Python

Just enough to be dangerous

Python is...

- A popular object oriented programming language.
- A *scripting language*, meaning a language which is easy to code in but isn't particularly fast. (Python is typically about 1/10 the speed of Java).
- Specifically designed to be easy to code in, but rigidly enforcing a certain programming style.
- A language which *requires* a certain indent style.
- A language with dynamic binding for everything (no type declarations)
- Older than Java. And with a bunch of warts.

Python versions

- There are two kinds of Python:
  - Python 2.x is what everyone uses
  - Python 3.x is intended to be the future. It's not yet clear if it will be.
- We will use Python 2.x

Hello World

This is a comment.

```
# This is a comment
```

This is a statement.

```
print "Hello, World!"
```

Statements in Python end in a *new line.*

***print is not a function:* it does not have parentheses.**
[Note: as of 3.x, print is now a function]

It is a special kind of **operator** like + or -. It is accompanied by one or more expressions.

"Hello, World!" is a string.

Hello World

***print* takes an string, number, or other object**

```
print "Hello World"
print 9
print 4 + 3
```

Variables and Numbers

Variables do not need to be defined. Nor do you have to declare their type. You can just start using them.

```
x = 9
y = 10
z = x + y * x
print z + x
```

Standard math operators:

+ - * / %

```
a = x % 2
```

Nonstandard math operators:

//	integer division	z = y // 4
**	power (a^b)	w = x ** 2

Integers and Floats

```
b = 2
c = 3.14159
```

Variables and Numbers

Certain math functions are built-in.

```
r = -4
q = abs(r)
q = round(c)
```

Standard comparisons

< > == != >= <=

```
print a < b
```

Binary arithmetic

x << y	left shift by y bits	a = a << 4
x >> y	right shift by y bits	b = b >> 1
	(I think it fills with 0)	a & ~b
x & y	bitwise AND	
x | y	bitwise OR	
x ^ y	bitwise XOR	
~x	bitwise NOT	

Boolean Values and Strings

Boolean Literals

```
a = True
b = False
```

Boolean Operators

and **or** **not**

```
print a and b
c = a or not b
d = not(3 < w or w * 4 = z)
```

String Literals

... can use *either* single or double quotes, and have the opposite inside the string. Escapes include \", \', and \n

```
"Hello, World"
'Hello, World'
'He said, "You dont say!" to me'
"I'm a little teapot"
"I'm good, but he said \"No way\"! It's True!"
"Hello\nthere"
```

Making a Function

def *functionName*(*arg, arg, ...*):
    body
    [one more blank line here]

VERY important: note that *body* is indented. You can indent with whatever you like (a tab, 4 spaces, 15 spaces, whatever), but you MUST BE CONSISTENT or Python will bail on you.

Even blank lines inside your function must be indented.

To return something, use the **return** statement.

```
def printTheSum(x, y):
    print x + y
```

```
def printTheSumAndProvideTheProduct(x, y):
    print x + y
    return x * y
```

Calling a Function

Function calls are expressions.

```
a = 7 + printTheSumAndProvideTheProduct(2,4)

print convertToFahrenheit(451 + abs(celsius( )))
```

If and While

if *expression:*
    statement
    ...
elif expression:
    statement
    ...
...
else:
    statement
    ...
<blank line here>

while *expression:*
    statement
    ...
<blank line here>

```
def compare(first, second):
    if first > second:
        return 1
    elif first < second:
        return -1
    else:
        print "They're equal, hurrah!"
        return 0
```

```
def countBeers(n):
    while n > 0:
        print n
        print "Beers on a wall"
        n = n - 1
```

Arrays (“Lists”)

A List starts with [
ends with]
delimits items with **,**

```
elt = ["Hello", "World", "What", "Is", "That?"]
```

You can **access element *i*** in the array with **[*i*]** (the first element is element 0)

```
print elt[3]
```

You set elements in the same way.

```
elt[4] = "Yo"
```

Elements need not be all the same type. Here one element has been set to number.

```
elt[2] = 100
```

You can print the whole thing.

```
print elt
```

Some List Operations

+ concatenates lists

```
elt + [100, 200, "whoa!"]
lis = ["a", "b", "c", "d", "e", "f", "g", "h", "i"]
```

: creates sublists:

[*a* : *b*] From item *a* up to but not including *b*

[4 : 5] Element 4

[2 : 6] Elts 2 through 5

[: 6] Elts up to 6, but not including 6

[3 :] Elts from 3 and on

[:] Just copy the whole list

lis[4:5]	*['e']*
lis[2:6]	*['c', 'd', 'e', 'f']*
lis[: 6]	*['a', 'b', 'c', 'd', 'e', 'f']*
lis[3 :]	*['d', 'e', 'f', 'g', 'h', 'i']*
lis[:]	*['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']*

