Introduction to Programming with Python

Guy Dickinson

NYC Resistor
guy@gdickinson.co.uk
@gdickinson

October 6, 2012

Hello

Here's what I'd like to do today:

- Give you an overview of basic programming concepts
- Give you a foundation of understanding of Python
- Give you the tools you need to continue to explore on your own

Here's what I can't do in one day:

- Transform you into a rockstar software engineer
- Make you ready to interview at Google/Facebook/wherever

What you should have with you

You should have a computer with a UNIX-like environment on it already, like Linux or MacOS X. A VM is fine, too. Trying to do this class using Windows is an exercise in pain and suffering.

What I Assume

It's always hard to gauge where students are coming from, but I assume the following:

- A strong intuitive understanding of computing/computers. It's especially helpful if you've used a terminal before.
- A willingness to learn and cope with frustration. Programming is hard and non-intuitive. Learning to program well is very, very hard and even less intuitive.
- No prior programming experience.

What is a Programming Language?

- Computer hardware knows nothing about Python. Or C. Or Java.
- Processors only know about *Machine Code*.
- Programming languages are fed to an intermediate program called a *compiler* or *interpreter* to be turned into machine code that your computer can understand.
- Programming languages help us to hide the tedious nature of machine code so as a developer, you don't need to worry about it.

The interpreter in action

- Let's fire up the interpreter. Open a terminal and run this:
  `$ python`
- You should get a prompt like this:
  ```
  Python 2.7.1 (r271:86832, Jul 31 2011, 19:30:53)
  >>>
  ```
- This is called a Read-Eval-Reply Loop (REPL); you can type code in here and each line will be *evaluated* and the results printed back to you.
- This is the point in the proceedings where we do the necessary, but not sufficient "Hello world!" program:
  ```
  >>> print("Hello world!")
  Hello world!
  ```
- We can also use Python as a desk calculator. Try `1+1`, `2.9 * 9`, `2 ** 16`, `98 % 3`...

Basic Syntax

- Statements are ended by a newline. You don't need to end things with a semicolon like in C or Java.
- Anything between a # and the end of a line does nothing. This is a *comment*.
- Everything is case sensitive. Print is different than **print**.
- Indentation matters. More on this later.

Variables

- We've done some basic operations on numbers (and one on a string).
- We can store the results in memory for later use instead of just having them printed out.
- The equals sign does this for us. The *value* on the right of the equals sign is evaluated and assigned to the name on the left.
- The variable can be used anywhere the original value could have been used.

Storing Programs in Files

- You don't have to keep typing everything into the interpreter; you can store your source code in a file instead.
- Files containing Python code should:
  - Start with the special incantation *#!/usr/bin/python* on the first line.
  - Be named with a suffix `.py`, like `helloworld.py`
- You can then run your code with `$ python helloworld.py`

A boring first program

```
#!/usr/bin/python

"""A super simple first program"""

input = raw_input()
print("You typed %s" % input)
```

Some questions:

1. What does this code do?
2. What's raw_input()?
3. What's the % thingy in the **print**()?
4. What happens when we reach the end of the code?

Conditionals

Suppose we want the user to guess some sort of secret. How would we go about that? We'll need a *conditional*. The simplest one is **if**. It looks like this:

```
if ( some_condition ):
  # Do stuff
  # Note the indentation; it matters.
else:
  # Do this instead
# Keep going here
```

What's a condition? It's an expression which must evaluate to true or false. True and False are *boolean* values. Everything in Python is considered to be "true" for the purposes of conditionals except False, None, and [] (the empty list which we'll see more of later).

Comparisons

How do we get a boolean value? We can use a comparison. Here are some common ones:

- $>$: Greater than
- $<$: Less than
- $==$: Equal to
- $!=$: Not equal to (**not** negates any boolean)

With variables, comparisons, and conditionals we have enough to implement a secret-guessing program.

Secret Guessing

```
#!/usr/bin/python
"""Guess a secret"""

secret = "cheddar"

print("What sort of cheese am I thinking of?")
guess = raw_input()

if (guess == secret):
  print("%s is right!" % secret)
else:
  print("Nope, I wasn't thinking of %s" % guess)

print("Thanks for playing")
```

More than one guess

Suppose we wanted to allow for more than one guess before the program exits. How do we do that? We'll need another type of conditional called a **while** loop. This is its basic structure:

```
while (some_condition):
  # Do this
# Once some_condition isn't true, carry on here.
```

Questions:

1. How would you allow for as many guesses as you wanted?
2. A fixed number of guesses?

Lists

- So far, we've seen numbers and strings. What if we want more than one of those?
- Lists can contain a arbitrary number of anything. We make them like this: $\text{mylist} = [1,\ 2,\ -0.5,\ \text{'a'},\ \text{"Hello!"}]$
- We can get to the stuff in there like this:

```
print(mylist[0]) # Outputs "1"
mylist[1] = 500 # Sets a new value
```

Dictionaries

- Dictionaries are composed of *key-value* pairs. They behave quite like lists except instead of using numbers based from 0, they use any object to get at its contents:

```
personinfo = {}
personinfo["firstname"] = "Guy"
personinfo["lastname"] = "Dickinson"
personinfo["hacker"] = true
personinfo.has_key["firstname"] # true
```

Functions

What is a function? A function hides away operations so you can use them again. This is best shown by example:

```
def plusone(x):
  return x + 1
plusone(x)
```

Functions: Terminology

- A function has 0 or more *arguments*, and *returns* 0 or 1 things.
- "Making a function go" is known as *calling* or *invoking* a function.
- Functions can call other functions, and can even call themselves. The latter is called *recursion*.

Another Example:

```
def factorial(x):
    if (x == 1):
        return x
    else:
        return x * factorial(x - 1)
factorial(10) # 3628800
```

Objects

- Objects glue together data and the functions which operate on that data.
- Suppose we wanted to model a car which has a direction, a speed, and number of occupants. How would we do that, in a world without objects?

Objects

Pretty tediously. Maybe we can do something like this:

```
car_speed = 0
car_occupants = 0
car_dirction = 0

def get_in():
  car_occupants += 1

def get_out():
  car_occupants -= 1

def steer(directionchange):
  car_direction =
    (car_direction + directionchange) % 360
```

What's wrong with this?

Objects

```
class Car:
  def __init__(self):
    self.speed = 0
    self.occupants = 0
    self.direction = 0

  def steer(self, dirchange):
    self.direction =
      (self.direction + dirchange) % 360
c = Car()
c.steer(-5)
c.direction # 355
```

Objects

Because now we're in object-oriented programming land, the same stuff has different names:

- Functions in an object are called *methods*
- Variables are called *attributes* or *fields*

Also there are some new things:

- A *class* is the blueprint for an object.
- The `__init__()` method is special. It is called a *constructor*. It is always called when you create an object.

Objects

Everything in Python is an object.
This is so important it bears repeating.
Everything in Python is an object.

Even simple things like strings are objects. They have methods, like upper():

```
name = "nyc resistor"
name.upper() # NYC RESISTOR
```

Documentation

You don't have to memorize all the methods that every object has.
That would be impossible.
You can make Python tell you.

```
>>> help(str)
```