Many of these reasons highlight why I have chosen to work with Python for this course. However, it is important to go over a number of the other major programming languages used in the geosciences, at very least such that their names are familiar to you and you will not feel like you are starting from nothing if you have to program in one of them someday. These are:

C, C++, and Fortran (f77 and f95) are the most commonly-used compiled programming langauges for scientific computing. These are the true "heavy lifters" in computational science. Fortunately, Python is written in C (giving it a native interface with C code), and a compiler called "f2py" allows Fortran code to be compiled in such a way that Python can talk to it. Java is occasionally used, but is less developed in the numerical realm.

Many of your colleagues in the general geosciences use Matlab, which is a very similar language, except that it is more focused on matrix operations, data analysis, and scientific computing... and that it is closed-source, meaning that you cannot see how it actually works. Furthermore, you need to purchase a license to use it. For those of you who are familiar with Matlab already, I would suggest the Mathesaurus

entry "Numpy for Matlab Users" (Numpy is the numerical package for Python) at http://mathesaurus.sourceforge.net/matlab-numpy.html.

IDL is commonly used by atmospheric scientists as well as the community involved with the ENVI GIS package. However, the other major GIS packages, both proprietary (ArcGIS) and open-source (GRASS GIS and QGIS) use Python as their interface language.

R is a programming language designed to perform statistical analyses. It excels in this field. However, outside of this scope, it is fairly limited, which is why we won't learn it in this course. After learning Python, learning R should be straightforward.

The other reasons to teach you Python are that it has functions that streamline file input and output and can connect easily to programs written in other languages, including compiled langauges, making it an effective language to "glue" together pieces of code written in highly efficient (but clunkier to manipulate) C or Fortran code. It also produces excellent graphical outputs. In short, what I am writing that I have chosen Python after thinking long and hard about the decision and using it for an extended period of time myself, and I think that learning it will be a worthwhile use of your time.

4 Practical interlude: downloading and installing

4.1 Python interpreter and development environment

Now we know that we can write code in Python that can be interpreted into a machine language. Great! So how does that happen?

With an interpreter, of course! You must download and install one onto your computer. Linux/*nix and Mac computers come pre-loaded with a basic Python interpreter, but we will need more packages than that, and the Mac version of the Python interpreter is often not up-to-date. Windows machines generally do not come with Python by default.

While the interpreter changes human-readable text into machine language, it does not help humans produce that human-readable text. This is where a development environment comes in. This is something that will format the text for us to help us follow the flow of the code, automatically complete expressions, and more. There are a two main options to do this:

- Command prompt and text editor. In each of these cases, I recommend **ipython** as the program to execute the Python code at the command line interface (= command prompt). Some examples of combinations of these are:
  - Windows: Cygwin and notepad++
  - Linux: Terminal and gedit
  - Mac: Terminal and textwrangler
- Spyder: a Matlab-like Integrated Development Environment (IDE)

Shell and text editor: gedit/notepad++/textwrangler and ipython or Spyder
The Python packages that we will want at the outset of this course are:

- The most recent Python 2.*.* (not Python 3.* – this is a fairly different take on Python that is not connected to the scientific programming packages)
- NumPy (**Num**erical **Py**thon): a package to handle arrays, matrix operations, and more
- SciPy (**Sci**entific **Py**thon): a package that contains special functions, interpolation techniques, linear algebra solvers, and other tools that are useful for scientific programming
- Matplotlib: a plotting library with an extensive range of applications (see http://matplotlib.org/gallery.html); we @TODO: will use the map-plotting extensions to matplotlib later in this course.

4.1.1 Linux

Use your package manager. For Debian/Ubuntu, type at the command line:

```
# Basic packages
sudo apt-get install \
python python-numpy python-scipy \
python-setuptools python-matplotlib

# pip (recommended for automatic installs via setuptools)
sudo apt-get install python-pip

# iPython console -- very useful (optional)
sudo apt-get install ipython

# Sypder IDE (I don't personally use it but many others like it: optional)
sudo apt-get install spyder
```

(The package "python" should be installed by default[1], but is included in this list for completeness.)

4.1.2 Windows

Download **python(x,y)** (https://code.google.com/p/pythonxy/wiki/Downloads) or another full-featured distribution such as **Anaconda** (https://store.continuum.io/cshop/anaconda/). These and several others also contain the required packages (including the numerical libraries), the iPython console, and the Spyder IDE; **Spyder**

[1] Once I was trying to change which version of Python I was using, and without thinking about it (or knowing nearly so much as I do now), I simply typed "sudo apt-get remove python" and entered my password and pressed enter. However, had I looked at the screen to see the list of packages that would also be removed, I would have realized that the list of core functions of my computer that depend on the Python interpreter was enormous. Long story short: had to reboot to a command-line terminal to "apt-get install" and put all of the core software back on my computer, just so I could boot to the desktop!

(https://code.google.com/p/spyderlib/) is a nice IDE that will provide a familiar-looking interface for users accustomed to Matlab.

> **Important: those of you with a pre-existing ArcGIS install that comes with its own (albeit more limited) version of Python, you sh Python(x,y) you should install Anaconda. Python(x,y) will not work.** It should be possible to link Arc to your new install of Python, but I am not sure if you will need to re-install Arc to make this work – this would be recommended if you would like to program with ArcGIS, as you will have a more complete version of Python to use.

