LECTURE 1

Getting Started with Python

ABOUT PYTHON

- Development started in the 1980’s by Guido van Rossum.
  - Only became popular in the last decade or so.
- Python 2.x currently dominates, but Python 3.x is the future of Python.
- Interpreted, very-high-level programming language.
- Supports a multitude of programming paradigms.
  - OOP, functional, procedural, logic, structured, etc.
- General purpose.
  - Very comprehensive standard library includes numeric modules, crypto services, OS interfaces, networking modules, GUI support, development tools, etc.

PHILOSOPHY

From *The Zen of Python* (https://www.python.org/dev/peps/pep-0020/)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

NOTABLE FEATURES

- Easy to learn.
- Supports quick development.
- Cross-platform.
- Open Source.
- Extensible.
- Embeddable.
- Large standard library and active community.
- Useful for a wide variety of applications.

GETTING STARTED

Before we can begin, we need to actually install Python!

It is recommended that you set up a Linux virtual machine – this will save you a lot of headache later on in the course.

You can use any VM software you'd like, but I recommend Virtual Box. To ensure that the class is all on the same page, I'd also recommend using Ubuntu 14.04 as your virtualized OS. This will make installing and setting up packages much easier for you.

Your very first task is to set up your VM, make sure Python 2.7 is installed (it should be!) and write a simple Hello World program to make sure you're good to go. Do not put this off until Friday (when your first assignment is due)!

GETTING STARTED

- Choose and install an editor.
  - For Linux, I prefer SublimeText.
  - Windows users will likely use Idle by default.
  - Options include vim, emacs, Notepad++, PyCharm, Eclipse, etc.

Throughout this course, I will be using SublimeText in an Ubuntu environment for all of the demos.

INTERPRETER

- The standard implementation of Python is interpreted.
  - You can find info on various implementations here.
- The interpreter translates Python code into bytecode, and this bytecode is executed by the Python VM (similar to Java).
- Two modes: normal and interactive.
  - Normal mode: entire .py files are provided to the interpreter.
  - Interactive mode: read-eval-print loop (REPL) executes statements piecewise.

INTERPRETER: NORMAL MODE

Let’s write our first Python program!

In our favorite editor, let’s create helloworld.py with the following contents:

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

From the terminal:

```
$ python helloworld.py
Hello, World!
```

Note: In Python 2.x, print is a statement. In Python 3.x, it is a function. If you want to get in the 3.x habit, include at the beginning:

```
from __future__ import print_function
```

Now, you can write

```
print(“Hello, World!”)
```

INTERPRETER: NORMAL MODE

Let’s include a she-bang in the beginning of helloworld.py:

```
#!/usr/bin/env python
print "Hello, World!"
```

Now, from the terminal:

```
$ ./helloworld.py
Hello, World!
```

INTERPRETER: INTERACTIVE MODE

Let's accomplish the same task (and more) in interactive mode.

Some options:
-c : executes single command.
-O: use basic optimizations.
-d: debugging info.
More can be found here.

```
$ python
>>> print "Hello, World!"
Hello, World!
>>> hellostring = "Hello, World!"
>>> hellostring
'Hello, World!'
>>> 2*5
10
>>> 2*hellostring
'Hello, World!Hello, World!'
>>> for i in range(0,3):
...     print "Hello, World!"
...
Hello, World!
Hello, World!
Hello, World!
>>> exit()
$
```

SOME FUNDAMENTALS

- Whitespace is significant in Python. Where other languages may use {} or (), Python uses indentation to denote code blocks.

- Comments
  - Single-line comments denoted by #.
  - Multi-line comments begin and end with three “s.
  - Typically, multi-line comments are meant for documentation.
  - Comments should express information that cannot be expressed in code – do not restate code.

```
# here’s a comment
for i in range(0,3):
    print i
def myfunc():
    """here’s a comment about
    the myfunc function""“
    print "I'm in a function!"
```

PYTHON TYPING

- Python is a strongly, dynamically typed language.
- Strong Typing
  - Obviously, Python isn't performing static type checking, but it does prevent mixing operations between mismatched types.
  - Explicit conversions are required in order to mix types.
  - Example: 2 + "four" ← not going to fly
- Dynamic Typing
  - All type checking is done at runtime.
  - No need to declare a variable or give it a type before use.

Let's start by looking at Python's built-in data types.

NUMERIC TYPES

The subtypes are int, long, float and complex.

- Their respective constructors are int(), long(), float(), and complex().

- All numeric types, except complex, support the typical numeric operations you'd expect to find (a list is available here).
- Mixed arithmetic is supported, with the "narrower" type widened to that of the other. The same rule is used for mixed comparisons.

NUMERIC TYPES

- Numeric
  - **int**: equivalent to C's long int in 2.x but unlimited in 3.x.
  - **float**: equivalent to C's doubles.
  - **long**: unlimited in 2.x and unavailable in 3.x.
  - **complex**: complex numbers.

  - Supported operations include constructors (i.e. int(3)), arithmetic, negation, modulus, absolute value, exponentiation, etc.

```
$ python
>>> 3 + 2
5
>>> 18 % 5
3
>>> abs(-7)
7
>>> float(9)
9.0
>>> int(5.3)
5
>>> complex(1,2)
(1+2j)
>>> 2 ** 8
256
```

SEQUENCE DATA TYPES

There are seven sequence subtypes: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.