Some List Operations

.reverse() reverses the list

```
lis.reverse()
print lis        ['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

.sort() sorts the list

```
stuff = [1, 4, 9, 2, 6, 8, 3, 5, 7, 0]
stuff.sort()
print stuff                [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

.append(*object*) adds it to the end of the list

```
stuff.append("yo")
print stuff        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "yo"]
```

.pop() removes the last object and returns it, shortening the list.

```
a = stuff.pop();
print a                                          "yo"
print stuff                [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

These functions **modify the original list.**

Composite Data Structures

Arrays (or *Lists*)
Modifiable arrays of items

```
names = ["ZD", "Michael", "George", "Biff"]
ages = [100, 23, 49, 41, 46]
firstName = names[0]
names[2] = "Bob"
```

Tuples
Non-modifiable arrays of items

```
names = ("ZD", "Michael", "George", "Biff")
ages = (100, 23, 49, 41, 46)
firstName = names[0]
```

Sets
Unordered collections of items
Not often used.

```
names = set( ["ZD", "Michael", "Bob"] )
ages = set( [100, 23, 49, 41, 46] )
```

Dictionaries
Unordered collections of pairs of items (a *key* and a *value*)

```
agesPerPerson = { "ZD" : 100, "Michael" : 23}
ageOfZD = agesPerPerson["ZD"]
agesPerPerson["ZD"] = 53
agesPerPerson["Bob"] = 42
```

Some Dictionary Operations

.has_key(*key*)
Returns true if the key exists in the dictionary.

```
myDictionary.has_key("ZD")
```

.get(*key, defaultValue*)
Returns the value associated with the key. If there is no such key, returns the defaultValue.

```
myDictionary.get("ZD", -1)
```

.keys( )
Returns all the keys in the dictionary as a list.

```
keys = myDictionary.keys( )
```

del *dictionary*[*key*]
Deletes a given key-pair in the dictionary.

```
del myDictionary["ZD"]
```

Composite Data Structures

Need an array of dictionaries? A list of lists (essentially a two-dimensional array)?

No problem.

```
a = [ { "Yo" : 4, "Whassup" : 15 }, { "Okay" : 9 }]

b = { "Yo" : [1, 2, 3, 4], "Okay" : (1, 2, 3) }

c = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Keys for dictionaries can be any read-only object (numbers, strings, tuples, etc.)

```
a = { 2 : "Hello", (1, 2, 3) : "No way" }
```

for

for *variable* in *composite:*
    statement
    ...
<blank line>

```
for i in "Hello World":
    print i

a = 0
for i in [1, 2, 3]:
    a = a + i

w = {"first": "Zoran", "last": "Duric", "age": 100}
for i in w:
    print i
    print w[i]

w = [{"x": 4, "y": 5}, {"x": 9, "y": 10}]
for i in w:
    print "Next Coordinate"
    for j in i:
        print j
        print i[j]
```

for and range

range(*start, end)*
Produces a list of numbers from start inclusive to end exclusive

for *variable* in range*(start, end):*
    statement
    ...

this doesn't work:

You'd think that this would create a giant list and thus be very memory wasteful, but it's not. Python recognizes and handles it:

```
b = range(1, 1000)
```

```
for i in range(1, 100):
    print i
```

```
for i in range(100, 1):
    print i
```

```
a = 1
for i in range(1, 100000):
    a = a + i

print a
```

Objects

Python has classes and instances.

```
# Assume that a class called Robot is defined
```

Objects are created from classes using a *function of the same name as the class.*

```
myRobot = Robot( )
# or perhaps...
myRobot = Robot(2.0, 1.23)
```

Calling a method on an object is very much like in Java.

```
myRobot.goForward(1.5)
sensors = myRobot.currentSensors( )
```

Defining a Class

class *ClassName*(*superclass*):
**   def __init__(self):**
**   *constructor code***
**   def *methodName*(self, *args*):**
**   *method code***
**   ...**

The standard top-level superclass is object

Instance variables are not defined in the class itself. You create them on-the-fly or access them with.

**   self.*variableName***

```
class SimpleStack(object):
    def __init__(self):
        self.stack = [ ]  # set it to empty list
    def push(self, object):
        self.stack.append(object)
    def pop(self):
        return self.stack.pop();
```

```
# instance variables are always
# accessed with "self".  You can't just
# say "stack" above because that would
# be a local variable.  The instance variable
# is "self.stack"
```

Modules

Python modules perform the same function as Java packages. In Python, a module consists of a single file.

You load a module and all of its class definitions and functions with

import *modulename*

```
import math
import create
```

You access functions and class names in the module as

modulename.function
modulename.ClassName

```
a = 5.2 + math.sin(b * 3.2)
robot = create.Robot(5000)
```

Where to go next?

- diveintopython.org is a great book, and it's free
- learnpythonthehardway.org by a famous and cantankerous author
- www.python.org the official python site