4.1.3 Mac

One recommendation is to use a package manager like **homebrew** (http://brew.sh/). With this you can install Python, and then move on to using **pip** (or homebrew) to install the Python modules. A good introduction to this can be found here: http://www.thisisthegreenroom.com/2011/installing-python-numpy-scipy-matplotlib-and-ipython-on-lion. See the **Linux** instructions for the list of packages that you will need; after installing pip, these commands can be substituted as follows, e.g.,

```
# Homebrew
sudo brew install python-numpy
# Pip
pip install numpy
```

Recent efforts to download Python distributions (both **Anaconda** and **Enthought**) have not met with success with both gFlex and GRASS, though **Anaconda** has been tested successfully with Windows. As a result, it should be more successful to keep the Python packages managed better by something like **homebrew** with **pip**.

> **If you do not have Arc and want to do GIS programming, three main options exist.** You may use GRASS GIS (http://grass.osgeo.org/download/, QGIS https://www.qgis.org/en/site/forusers/download.html), or simply use the GDAL an OGR libraries with Python.

5 Basic Python programming (and programming concepts in general)

The following sections provide a brief introduction to programming in Python. A more exhaustive treatment can be found in the textbook, *Think Python: How to Think Like a Computer Scientist* (PDF and HTML available at http://www.greenteapress.com/thinkpython/).

5.1 So you want to write a program

By now, you know that a program is a set of user-readable text that the computer can understand and use to drive its actions. Even though at this point you might not be familiar with a program, immersion is a way of learning, so here is a simple program that we will use for examples in the next several sections.

```
#! /usr/bin/env python

# firstExample.py
#
# Written by ADW on 05 May 2015
# while finishing these notes at the last minute
# (as usual) to teach class tomorrow
#
# This program will continue adding numbers to a starting
# value until the number reaches 50.
# It then multiplies this by 2 three times.

import sys

counter = 0 # value to be modified

while counter < 50:
  counter += 1

if counter == 50:
  # It should at this point!
  print "50!"
else:
  print "Error!"

for i in range(3):
  # i = [0, 1, 2] (0-indexing)
  counter *= 10

if counter == 50 * 10**3:
  print "Checks out -- done!"
  print "Final counter =", counter
else:
  sys.exit("Fail!")

print "Now let's divide counter by 25001, just over half"
print counter/25001

print "Now let's divide counter by 25001., just over half"
print counter/25001.
```

Before going any further, I will make a note of a couple of pieces of this code here:

```
#! /usr/bin/env python
```

This is the "shebang". It is important on Unix-like operating systems (e.g., Linux, Mac) to tell the computer which interpreter to use.

Next,

```
#! /usr/bin/env python

# firstExample.py
#
```

```
# Written by ADW on 05 May 2015
# while finishing these notes at the last minute
# (as usual) to teach class tomorrow
#
# This program will continue adding numbers to a starting
# value until the number reaches 50.
# It then multiplies this by 2 three times.
```

The pound sign (hashtag!), #, denotes comments. These can be used throughout your code. And they should be used! Many hours of work writing code are often rendered useless when someone else tries to use it only to find no comments among a pile of expressions in a computer language. Even worse, many people have been known to not recognize their own code 6 months or 1 year out unless they have written good comments. Get in a habit of writing many good comments early, and you will be better-off for it.

> For comments, see Downey, p. 19

In comments, I typically write the name of the code (in case I change it later – to remember), the date that I started to write it (and perhaps other dates of significant code development), and my name or initials. I also write a short note about what the program does.

After this, there is "import" call.

```
import sys
```

"import" brings in a specific package of extra functions in Python. This is called a "module". Eventually, when you write programs, you may create your own modules that are related to the problems that you like to solve. Such a module may relate to, for example, solutions for stress inside the Earth. The module "sys", as its name suggests, connects to the computer system. Towards the end of the code,

```
  sys.exit("Fail!")
```

causes the program to crash if it is executed. This can be useful to provide an useful message to the user if something is badly wrong. Modules have many helpful functions, most of which do not abruptly terminate the operation!

5.2 Variables and basic types

> See Downey, pp. 13–15 And for more on strings, see Downey, Chapters 8 and 9

Variables can be things like:

```
a = 152 # Integer (int)
b = 16.2 # "Floating point" number (float)
c = 'Ada Lovelace' # Character string (str)
```

```
c2 = "Lord Byron" # double quotes also define a string (str)
d = True # Boolean (bool)
```

Each of these four examples with distinct letters is a different variable **type**, as noted in the comment (# sign) after each entry. They behave differently. As might be expected, integers and floats play with each other better than do strings. The other important piece in the above code is the use of = as the *assignment operator*. It assigns the value to that particular variable letter. This is quite powerful, as variables can allow a code to be flexible and work with a wide range of input values – or even input types!

So now, in our above code, we can see that we have defined the variable "counter", and that it is an integer.

```

counter = 0 # value to be modified
```

What if we had wanted to make it a floating-point number? Well, we simply would put a decimal point after it.

```
counter2 = 0.
```

This is important when you want to take advantage of the ways in which integers and floating point numbers differ. See Section 6.5 for a common example of this.

Types apply to all values in a code, whether they are assigned to variables or just added "on the fly" to the code.

