Lectures 1–?: Introduction to Computer Programming and Python

Andrew D. Wickert

May 6, 2015

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.

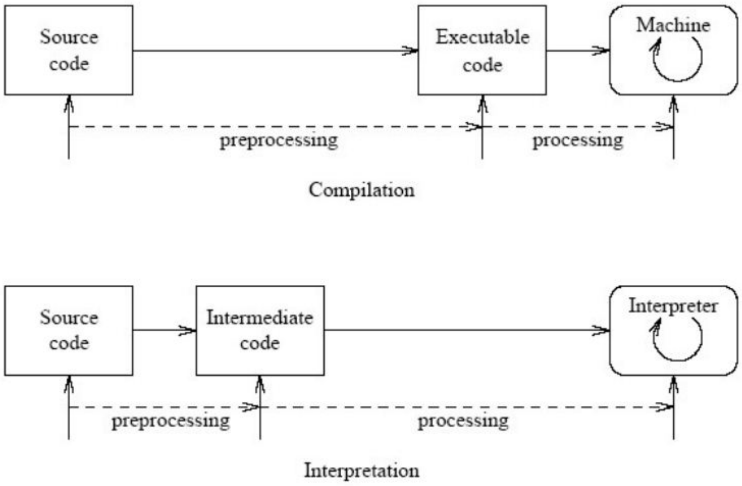


Figure 1: Compiltered languages are pre-processed (i.e. compiled) for efficiency for the computer (but not the human!). Interpreted languages are processed at runtime – one less step for developers (i.e. humans), but more effort for the computer. Image from Aniket Thakur at http://opensourceforgeeks.blogspot.com/2013/03/difference-between-compiler-interpreter.ht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

3 Python and the major programming languages

In this course, we will learn Python. It is an interpreted, multi-paradigm, multipurpose, 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	Python
	C/C++	R
	Java	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**; both of these distributions have been tested successfully with gFlex. Python(x,y), Anaconda, and several others also contain the required packages (including the numerical libraries), the iPython

[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!

console, and the Spyder IDE; **Spyder** (https://code.google.com/p/spyderlib/) is a nice IDE that will provide a familiar-looking interface for users accustomed to Matlab.

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**.

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

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.

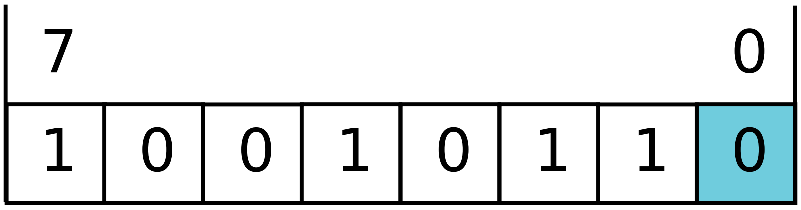


Figure 3: Graphically, a byte can be displayed as a set of 1's and 0's. This shows an example at which the bit numbering starts at the right and goes left, but the more important piece to notice at this stage is just how the different blocks of 1's and 0's can combine to make a larger number, that then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

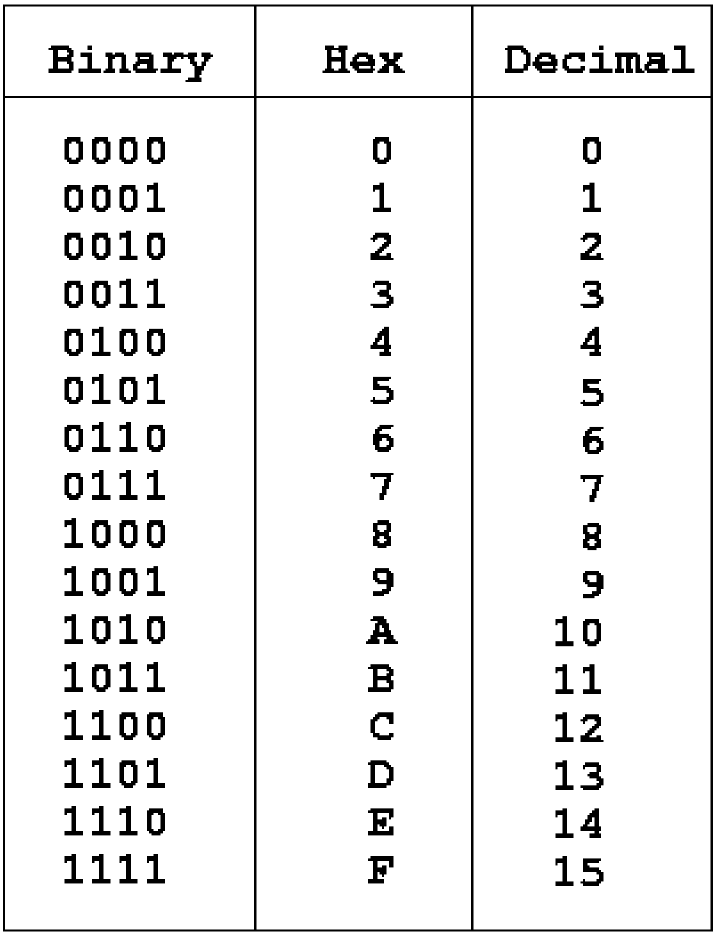
What happens if we want negative numbers? Well, we can choose the sign with one of our bits. But then we have only seven to give the values. Thus, we get a range of -128 – $+127$ for **signed 8-bit integer**.

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 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.

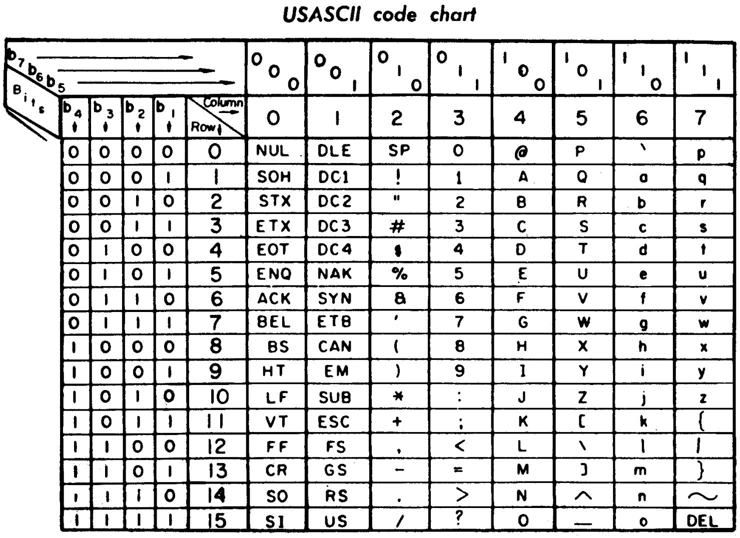
Python handles changes between these types, and the amount of memory required to store them, pretty much automatically. But not all programming languages do, and this is important to know for general programming literacy!



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 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



USASCII code chart

b7 b6 b5 → Bits					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	Column → Row ↓	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 the most simple and possibly most widely-used way of encoding letters and numbers. 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.

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

The most 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

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

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

```
# But varibales just point to a reference to containers for variables like lists.
# Hence, values are shared
a = [5]
b = a
a[0] -= 1
print 'a =', a, '; b =', b
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 Numpy, namespaces, and numpy arrays

I mentioned Python modules briefy above; a module is a set of functions and/or variables that can be useful for a specific task. "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!

6.8 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. @TODO: start here next time!

7.1 imperative

7.1.1 Without functions

7.1.2 With functions

7.2 Object-oriented