# Introduction to Programming for the Non-Programmer
# Mac OSX Edition

## Questions We Aim to Answer

- What can programming do for me?
- What kinds of problems are programs not suitable for?
- How do programmers think about code?
  - What is good structure? Bad structure?

## Concrete Goals

- Introduce python as a powerful "super-calculator"
- Get familiar with the python programming language
- Write some small python scripts
  - Learn what a "script" is, anyway
- Learn to use a terminal emulator ("terminal") to navigate the filesystem
  - Learn to use the terminal to run python scripts

# Part 1: Getting Set-Up

## Find the Terminal

Go to Spotlight Search or Applications and find the Terminal app. Many people prefer the freely available iTerm app, but both will behave the same for our purposes. Once you have launched the terminal, you will be presented with a command prompt, which is a simple text entry field. You can now type commands into the terminal, such as `ls` to list the contents of the current working directory. Do not worry too much about using the terminal – we will revisit the commands available there later on.

## Make Sure Python is Installed

Type `python` into the command prompt (the terminal's text field), and hit Return. You should be presented with something like this:

```
Python 2.7.3 (default, Aug  1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Don't worry too much about the specifics. The important part is that this program, the Python interpreter, launches. You will notice that the text field into which you typed `python` was preceded by text ending in a `$` symbol (we will discuss the meaning of this text when we cover Terminal commands), while the new field is not. Instead, you have the new text field or prompt, `>>>` . We will call this the "python prompt" or the "interpreter", but these both refer to the `>>>` text field. If python does not launch, your system is probably in some kind of trouble. You can install it from Python.org.

## Exit the Interpreter, and Quit Terminal

Let's make sure we're comfortable with launching python.

- Hit "Ctrl-D" or enter `exit()` to exit the python interpreter.
  - Ctrl-D is the "EOF" or "End of File" character, a special character which indicates the end of a file. You can conceive of characters as keystrokes that may or may not be possible to enter on a standard keyboard, and Ctrl-D as being equivalent to typing a special key, distinct from D.
    - Python interprets the EOF as a signal to exit.
  - `exit()` is a "function call". Essentially, it is the name of some piece of code which we would like to run.
    - Python evaluates `exit()`, which involves running the function named `exit`, which exits the interpreter
    - For now, you can think of `exit()` as being a magical command that exits the interpreter. We will talk in depth about functions soon
- Quit Terminal just like any application. Command+Q, or the menu
  - If we had not closed the interpreter before quitting, Terminal may have complained that a program was still being run by the Terminal

## Part 2: Python as a Calculator

### Reopen Python

Open python as before, let's start computing some things with it.

### Arithmetic

- Although python is a fully fledged language, for the moment we're just going to use it to perform simple calculations
- Try typing some simple arithmetic expressions into the interpreter, it should evaluate them. For example:

```
>>> 2 + 2
4
>>> (2+3)*4
20
>>> 4*8-2
30
>>> 2**3
8
```

- As in mathematics, we call `+`, `-`, `*`, and `/` operators. They are binary operations with specific precedence rules, etc. The only tricky business is the `**` operator, which python uses for exponentiation.
- Likewise, parentheses can be used to group operators and their operands
- Mysterious Behavior
  - What's going on here? You can play in the interpreter to find out, or read on.

```
>>> 1/2
0
>>> 3/2
1
>>> 2/3
0
```

- Python doesn't divide correctly! How can we fix this?
  - The issue is not quite that the division is incorrect. Python is performing Integer Division, trying to keep us in the realm of integers. Although Integer Division seems like a terrible thing, it is actually quite useful behavior in many programs. Oftentimes, programs use integers as indices into sequences of elements, so non-integer values are meaningless. Integer division makes it easier to check the correctness of these programs. The modern stance, however, is that programmers should specify when they want integer division.
  - Notice that integer division truncates, it does not round. For example, `2/3` is `0`, not `1`. Truncation is a simple operation to do by hand: simply discard the non-integer component of the quotient.
  - Python identifies real values as being distinct from integers, so `2.0/4.0` will evaluate to `0.5`, as it should.
- Okay, so Integer Division is a sometimes-useful thing, how do I do real division?
  - For now, we are going to enter a very specific command which fixes the behavior of division. `from __future__ import division` is a special statement which should be read as "use the version of division which will be the default in the future". In python versions earlier than 3.0, division works as we have described, but in python 3.x, we will have true division. This statement allows us to use true division without running python 3.0 or later.

```
>>> 1.0/2
0.5
>>> 1/2
0
>>> from __future__ import division
>>> 1/2
0.5
```

  - After we have imported true division, things work as we would like. If we want the older style, integer division, we use the `//` operator, so...