Boolean values can be `True` or `False`. These are often used for "flow control", or telling your code how to step from one step to another. This is explained further in section 6.1.

Additional types for individual values include `Null` for non-values and `inf` for infinite values.

6 Letters and numbers: A look from the computer's point of view

Now that we've talked a bit about different variable types, I might explain why they are important. As we all know, computers record data in binary: 0 and 1. How does this turn into letters and numbers? Well, let's think a bit about how computers store data, because we all are familiar with that from saving files, buying hard drives, etc.

- 1 bit can have two values: 0/OFF/FALSE/LOW and 1/ON/TRUE/HIGH. This is the fundamental unit of computation.
- 8 bits in one byte
- 1024 bytes in one kilobyte (KiB), though note that often a metric convention of 1000 is used for this and later conversions!

- 1024 KiB in one megabyte (MiB)
- ... and so on.

From this we already know that a bool, which can be True or False, is 1 bit! This is really memory-efficient.

Now let's look at the second value. What is special about 8 bits being in one byte? Well, we know that:

$$2^8 = 256 \tag{1}$$

Do you remember computers with 256-color displays? Exactly! They were using one byte per pixel to describe the color on the screen.

When using numbers, we like to include zero. So instead of having a list of values that is [1, 2, 3, ..., 255, 256], we have one that is [0, ..., 255]. Does this sound familiar? Like, for example, a color-picker in a photo-editing or illustrating application? We call this an **unsigned 8-bit integer**. "Unsigned" means that it does not have anything to say whether it is positive or negative.

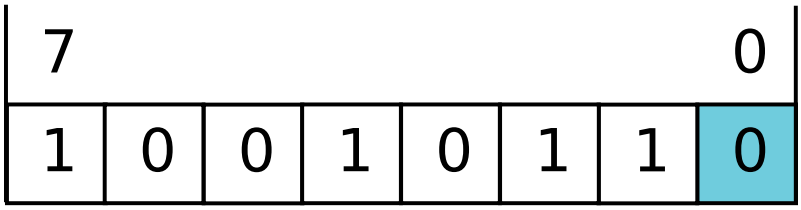

Figure 3: Graphically, a byte can be displayed as a set of 1's and 0's. This shows an example at which the bit numbering starts at the right and goes left, but the more important piece to notice at this stage is just how the different blocks of 1's and 0's can combine to make a larger number, that then can represent something inside the computer.

What happens if we want negative numbers? Well, we can choose the sign with one of our bits. But then we have only seven to give the values. Thus, we get a range of -128 – $+127$ for **signed 8-bit integer**.

8-bit inetegers are also helpful for printing letters. In ASCII, the American Standard Code for Information Interchange, which was first established in 1963, letters, numbers, symbols, and special "control" characters are designated by a seven-bit number with the last bit for optional error-checking. (The last bit was optional so it could be dropped to save costs when transmitting even this small amount of data was a big deal – thus many teletype machines used 7 bits! This chart was published in February 1972.)

The rules for bytes can then be extrapolated to 16-, 32-, and 64-bit values. These can generate huge integers!

But what about numbers with a decimal place (float)? These "floating point" numbers store values in scientific notation, with one bit for the sign, some number of bits

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Figure 4: Binary, hexadecimal, and decimal conversion table. From Nitin Kumar, http://www.easycppcodes.com/conversion/program-to-convert-hexadecimal-to-binary-c/

for the exponent, and some bits for the value that is then multiplied by 10 to the specified power. You might imagine that this can create rounding errors, as there are infinite numbers but only finite bits in such a representation. And indeed it does: see *What Every Computer Scientist Should Know About Floating-Point Arithmetic* (http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html) to learn more.

USASCII code chart

Bits b_4 b_3 b_2 b_1	Column → / Row ↓	b_7 b_6 b_5 = 0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
		0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

Figure 5: ASCII is probably the most simple and widely-used method of encoding letters and numbers. It was originally developed in the 1960's for telephone operators for teletype machines, and was based on telegraph codes. Ever wonder why we can't have umlauts in email addresses? Or why so many people use all flat lines (hyphen, minus sign, and dashes) as the same thing? It's ASCII's fault! But there is only so much one can do with 128 characters.

Python handles changes between these types, and the amount of memory required to store them, pretty much automatically. But not all programming languages do, and this is important to know for general programming literacy!

For a really nice review of how computers actually work, check out the first answer at http://programmers.stackexchange.com/questions/81624/how-do-computers-work. Also, you'll become qutie good friends with StackExchange/StackOverflow, as you can do a Google search on literally nearly any problem you may encounter while programming, find someone who already asked the question there, and find the best answer that typically explains what is going on really well. Much of my programming knowledge comes from Internet help, either by directly answering my question, or by giving me some ideas to help reason through it on my own!

6.1 Loops: for, while

Loops repeat execution of a piece of code while a particular condition is met. In the above code:

```
while counter < 50:
  counter += 1
```

changes "counter" by adding 1 to it so long as "counter" is less than 50. Pretty simple right? Well, what if we changed it to this?

```

trouble = True
while trouble == True:
  counter += 1
```

"==" is the logical equals expression; while "=" assigns values, "==" checks for equivalency. Would the prgram ever end?

> **The power of CTRL+C (Control+C, Strg+C):** It doesn't just copy text! It forcibly terminates your programs. Very useful for stopping mistakes from getting out of hand. Like infinite loops.

