Python Syntax

- Overview
- Python Shell
- Modules & Imports
- Commenting
- Control Statements
- Operators
- Variables
- Functions
- Scope

Example Python Script

In [1]:
```
import time

def square(x):
    return x * x

def yell():
    print "WOOOOO"

# starts executing here
print "The current time is:", time.asctime()

# conditional statement here
a = 10
if a < 10:
    print "a^2 when is equal to: %d" % square(a)
elif a == 10:
    print "a is just right."
else:
    print "a is too big!"
```

```
The current time is: Fri Mar 11 23:03:48 2016
a is just right.
```

Notes on Syntax

- Indentation is important in Python! 2-4 spaces or tabs are best, but be consistent!
- Colons are used for conditional, function, and class declarations
- No semicolons needed
- Single line comments use #

In [76]:
```
```

The Python Shell

```
$ python
```

or

```
$ ipython
```

Lab 1: "Hello, Your Name Goes Here!"

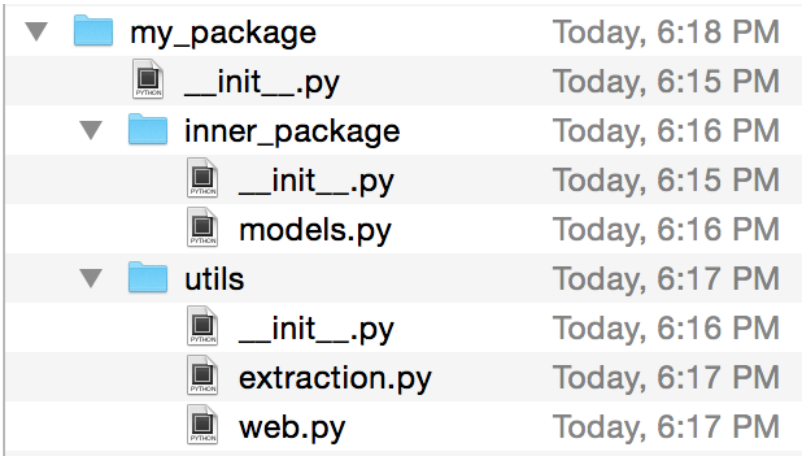
1. Create a file named `hello.py`
2. Run "`chmod +x hello.py`"
3. Put the below code in
4. Run it with `python hello.py`
5. Type in your name when prompted

In [1]:
```
#!/usr/bin/python
print "What is your name?\n"
name = raw_input()
print "Hello %s!" % name
```

```
What is your name?

Joe
Hello Joe!
```

Modules & Imports

- Modules == files
- Packages == folders



How do we import?

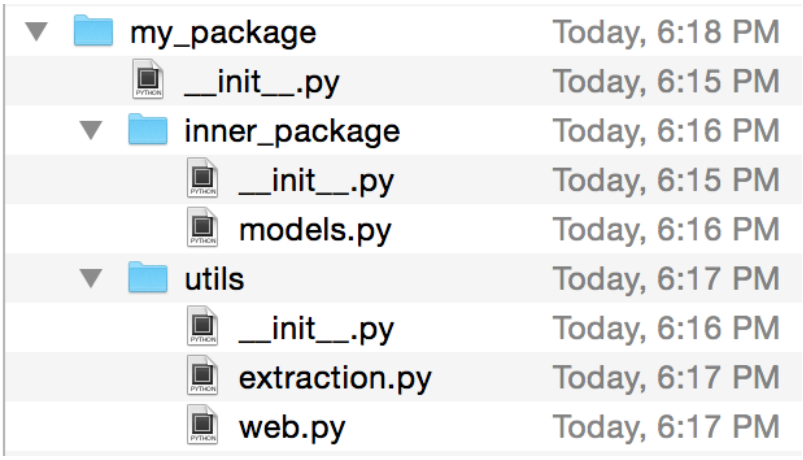
The Tau of Importing

In [ ]:
```
# most verbose
import my_package.inner_package.models.MyModel
m1 = my_package.inner_package.models.MyModel()

# less verbose
from my_package.inner_package.models import MyModel
m2 = MyModel()

# don't do this!
from my_package.inner_package.models import *
m3 = MyModel()

# a good compromise
from my_package.inner_package import models
m4 = models.MyModel()
```

Lab 2:

Open up your Python terminal and make sure you can import and run the `scrape()` function from the `utils` subpackage.

HINT: `scrape()` takes a URL string like "http://google.com (http://google.com)" and returns the HTML content of the page.