```
>>> 1//2
0
```

  - Try to predict the results before typing in these expressions
    - `1+2.0`
    - `3.0/4.0`
    - `6.2/3.1 ** 1`
    - `1.0 * 2**0.5`
    - `1.0/3.0 + 1.0/3.0 + 1.0/3.0`
    - `3//2`
- We will always assume, from this point forward, that we are using true division. Make sure that you run `from __future__ import division` each time that you start python. When python 3.x rolls around, this statement will no longer be necessary, but for now it is.

## Part 3: Beginning Abstraction

Thusfar, everything we have done has been relatively unexciting. We have used the Python Interpreter purely as a calculator, and though it can do amazing computations like $2^{15000}$ very fast, it isn't offering us much more than a TI-84 would. I would argue that building abstractions on top of one another is the core of programming as a technical skill, and has a central role in broader computer science as well.
We will begin by looking at variables, which are just named values, and move on to some simple functions. The first functions that we will deal with are of the same ilk as "f(x) = 2x" and other simple curves from algebra, but we will very quickly see that functions can be much more expressive than that. Understanding good abstractions is a fundamental skill that takes time, so don't rush, and try not to get frustrated if you find anything difficult to understand.

### Variables

A variable is just a named container. There is an important distinction between programatic variables and algebraic ones. When we deal with a variable, $x$, in algebra, it has only one fixed value. We would say that $x = 2$ and $x = 3$ are inconsistent statements about the value of $x$. By contrast, when we are programming, $x$ is just a name whose value might change over time. So $x$ only has a value at a specific point in time, not one fixed value throughout its lifetime, and we assign different values to $x$ over the course of our interaction with it.
It may help to think about variables a bit like files in your computer's filesystem. If you have "Essay.doc" on your Desktop, it is a file with only one possible given set of contents at a time, but you can replace it with a new, distinct document, with the same name. If you replace it by saving a new file, also named "Essay.doc" to the same location, however, the original contents will not be recoverable. This is a case of learn by doing, so, as you read the examples in this section, be sure to try a few things out yourself to see if you understand.

- Let's start by creating some variables in the interpreter. This should feel pretty intuitive. We assign values to variables, and create new variables, using the equals sign.

```
>>> x = 5
>>> x+1
6
>>> y=x/2.0
>>> y
2.5
>>> x = 4
>>> x
4
>>> y
2.5
```

- Notice that `y`'s value did not change when `x`'s value did, even though we defined `y` in terms of `x`. This is because `y` was defined by evaluating `x/2.0`, with the value of `x` at that point in time. After the assignment, `y` *does not retain any knowledge of how it was defined*, it's just a container with the value `2.5` inside.

- Variable assignment is always written with a variable name on the left and a value on the right.

Can you imagine why this is? What would the values of `x` and `y` be in the above example, if we followed it with the line `>>> x = y`? Try it and find out.

Ultimately, we are going to use variables to describe problems that we want to solve. If you have the ages of two people, Bob and Alice, and want to find their average age, you might think of this as

$$\frac{Age_{Bob} + Age_{Alice}}{2}$$

We would like our programs to be able to express this same idea, and as we will see, variables do precisely that.

- Let's start by expressing the average age of Alice and Bob in python.

```
>>> AgeBob = 32
>>> AgeAlice = 35
>>> (AgeBob+AgeAlice)/2
33.5
```

- By writing down the expression in this way, we have allowed for the ages of Bob and Alice to be set arbitrarily, while still computing the correct value. In the present case, this may seem unnecessary, but we can easily describe systems for which ensuring correctness is not necessarily trivial. Consider the following scenario in which we use Newtonian laws to describe a rolling disc.