Would this code create an infinite loop?

```

trouble = True
while trouble:
  counter += 1
```

Here, we are missing the `== True` after `Trouble`. However, this still works. Why is that? Well, the code simply evaluates whether the statement is nonzero when trying to decide whether to continue the `while` loop. Consider `while counter < 50` above, for example. An additional helpful piece of information: Boolean `True` is 1 (and `False` is 0).

Another type of loop is a `for` loop, in which the loop runs until you run out of values in a list. For example:

```
for i in range(3):
  # i = [0, 1, 2] (0-indexing)
  counter *= 10
```

`range` is a special Python command that creates a set of values that goes up to (but does not exceed) the value inside the parentheses. So, as the comment notes, `range(3)` produces `[0, 1, 2]`. These square brackets indicate that this is a `list`, a variable type that is a container for other values; this is discussed in section 6.6.

"Nesting" is the process of combining multiple for loops. You may use the below space to write an example of nested loops and to write some thoughts about why you might want to do this.

6.2 Logical operators and flow control: if, else if, else, and, or, not

> See Chapter 5 in Downey for a much more thorough introduction

Two portions of the code above use "if" and "else". This provides "flow control" for the code by giving causing events to be based on some set of inputs. In this case, it is fairly simple, but you can imagine much more complex trees of nested if/else loops.

```
if counter == 50:
  # It should at this point!
  print "50!"
else:
  print "Error!"
```

```
if counter == 50 * 10**3:
  print "Checks out -- done!"
  print "Final counter =", counter
else:
  sys.exit("Fail!")
```

These logical operators can be nested together with loops using `for` and `while` to generate more complex sets of expressions. For example, you could say that while one value is true, if another value is less than five, you multiply it by five. While there would probably be no reason to ever execute this simple example, it is possible to imagine another case – in which each pass through a `for` loop was one step in time, and then you would change the position of rock units `if` a condition for fault slip has been met. And then, for example, you would reset the fault-slip condition so it could accumulate strain before the next earthquake. While fault slip and interseismic strain are complex topics, not accurately represented in such a simple model, it is possible to imagine this approach as a way to accomplish many tasks when programming.

If you want to string together multiple conditions for something there are two options. Either you can create multiple nested `if`/`else` loops, or you can (usually more efficiently) use statements like `and`, `or`, and `not` to combine conditions. You can think of `and` like multiplication: if one is `True` (1) and the other is `False` (0), then the outcome is `False`. Likewise, `or` is a bit like addition in that so long as at least one condition is `True` (1), then the sum will be > 0. This is *like* addition in that the result of many `or` statements will never be > 1. I mention this because it is also possible to have multiple variables with mathematical operators, such as addition and multiplication, that tell the computer whether or not to execute an if-statement. An important note is that negative values are also `False`. This can be quite useful when writing numerical models in which the sign of a value, or its being nonzero, determines what the model should do. `not` returns `True` when the statement that it is evaluating is `False`.

6.3 Exceptions: try and except

You may use `try` and `except` in a way that is similar to the logical operators. This takes an action that you knnow might fail in the `try` portion, and instead of crashing the code, "catches" it and moves on to the `except` portion.

See pages 162-163 (Downey)

6.4 "Print" and other statements

See Downey, p. 15

print sends the output to the screen. Try it! You can print several things on the same line by separating them with commans, as is accomplished in the second line of the above code, reproduced here:

```
print "Checks out -- done!"
print "Final counter =", counter
```

6.5 Operators: Mathematical and String

See Downey, p. 16, 18

Operator	Numbers	Strings (and lists)	Booleans
+	Addition $1.5 + 3 \rightarrow 4$	Concatenation 'cow' + 's' → 'cows' [1, 2] + ['cow'] → [1, 2, 'cow']	OR bool(True + False) → True
−	Subtraction $1.5 - 3 \rightarrow -1.5$		NOT (XOR) bool(True − False) → True bool(False − True) → True
*	Multiplication $1.5 - 3 \rightarrow -1.5$	Repetition 'cow' * 2 → 'cowcow' [1, 2] * 2 → [1, 2, 1, 2]	AND bool(True * False) → False
/	Division Division		
**	Exponentiation $3 * *2 \rightarrow 9$		
%	Modulus (remainder) $15\%4 \rightarrow 3$		

Table 1: Common operators.

Operators work on numbers how you would expect; please see Downey about how they work with strings (or just experiment!). Also, note this code from above with regard to variable type:

```
print "Now let's divide counter by 25001, just over half"
print counter/25001

print "Now let's divide counter by 25001., just over half"
print counter/25001.
```

When an integer is divided by another integer, the code always rounds down. When there is a floating point number involved, it then gives a more exact solution.