Commenting

Is extrordinarily important. Python is meant to be easy to read, but for anything non-obvious, they're a must.

In [ ]:
```
# this is a single line comment
print 2 + 2
print 3 + 3  # comments can also go here (generally two spaces after exp
ression end)

"""
This is a multi-line comment that might be used to explain a more diffic
ult concept:
- Can go
- For many lines
"""
print 10 * 10

# print "Don't execute this line"
```

Control Statements

In [24]:
```
counter = 0
while counter < 10:

    if counter % 2 == 0:
        print "Counter is even!"

    elif counter == 7:
        print "Counter is exactly seven!"

    else:
        print "Counter is odd, but not 7..."

    counter += 1

# how would we do this with a for loop?
```

```
Counter is even!
Counter is odd, but not 7...
Counter is even!
Counter is odd, but not 7...
Counter is even!
Counter is odd, but not 7...
Counter is even!
Counter is exactly seven!
Counter is even!
Counter is odd, but not 7...
even
odd, not 7
even
odd, not 7
even
odd, not 7
even
it's 7
even
odd, not 7
```

In Java or C/C++ (ish)

In [ ]:
```
for (int i = 0; i++; i < 10) {
    if (i % 2 == 0) {
        // even
    } else if (i == 7) {
        // 7
    } else {
        // other odd
    }
}
```

In Python

In [5]:
```
for i in range(0, 10, 1):
    if i % 2 == 0:
        print "even"
    elif i == 7:
        print "it's 7"
    else:
        print "odd, not 7"
```

```
even
odd, not 7
even
odd, not 7
even
odd, not 7
even
it's 7
even
odd, not 7
```

For loops, "continued"

- `break`: exits iteration of the loop
- `continue`: skips the rest of the body of the loop

In [152]:
```
for i in range(10):
    if i == 5:
        continue
    if i == 8:
        break
    print i
```

```
0
1
2
3
4
6
7
```

Operators in Python

- Arithmetic (+, –, *, /, //, %, **)
- Comparison (<, >, <=, >=, ==, !=)
- Assignment (=, +=, -=, *=, /=, %=, **=, //=)
- Bitwise (&, |, ^, ~, <<, >>)
- Boolean (`and`, `or`, `not`)
- Identity (`is`)
- Membership (`in`)

Pretty much like other languages, but without in/decrement operators like ++ or --.

Arithmetic Operators

In [31]:
```
x = 10
y = 3.0

# addition
print "x + y =", x + y

# subtraction
print "x - y =", x - y

# division vs. integer division
print "x / y", x / y
print "x // y:", x // y

# modulus
print "x % y = ", x % y

# exponentiation
print "x ** y = ", x ** y
```

```
x + y = 13.0
x - y = 7.0
x / y 3.33333333333
x // y: 3.0
x % y =  1.0
x ** y =  1000.0
```

Comparison Operators

In [ ]:
```
x = 10
y = 5
z = 6

# some examples
print x < y + z
print y + 1 == z
print x != z
print y <= 10

# you can also chain them
print 1 < 3 < 5 <= 5 < 100
```

Assignment Operators

These operators assign value to the left-hand side of the expression.

In [37]:
```
x = 10
y = 2
z = 1

# mulitplicative
x *= y
print x

# moduolo
x %= 3
print x

# and so on...
```

```
20
2
```

Multiple Assignment

Python supports multiple assignment, but you should think twice if you ever *really* need to use it.

In [110]:
```
# multiple assignment
x = y = z = 2
print x, y, z

# what's more interesting...
x = 3
print x, y, z
```

```
2 2 2
3 2 2
```

Bitwise Operators

Two types:

- Shifting
- Logical

Shifting

In [59]:
```
# Shifting
a = int('00000100', 2)  # 2^2 = 4
print "original a =", a

# Left shift
print a << 1  # 00001000

# # Right shift
print a >> 1  # 00000010
print a >> 2  # 00000001
print a >> 3  # 00000000
```

```
original a = 4
8
2
1
0
0
```

Logical / Binary

These are mostly used for lower-level or performant numeric math (graphics, images, etc) but have also been overridden for certain Python data structures in ways we'll see later.