```
>>> acceleration = 5
>>> time = 20
>>> velocity = acceleration*time
>>> mass = 350
>>> force = acceleration*mass
>>> radius = 2
>>> torque = force*radius
>>> displacement = (1/2)*acceleration*time**2
>>> angularVelocity = velocity*radius
>>> moment = (1/2)*mass*radius**2
>>> energy = (1/2)*mass*velocity**2 + (1/2)*moment*angularVelocity**2
```

- We can easily see that we have computed numerous values, displacement, angular velocity, and torque to name a few, which depend upon the values of acceleration and time. Without describing these as variables, and thereby encapsulating the operations by which we produced them, the computation for energy would be hard to read and understand: `(1/2)*350*(5*20)**2 + (1/2)*((1/2)*350*2**2)*(5*20*2)**2`

- Perhaps more importantly, if we do not record the value of `velocity`, then we must recompute it each time we use it. Not only is computational time we would like to save the computer (imagine that we are asking for values that take the computer much longer than simple multiplication), but it means trouble if I realize that I want to write `revolutions = displacement/(2*3.1415*radius)`. Without variables, I must think about a value, `displacement`, in terms of time and acceleration, and more than that, I must have the actual values of time and acceleration in mind when I write the arithmetic expression down.

- Remember, however, that if `acceleration` or `time` change, we must recompute all of the other values. They have no idea of how they were computed. To describe computations, rather than raw values, we will have to discuss functions...

## Simple Functions and the Math Library

We will begin by considering a few functions which Python provides for us. To gain access to Python's Math Library, we need to `import math`. We say that `math` is a Library because it is a repository of values and functions, and importing is the process by which we make an existing body of code available for our use. Think of `math` as a long list of definitions like `pi = 3.1415...` By importing `math`, we just evaluate a long series of these definitions, so that we can use them.

- Accessing values from the Math Library is simple, but requires that we use the "Dot Syntax", a notation that allows us to specify where a variable comes from. Observe:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> radius=10
>>> area = math.pi*radius**2
```

- The dot can be thought of as a kind of locational specifier, much like sections, subsections, and subsubsections of a large contract or similar document. `A.B` refers to a variable $B$ in some container, $A$.
  - If $B$ itself contains variables, we might right `A.B.C` to refer to one of these.
  - Dot notation shows up in a few other contexts, but always with this same general meaning of `X.Y`: "`X` is some characteristic or attribute of `Y`"
- Constants like these are really just variables, and python will even let us modify them. If we write `math.pi = 4`, we will really overwrite the value for $\pi$. The easiest way to fix this if you do it is to exit and reopen python, but we will revisit this issue later with a better solution.

We are not presently interested in the constants provided by `math`, but rather the functions it includes. You can find a full listing of these in the official Python documentation if you are curious, but we will inspect a few of interest, and introduce some basic vocabulary for programatic functions.

- If we have a python function named `f`, and write `y = f(x)`, then we say that `x` is an *argument* to `f`, and that `y` is a variable containing the *return* value of `f`, or that `f` *returns* `y`. Although these specific terms may not matter for us now, they are common parlance. Not only will they allow you to read official Python documentation, but they are also how I will describe functions as we work with them.
- Let's start with the square root function, named `sqrt`. Functions behave much as they do in algebra, so we can think of `sqrt` as $sqrt(x) = y$, where $y$ is the square root of $x$. `sqrt` takes a real number as an argument, and returns its square root.

```
>>> import math
>>> math.sqrt(100)
10.0
>>> math.sqrt(5)
2.23606797749979
```

- As a quick aside, can you guess why `math.sqrt(100)` produces `10.0` instead of `10`? Even though we imported true division, Python still distinguishes integers from real numbers, called "floats" in reference to the IEEE Floating Point standard.

We will not define our own functions just yet, but first get comfortable with the functions we have available, and what we want to describe. `math` defines trigonometric fuctions much as we would expect, so that `math.cos(2*math.pi)` is `1.0`, along with inverse trig functions, exponential functions, and an array of tools which we will not discuss.

- Let's take a look at the `math.pow` function. `math.pow` takes two arguments, and computes the same thing that the `**` operator does.

```
>>> import math
>>> math.pow(2,3)
8.0
>>> math.pow(3,2)
9.0
```

- Although `math.pow` does not provide us with new functionality, it introduces a few things which we will find very useful.

  1. Functions can take multiple arguments. Just as we might define a mathematical function in three dimensions by $f(x, y) = x + y$, `math.pow` operates on more than one parameter. In fact, a function can operate on any fixed number of arguments, and even, if we pull some tricks, an arbitrary, unknown number of arguments.
  2. The order of arguments matters. Giving a function the pair $(x, y)$ is different from the pair $(y, x)$. Sometimes this will matter, if we are doing a computation like $f(a, b) = a - b$, and sometimes it will not: $f(a, b) = a + b$.
  3. Function arguments are comma separated lists. This may seem obvious, and it is the modern convention across programming languages, but it is good to confirm.

- Thinking about the model set by `math.pow`, let us consider the problem we set ourselves earlier, averaging the ages of Bob and Alice.

  – We are not so much concerned with what we are averaging, but more with the problem of averaging two numbers. From now on, let us talk about the problem as one of averaging two numbers.
  – The function should take two numbers as parameters, and return their average.
  – Averaging numbers is commutative, so it should not matter which order we give the numbers in.
  – We know that the function needs to compute and return one half of the sum of its arguments.
  – It probably makes sense to give the function a name like "avg", so a *call* to it – also sometimes called an *invocation* of the function – should look like `avg(AgeBob,AgeAlice)`.