6.6 Types that combine several items, 1: lists, (dictionaries), tuples

> See Downey, Chapters 10: Lists, 11: Dictionaries, and 12: Tuples

int, float, bool, and str are four common possible **types** that single values may have. But these single values may be nested within structures with more complexity. One example of these is a list. Another is a tuple. They differ primarily in that items inside lists may be changed, while those inside tuples may not be. Lists and tuples are indexed starting with 0, and these indices are placed within brackets when selecting values. They may contain a mix of variable types. This is a lot to take in in a fairly dense paragraph, so I will really point you to three (!) book chapters, and use the following code example to show some important pieces in working with them:

This provides an example of usage

```
l = [1, 2, 3, 4, 5] # list
t = (1, 2, 3, 4)    # tuple

# You index starting at 0
print l[0] # it is 1!

# Indices that are negative give you values starting from the end!
print t[-1]

# You use a colon to indicate multiple indices
# This is INCLUSIVE of the first value but EXCLUSIVE of the second value!
print l[0:2] # you get 0 and 1
# If you do not have a value before or after the colon, it is assumed
# that you want all values before or after the one that you have listed
print l[:3]
print t[-2:]
print t[:]
print t, "Is the same as the printout right above!"

# Tuples and lists may contain multiple variable types
l2 = [1, 2.16, True, 'Cow']
t2 = [1, 5., False, 'Duck']

# Tuples are immutable: you may not change their values. But lists are not.
print "In a list, a Cow can change to a Duck"
print l2
print "Abra cadabra!"
l2[-1] = 'Duck'
print l2

print "But in a Tuple, we can't change 'Duck' to 'Cow' -- or anything!"
print t2
print "Abra cadabra!"
try:
  t2[-1] = 'Cow'
except:
  print 'Magic trick failed!' # Single and double quotes both define strings

# Lists can be concatenated with "+", just like strings
print l + l2

# And can be repeated with "*", just like strings.
print l*3
```

```
44
# If you have a value that changes with time
46 # and you want to record how it changes,
# you can use the list "Append" command to expand a list
48 dogs_at_park = [] # empty list
for i in range(5):
50   j = i**2
   dogs_at_park.append(j)
52 print "The number of dogs at the park is growing quadratically!"

54 # When you are just using values, variables capture the value
# (i.e. make a copy of it)
56 # and that stays the same no matter what happens to the original
a = 5
58 b = a
a -= 1
60 print 'a = ', a, '; b =',  b, ": they are different if you change a scalar"

62 # But varibales just point to a reference to containers for variables like
     lists.
# Hence, values are shared
64 a = [5]
b = a
66 a[0] -= 1
print 'a =', a, '; b =', b
68 print "for lists, a and b change in tandem!"
```

Dictionaries are very useful, but will not be discussed here, beyond noting that they connect keyword–value pairs. Please read Chapter 11 in Downey's book to learn more about them.

6.7 Python modules

I mentioned Python modules briefy above; a module is a set of functions and/or variables that can be useful for a specific task. Python comes with a number of standard modules that assist with certain tasks. These include:

- os: Operating system-related tools
- sys: Access to some Python interpreter objects (objects are containers of functions and/or variables, and will be introduced later)
- glob: Pattern-matching, e.g., for listing files in directories
- re: Regular expressions (regex) for advanced pattern-matching; often helps to format messy data files (for a description of regex, see https://en.wikipedia.org/wiki/Regular_expression)

Other useful modules that you should have installed (and you should now if you haven't) include:

- scipy: Special functions, solvers, mathematical tools
- numpy: Arrays and matrices, more mathematical tools (see 6.8 below for more information)

- matplotlib: Plotting and graphics

Modules are imported at the beginning of your script as follows:

```
import os # Import it just as "os"

print "our current directory is:"
print os.getcwd()

from os import path # imports one part of a module

print "We can check if the current working directory is a directory,"
print "which of course it is."
print path.isdir(os.getcwd())
print "And we can do it with "os" alone, because "path" is part of os.
print os.path.isdir(os.getcwd())

import numpy as np # Use "np." instead of "numpy." before commands

from matplotlib import pyplot as plt # combination -- pyplot is plt, but base
                                     # matplotlib is not imported
```

6.8 Numpy, namespaces, and numpy arrays

"Numpy", which stands for "Numerical Python", is a module that we will use quite often. To import it at the start of a script, we write:

```
import numpy as np
```

The "as np" portion of this means that we have only to write "np." before any numpy command or variable instead of "numpy.". For example:

```
import numpy as np

print np.pi
print np.e
```

One could also write:

```
from numpy import *

print pi
print e
```

and forego all of the "np.*" alltogether. However, let's say that you forget that "e" is a special character and decide you want it to be the number of electroncs in your sample.

```
from numpy import *

e = 5128

print pi
```

```
print e
print "Oh, no! We lost Euler"
```

This is an introduction to a very important concept, which is protecting **namespaces** so you do not accidentally ovewrite functions or variables.

To learn about Numpy, there are major resources.

1. There is a really nice tutorial at http://wiki.scipy.org/Tentative_NumPy_Tutorial
2. For those of you familiar with Matlab, there is a chart of equivalent functions, operators, etc. at http://mathesaurus.sourceforge.net/matlab-numpy.html.

You should become familiar with numpy arrays, as they will become the basis for much of what we do in this class!

6.9 Plotting values

Plotting doesn't exactly fall into "standard" programming – but hey, you'll be doing scientific programming, and it is important! The great thing about being able to write computer scripts that generate plots is that if your data change or if you want to do something different, you don't have to start from nothing. Plus, those generated by Python's module, Matplotlib, look so much nicer than any default Excel plot.

To learn how to use Matplotlib, see the online tutorial at http://matplotlib.org/users/pyplot_tutorial.html.

7 Writing programs

All right! You finished the basics. Or you skipped them. Whatever, either way you're ready to start writing computer programs. That is great.