In [ ]:
```
# bitwise AND
print 1 & 0

# bitwise OR
print 1 | 0

# bitwise XOR
print 1 ^ 0

# bitwise NOT, a unary operator
print ~1  # can you guess what this outputs? :)
```

Boolean Operators (`and`, `or`, `not`)

In [68]:
```
# can be used with booleans
print True or False  # still True
print True and False  # not True!

# or with expressions themselves
# and `not` is unary, so we can negate `and` or an `or` with it!
print 1 < 2 and not 3 > 4

# we can also chain them, or use parens
print ((1 < 2 and 3 < 4 or 2 < 1) and 10 < 100) or not 1 < 99
```

```
True
False
True
True
```

Identity Operator (`is`)

In [69]:
```
# gives us a stronger gurantee than equality
print True is 1
print True is True

# can also use with the boolean operator, `and`
print True is not False
```

```
False
True
True
```

Membership Operator (`in`)

We haven't talked about containers, lists, or sets yet, but this operator tests for membership of an element.

In [71]:
```
print 'b' in ['a', 'b', 'c']
print 'd' not in ['a', 'b', 'c']
```

```
True
True
```

Variables

Python is not a statically typed language - as you've noticed we don't delcare types.

It's a dynamic or "duck typed" language: if it looks and it talks like a certain datatype, that's what it is!

Primative Types

- Numeric
  - Integer
  - Floating point
- Boolean
- None
- String

No `char` and no native `int8`, etc in Python.

Naming Conventions in Python

- `UpperCaseClassName` (classes)
- `variable_or_function` (variables, functions)
- `CONSTANT_VALUE` (constants)

Never camelCase!

Numeric Types (`int`, `float`)

In [97]:
```
# numeric types
a = 10
b = 3.0

print "a is an integer:", a
print "b is a float:", b
print "a, converted to a float:", float(a)

# why do we care?
print a / int(b)
print a / b
print "%.20f" % (1 + 0.1,)
```

```
a is an integer: 10
b is a float: 3.0
a, converted to a float: 10.0
3
3.33333333333
1.10000000000000008882
```

Boolean Types (`True`, `False`)

In [77]:
```
a = True
b = False

print a or b
```

```
True
```

None Type (`None`)

This is an important type - it serves much like `NULL` or `nil` do in other languages.

In [99]:
```
a = None
print "a =", a

# also, None is False-y
if a:
    print "A is something"
else:
    print "a is 0, False, None, or an empty iterable"
```

```
a = None
a is 0, False, None, or an empty iterable
```

String Type

There are no `char`s in Python, yet strings are still array-like in usage. We'll get much more in depth with strings later.

In [14]:
```
message = "This is a sentence."
# print message

# we can access a character by location (0-indexed)
print message[-2]

# or even a range of characters from i -> end
#print message[5:]

# or a closed range, from the 11th character to the 19th
#print message[10:18]
```

```
e
```

In [ ]:
```

```

Most Importantly...

...variables are dynamically typed

In [107]:
```
# value is None
value = None
print value

# assign to a numeric
value = 1
value += 4
print value

# now make it boolean
value = False
print value
```

```
None
5
False
```

but also "duck typed"...

In [108]:
```
# show with a conditional
a = 0
if a:
    print "a is Truth-y"

# try with negation
b = None
if not b:
    print "B could be None, False, 0, [], (,) or set([])!"
```

```
B could be None, False, 0, [], (,) or set([])!
```

Multiple Assignment

Python also supports multiple assignment - though again this is discouraged unless you have a good reason for doing so, usually in the case that the variables you are assigning are conceptually linked in some way.

In [114]:
```
x_coord, y_coord, z_coord = 2, 4, 8

print x_coord
print y_coord
print z_coord
```

```
2
4
8
```

Lab: Coding Excercises

Fill in the method definitions in the file `excercises/syntax.py`.

Make sure you can pass tests with:

```
$ py.test tests/test_syntax.py::SyntaxExcercises::<function_name>  # test si
ngle function
$ py.test tests/test_syntax.py::SyntaxExcercises                   # test al
l at once
```

Syntax Wrap-Up

- **Python Shell**
  - Python
  - iPython (tab complete, system calls)
- **Modules & Imports**
  - Modules == Files
  - Packages == Folders
  - The Tau of Importing
- **Commenting**
  - Single line with (#)
  - Multi-line with triple quotes
- **Control Statements**
  - if / elif / else
  - for / while
- **Operators**
  - Arithmetic (+, -, *, /, //, %, **)
  - Comparison (<, >, <=, >=, ==, !=)
  - Assignment (=, +=, -=, *=, /=, %=, **=, //=)
  - Bitwise (&, |, ^, ~, <<, >>)
  - Boolean (`and`, `or`, `not`)
  - Identity (`is`)
  - Membership (`in`)
- **Variables**
  - Dynamic "Duck-typing"
  - Primatives
    - None
    - Numeric
    - Strings
    - Boolean
  - Multiple assignment