We now have a really good idea of what the `avg` function should look like, but no clear idea of how to write it. We will return to `avg`, but first we are going to write out a much simpler function, `double`. `double` will simply return twice the value of its only argument, so in standard algebraic notation $double(x) = 2x$.

- As we write out `double`'s definition, there are a few particulars worth paying attention to.

  – `def` is a keyword. This means that it is a word with special behavior. In the case of `def`, we read what follows as a function definition.
  – The colon symbol and indentation are syntactically significant. A colon indicates the beginning of a *block*, a series of expressions that should be treated as one unit. Functions are just one case in which blocks are important, but in this context, the block consists of the expressions which are evaluated each time that the function is executed.

  - Lines in a block following a colon must be indented. There are a lot of different rules that different people follow for exactly how the indentation should look, but Python requires that all lines in a block are indented the exact same amount. My personal preference for Python is to indent with four spaces, but most people find the tab character more convenient (just hit "Tab" to insert a tab character).
  - The `return` specifies the value that the function *returns* or produces. It must appear in the form
    ```
    return value
    ```
    where `value` is whatever we want the result of evaluating the function to be.
  - The *arguments* or *parameters* to the function are written, with its name, in a comma separated list, enclosed in parentheses. This is the same convention that we follow when defining algebraic functions, and, as in that case, this is where we give names to the variables that we will use inside of the function's definition.

- Here follows the definition of `double`
  ```
  >>> def double(x):
  ...     return 2*x
  ...
  >>> double(4)
  8
  ```

- Note that the interpreter converts the standard prompt " `>>>` " to an ellipsis "`...`" while we are writing a block. This is the interpreter's way of indicating to us that we are still inside of a block. When we define our functions inside of files, symbols like the prompt and ellipsis will be absent, and we will not be required to write them in – in fact, doing so would cause errors in the evaluation of those function definitions.

- We enter a newline without any content, the "Return" or "Enter" key, in order to end a block in the interpreter. That is why there is a gap between the definition of `double` above, and the *invocation* of it, or *call* to it, on the value `4`.

To recap, defining `double` has shown us the `def` keyword and the syntax it expects, the notion of blocks and their syntactic form in terms of colons and indentation, and the use of the `return` keyword to specify the value given by a function call. We also began using the terms *call* and *invocation* to refer to uses of a function, and *arguments* or *parameters* to refer to the variables appearing within a function definition and supplied when it is called.
Before we proceed directly to `avg`, our function which finds the average of two values, we will write out the similar and related functions `plus` and `minus` a few different ways. You may already be ready to write `avg` at this point, but the following examples should still prove instructive.

- The definition of `plus`:
  ```
  >>> def plus(alpha,beta):
  ...     return alpha + beta
  ```

- The definition above may seem intuitive, but consider the following invocations of it.

```
>>> x = 1
>>> plus(x,x)
2
>>> plus(double(x),x)
3
>>> x = plus(x,x)
>>> x
2
>>> x = plus(plus(x,double(x)),1)
```

- Work out, as an exercise, the final value of `x`. It may help to keep in mind that function invocation behaves much the same way as algebraic function evaluation, so you can evaluate this expression much as you would evaluate nested function applications $f(g(h(x)))$.
- An alternative definition of `plus`:

```
>>> def plus(alpha,beta):
...     x = alpha + beta
...     return x
```

- Two similar definitions for `minus`:

```
>>> def minus(alpha,beta):
...     x = alpha - beta
...     return x
...
>>> def minus2(alpha,beta):
...     return alpha - beta
...
```

Like variables, functions are a way of storing a computation so that we can use it later. We can write `avgofxandy = (x + y) / 2`, but this only defines a very specific average for us to use. If we want to define our general formula for the average of two values, we will want to write it as a function, and then we can say that `avgofxandy = avg(x,y)`. We want to effectively give the task or work – "the computation" – of averaging values a name, and refer to it by that name, `avg`.
To write `avg`, we will take two arguments, which we will call `x` and `y`, sum them and divide by two. Importantly, our work is not done: we need to return the result. If we fail to do so, our function `avg` will compute the average, but not hand it back to us to use. This behavior may be unintuitive now, but the distinction between computed values and returned values becomes important when we demand more from our functions. Consider the following definition of `avg`:

```
>>> def avg(x,y):
...     total = x + y
...     average = total / 2
...     return average
...
```