First-off, you should know that there are two general programming paradigms.

- **Imperative** programs list a set of commands in order. These may include **functions**, pieces of code that take an input and return an output
- **Object-oriented** programs combine variables and functions into "objects", which are then instantiated (created as a unique copy with specific attributes) and executed (run) by other code.

We are going to use the example of a horizontally-thrown snowball to learn about how to write code like this. It outputs its maximum distance and horizontal impact momentum.

7.1 Imperative

7.1.1 Without functions

```
#! /usr/bin/env python

# Snowball: imperative -- and assuming horizontal release

import numpy as np

name = 'standard snowball'
diameter = 10 # [cm]
density = 450 # [kg / m^3]
release_velocity = 15 # [m/s]
release_height = 1.5 # [m]

# Let's find its horizontal impact momentum when hitting a stopped object
p_horizontal = density * (4/3.) * np.pi * (diameter/2./100.)**3 \
               * release_velocity                                  # [N s]

# Let's see how far it will go before hitting the ground
# Ignoring wind resistance
# z = 0.5 g * t**2
g = 9.8 # [m/s**2]
t = (2 * release_height / g)**0.5
x_flight = t * release_velocity

# Print output
print ""
print name, 'travels', round(x_flight,1), 'meters and strikes its target'
print 'with a momentum of', round(p_horizontal,1), 'Newton-seconds.'
print ""
```

code/Introduction/snowball_imperative.py

7.1.2 With functions

> For more on functions, see Downey, Chapters 3 and 4.

Here and beyond, one can use the functions to simply produce several outputs based on some inputs without copying and pasting the entire code.

```
#! /usr/bin/env python

# Snowball: with functions -- and assuming horizontal release

import numpy as np

# Global constants
g = 9.8 # [m/s**2]

# Values in functions that are not output belong to those functions
# and cannot be accessed from the outside.
# Functions must be listed before they are accessed in the code.
def horizontal_momentum(rho, D, rv):
  """
  Let's find its horizontal impact momentum when hitting a stopped object
  """
  return rho * (4/3.) * np.pi * (D/2./100.)**3 * rv # [N s]
```

```
18
19 def travel_distance(rv, rh):
20     """
21     Let's see how far it will go before hitting the ground
22     Ignoring wind resistance
23     I have renamed the internal variables to better show how they are passed to
24     functions
25     """
26     # z = 0.5 g * t**2
27     t = (2 * rh / g)**0.5
28     return t * rv
29
30 def print_output(n, x, p):
31     """
32     Print output
33     """
34     print n, 'travels', round(x,1), 'meters and strikes its target'
35     print 'with a momentum of', round(p,1), 'Newton-seconds.'
36
37
38 # First run
39 name = 'standard snowball'
40 diameter = 10 # [cm]
41 density = 450 # [kg / m^3]
42 release_velocity = 15 # [m/s]
43 release_height = 1.5 # [m]
44 x_flight = travel_distance(rv=release_velocity, rh=release_height)
45 p_horizontal = horizontal_momentum(rho=density, D=diameter, rv=
       release_velocity)
46 print ""
47 print "Run 1:"
48 print_output(n=name, x=x_flight, p=p_horizontal)
49 print ""
50
51 # Second run -- directly pass values
52 print "Run 2:"
53 x_flight = travel_distance(rv=5, rh=1.5)
54 p_horizontal = horizontal_momentum(rho=250, D=15, rv=5)
55 print_output(n='fluff ball', x=x_flight, p=p_horizontal)
56 print ""
57
58 # Third run -- pass values just in right order (no keywords)
59 print "Run 3:"
60 x_flight = travel_distance(15, 2.3)
61 p_horizontal = horizontal_momentum(450, 10, 15)
62 print_output('high release', x_flight, p_horizontal)
63 print ""
64
65 # Fourth run -- pass everything straight to print_output,
66 #               including use of other functions
67 print "Run 4:"
68 print_output('wicked tiny iceball', travel_distance(35, 1.5), \
69              horizontal_momentum(917, 7, 35) )
70 print ""
```

code/Introduction/snowball_imperative_functions.py

7.1.3 Placing these functions in a module by themselves

> For more on custom modules, see Downey, pp. 166–167

Module:

```
import numpy as np

# Global constants
g = 9.8 # [m/s**2]

# Values in functions that are not output belong to those functions
# and cannot be accessed from the outside
def horizontal_momentum(rho, D):
  """
  Let's find its horizontal impact momentum when hitting a stopped object
  """
  return rho * (4/3.) * np.pi * (D/2./100.)**3 # [N]

def travel_distance(rv, rh):
  """
  Let's see how far it will go before hitting the ground
  Ignoring wind resistance
  I have renamed the internal variables to better show how they are passed to
  functions
  """
  # z = 0.5 g * t**2
  t = (2 * rh / g)**0.5
  return t * rv

def print_output(n, x, p):
  """
  Print output
  """
  print n, 'travels', round(x,1), 'meters and strikes its target'
  print 'with a momentum of', round(p,1), 'Newton-seconds.'
```

code/Introduction/snowball_functions_module.py

Driver:

```
import snowball_functions_module as sb

# First run
name = 'standard snowball'
diameter = 10 # [cm]
density = 450 # [kg / m^3]
release_velocity = 15 # [m/s]
release_height = 1.5 # [m]
x_flight = sb.travel_distance(rv=release_velocity, rh=release_height)
p_horizontal = sb.horizontal_momentum(rho=density, D=diameter)
print ""
print "Run 1:"
sb.print_output(n=name, x=x_flight, p=p_horizontal)
print ""

# Second run -- directly pass values
print "Run 2:"
x_flight = sb.travel_distance(rv=5, rh=1.5)
p_horizontal = sb.horizontal_momentum(rho=250, D=15)
sb.print_output(n='fluff ball', x=x_flight, p=p_horizontal)
print ""

# Third run -- pass values just in right order (no keywords)
print "Run 3:"
x_flight = sb.travel_distance(15, 2.3)
p_horizontal = sb.horizontal_momentum(450, 10)
sb.print_output('high release', x_flight, p_horizontal)
print ""

# Fourth run -- pass everything straight to print_output,
```

```
#                 including use of other functions
print "Run 4:"
sb.print_output('wicked tiny iceball', sb.travel_distance(35, 1.5), \
                sb.horizontal_momentum(917, 7) )
print ""
```

code/Introduction/snowball_functions_module_driver.py

7.2 Object-oriented

> For quite a lot more about objects, see Downey, Chapters 15–18

We could have the class held in the same file or in a separate file. Because you already have experience with things being in different files from the above module import, we will introduce classes in a separate file. Pay attention to the comments as well as the docstrings (the latter are between two lines of """), as they help you to understand how the classes work.

In short, a "class" is a concept – think of it as a generic Lego set sitting on a shelf. There are infinite Lego sets. But when you "instantiate" one, you make it yours, and give it properties. To return to the snowball example, you can give it properties like density and horizontal release velocity that effect the output – the distance it can be thrown (based on ballistics without air drag) and the impact momentum. So your Lego set, your snowball, your whatever – they are your personalized version of this class.

A class is instantiated as follows:

```
import snowball
myfluffball = snowball.snowball()
```

Then you can type `myfluffball.` inside iPython and press "tab" twice to see the contents of the class – variables and functions.

Variables within the class belong to the class, and as default contain `self.` in front of them (within the class) and the name of the instance when used outside of the class itself.

Module with class:

```
import numpy as np

# Global constants
g = 9.8 # [m/s**2]

class Snowball(object):
  """
  snowball physics
  """

  # __init__ is a special function that is run when the class is instantiated.
  # "self" refers to all functions, variables, and anything else held by the
  # class.
```

```
14
   # All of the "=" signs denote default values for the class.
16   def __init__(self, name='default snowball', rho=450, D=10, rv=10, rh=1.5):
       """
18     Set up and solve the snowball physics
       """
20     # make these into class variables.
       # I more want to do it just mostly so the class stores the snowball's
22     # characteristics, but also so I don't have to pass variables explicitly.
       self.name = name
24     self.density = rho
       self.diameter = D
26     self.release_velocity = rv
       self.release_height = rh
28     # And automatically run the first two functions
       # Note that with a class, we can run functions that lie below the current
30     # point
       self.horizontal_momentum()
32     self.travel_distance()

34   # All relevant variables are part of "self" (i.e. the class), so do not
   # need to be passed to each function. It's a help!
36   def horizontal_momentum(self):
       """
38     Let's find its horizontal impact momentum when hitting a stopped object
       """
40     # And we don't need to return anything, because this c
       self.p = self.density * (4/3.) * np.pi * (self.diameter/2./100.)**3 \
42                            * self.release_velocity # [N s]

44   def travel_distance(self):
       """
46     Let's see how far it will go before hitting the ground
       Ignoring wind resistance
48     """
       # z = 0.5 g * t**2
50     t = (2 * self.release_height / g)**0.5
       self.x = t * self.release_velocity
52
   def print_output(self):
54     """
       Print output
56     """
       print self.name, 'travels', round(self.x,1), 'meters and strikes its
       target'
58     print 'with a momentum of', round(self.p,1), 'Newton-seconds.'
```

code/Introduction/snowball.py

Driver:

```
#! /usr/bin/env python
2
# check out how easy this is with a class!
4
import snowball
6
# instantiate
8 boringsnowball = snowball.Snowball() # class has built-in default values

10 # create another instance, with custom values
# keyword--value pairs, if given, do not have to be in order!
12 mysnowball = snowball.Snowball(name='Deadly Ice Boulder', rho=917, \
                                 D=50, rh=1.3, rv=2.)
14
print ""
```

```
16 boringsnowball.print_output()
   print ""
18 mysnowball.print_output()
   print ""
```

code/Introduction/mysnowball.py

This concept becomes quite powerful when it can be applied to something like your Earth-science model or data analysis routine, in that it allows flexible reproducibility of your work.

8 Code philosophy

At this point, you know enough that I want to share a few points that I hold closely when writing computer code:

- Write something you care about.
- Reproducable results – you want to produce something that will work consistently. And sometimes this means that you need to be careful when coding!
- Keep the inputs separate from what runs the code. Having a module with a class that does the heavy lifting is a great way to do this; see the snowball class example, above.
- Generalized: change variables in a central place, don't have to change multiple values (have one master vlaue that is referenced), etc.
- But don't spend so much time making a single-use code perfect that you are wasting time... unless your goal is to learn how to write a really nice code!
- Know when to make a tactical retreat – don't stare at the computer trying random combinations of solutions. Leave, do something else, take a walk, do other work, ask a friend for help. These all help you avoid the computer black hole.

9 Version control

When we are in the lab or in the field, we keep notebooks of important steps along the way so we can track our path through the work and share our work with others. When producing software, we want the same things. Therefore, computer scientists have built "version control" tools. These track changes you have made to your code by leaving bookmarks where you have "committed" the code, and help you to collaborate with others and manage release versions of your code.

We can learn about version-control tools by interfacing with our course's own GitHub site. GitHub, uses "git", pronounced the same as "get". I could write a lot about it here, but it's been done before. So just go to this link and follow the help:

http://git-scm.com/doc

And if you prefer a video to describe some of this, well, this one seems okay:

https://www.youtube.com/watch?v=0fKg7e37bQE

Once you have done this, I challenge you to "clone" a copy of the course notes onto your home computer using the `git clone <url>` command. You can then use `git pull` to continually update your version of all of the course documentation.

You are of course not required to post your code on github or to create an account. But if you do wish to work with git, it is good practice, wheher your are interested in academia or industry. And it can provide me a good way of helping you with your projects by creating a "fork" of what you have done (think of this as a fork in the road by which one set of code takes one path, and one takes another), making some changes, and then requesting to merge my fork back in with yours. (If you do not use git, we can still do this over email.) You can also use git to help each other with your projects. And this is really one main power of git and version-control in general: it streamlines collaboration.

10 An example model: random-walk diffusion

This last example is something we can go through as a class. I challenge you to use the concepts you have learned about writing good code to improve upon this – consider how to make plotting less repetative, bundle the code into sections, and perhaps make it into a set of functions with inputs and outputs – or even a class (object).

```
#! /usr/bin/env python

import numpy as np # Numerical library
from matplotlib import pyplot as plt # Plotting library
import time

# Written for Computational Methods in Earth Sciences class

# Input
ntracers = 1000
stepsize = 1
update_period = 5 # update every this many time steps
plot_period = 50  # Plot every this many time steps
final_time_step = 5000

# Setup
plt.ion() # Turn interactive mode on for plots -- draw as you go.
          # This is replacing the experimental plt.show(block=False)
tracers = np.zeros(ntracers) # Array of 0's
tracer_id = np.arange(ntracers) + 1 # 1--500; can't do this with just range()
                                    # because + with lists is concatenation
                                    # instead, would be range(1, 501)
x = np.arange(np.floor(-2*np.sqrt(final_time_step)), np.ceil(2*np.sqrt(
    final_time_step)))
binwidth = 10 # For histogram

fig = plt.figure(figsize=(6,9)) # Create a figure OBJECT named "fig"
ax1 = fig.add_subplot(211)
ax1.plot(tracers, tracer_id, 'ko', alpha=.25)
ax1.set_xlim(-100, 100)
ax1.set_ylabel('Tracer ID')
ax2 = fig.add_subplot(212) # Add one subplot to it that fills the whole space
```

```
                        # This OBJECT is called "ax2" for "axes object"
ax2.hist(tracers, bins=np.arange(-100, 101, 5)) # Histogram
ax2.set_xlabel('Distance from starting point') # This could also be "plt.xlabel"
                                               # to work on the current figure
plt.ylabel('Frequency of occurrence here') # illustrating other way to do this
# I have set xlim and ylim based on what I know about diffusion
ax2.set_xlim(-2*np.sqrt(final_time_step), 2*np.sqrt(final_time_step))
plt.ylim(0, ntracers*binwidth/30.)
plt.tight_layout() # Automatically c
fig.canvas.draw()
#plt.show(block=False) # Allow script to keep running while plot is shown
time.sleep(0.1) # Time to look at plot

# Run
# "+1" because "range" is exclusive ( [0, 501) = [0, 1, ..., 499, 500] )
for t in range(1, final_time_step + 1):
  # This method wastes a lot of time steps doing nothing. Look at np.arange and
  # think about how this could improve the way we loop through this function.
  if t % update_period == 0:
    tracers += 2*np.random.randint(2, size=ntracers) - 1
  if t % plot_period == 0:
    ax1.clear() # If we want to remove prior time steps
    ax2.clear()
    ax1.set_xlim(-2*np.sqrt(final_time_step), 2*np.sqrt(final_time_step))
    ax1.set_ylabel('Tracer ID')
    ax2.set_xlim(-2*np.sqrt(final_time_step), 2*np.sqrt(final_time_step))
    ax2.set_ylim(0, ntracers*binwidth/30.)
    # Plot again
    ax1.plot(tracers, tracer_id, 'ko', alpha=.25)
    # plt.hist works like ax.hist, just on current axes
    # so if we had multiple axes up, we would have to specify which one to use.
    # (hint: we do in this case)
    # Easier to name them at the start
    ax2.hist(tracers, bins=np.arange(-250, 251, binwidth), color='b') # Histogram
    # Floor division with integer update_period is good for this
    ax2.plot(x, binwidth*ntracers/(2 * np.pi * t/update_period)**.5 * \
             np.exp(-x**2/(2.*(t/update_period))), color='0.', linewidth=2)
    plt.draw()
    #time.sleep(0.1) # Time to look at plot

plt.show()
```

code/Introduction/randomWalk1D.py