All data types support arrays of object but with varying limitations.

The most commonly used sequence data types are strings, lists, and tuples. The xrange data type finds common use in the construction of enumeration-controlled loops. The others are used less commonly.

SEQUENCE TYPES: STRINGS

Created by simply enclosing characters in either single- or double-quotes.

It’s enough to simply assign the string to a variable.

Strings are immutable.

There are a tremendous amount of built-in string methods (listed here).

```
mystring = "Hi, I'm a string!"
```

SEQUENCE TYPES: STRINGS

Python supports a number of escape sequences such as ‘\t’, ‘\n’, etc.

Placing ‘r’ before a string will yield its raw value.

There is a string formatting operator ‘%’ similar to C. A list of string formatting symbols is available here.

Two string literals beside one another are automatically concatenated together.

```
print "\tHello,\n"
print r"\tWorld!\n"
print "Python is " "so cool."
```

```
$ python ex.py
        Hello,

\tWorld!\n
Python is so cool.
```

SEQUENCE TYPES: UNICODE STRINGS

Unicode strings can be used to store and manipulate Unicode data.

As simple as creating a normal string (just put a ‘u’ on it!).

Use Unicode-Escape encoding for special characters.

Also has a raw mode, use ‘ur’ as a prefix.

To translate to a regular string, use the .encode() method.

To translate from a regular string to Unicode, use the unicode() method.

```
myunicodestr1 = u"Hi Class!“
myunicodestr2 = u"Hi\u0020Class!“
print myunicodestr1, myunicodestr2
newunicode = u'\xe4\xf6\xfc‘
print newunicode
newstr = newunicode.encode('utf-8')
print newstr
print unicode(newstr, 'utf-8')
```

Output:

```
Hi Class! Hi Class!
äöü
äöü
äöü
```

SEQUENCE TYPES: LISTS

Lists are an incredibly useful *compound* data type.

Lists can be initialized by the constructor, or with a bracket structure containing 0 or more elements.

Lists are mutable – it is possible to change their content. They contain the additional mutable operations previously listed.

Lists are nestable. Feel free to create lists of lists of lists…

```
mylist = [42, 'apple', u'unicode apple', 5234656]
print mylist
mylist[2] = 'banana`
print mylist
mylist [3] = [['item1', 'item2'], ['item3', 'item4']]
print mylist
mylist.sort()
print mylist
print mylist.pop()
mynewlist = [x for x in range(0,5)]
print mynewlist
```

```
[42, 'apple', u'unicode apple', 5234656]
[42, 'apple', 'banana', 5234656]
[42, 'apple', 'banana', [['item1', 'item2'], ['item3', 'item4']]]
[42, [['item1', 'item2'], ['item3', 'item4']], 'apple', 'banana']
banana
[0, 1, 2, 3, 4]
```

SEQUENCE DATA TYPES

- Sequence
  - **str**: string, represented as a sequence of 8-bit characters in Python 2.x.
  - **unicode**: stores an abstract sequence of code points.
  - **list**: a compound, mutable data type that can hold items of varying types.
  - **tuple**: a compound, immutable data type that can hold items of varying types. Comma separated items surrounded by parentheses.
  - a few more – we'll cover them later.

```
$ python
>>> mylist = ["spam", "eggs", "toast"] # List of strings!
>>> "eggs" in mylist
True
>>> len(mylist)
3
>>> mynewlist = ["coffee", "tea"]
>>> mylist + mynewlist
['spam', 'eggs', 'toast', 'coffee', 'tea']
>>> mytuple = tuple(mynewlist)
>>> mytuple
('coffee', 'tea')
>>> mytuple.index("tea")
1
>>> mylonglist = ['spam', 'eggs', 'toast', 'coffee', 'tea', 'banana']
>>> mylonglist[2:4]
['toast', 'coffee']
```

COMMON SEQUENCE OPERATIONS

All sequence data types support the following operations.

Operation	Result
x in s	True if an item of s is equal to x, else False.
x not in s	False if an item of s is equal to x, else True.
s + t	The concatenation of s and t.
s * n, n * s	n shallow copies of s concatenated.
s[i]	ith item of s, origin 0.
s[i:j]	Slice of s from i to j.
s[i:j:k]	Slice of s from i to j with step k.
len(s)	Length of s.
min(s)	Smallest item of s.
max(s)	Largest item of s.
s.index(x)	Index of the first occurrence of x in s.
s.count(x)	Total number of occurrences of x in s.

COMMON SEQUENCE OPERATIONS

Mutable sequence types further support the following operations.

Operation	Result
s[i] = x	Item i of s is replaced by x.
s[i:j] = t	Slice of s from i to j is replaced by the contents of t.
del s[i:j]	Same as s[i:j] = [].
s[i:j:k] = t	The elements of s[i:j:k] are replaced by those of t.
del s[i:j:k]	Removes the elements of s[i:j:k] from the list.
s.append(x)	Same as s[len(s):len(s)] = [x].

COMMON SEQUENCE OPERATIONS

Mutable sequence types further support the following operations.

s.extend(x)	**Same as s[len(s):len(s)] = x.**
s.count(x)	Return number of i’s for which s[i] == x.
s.index(x[, i[, j]])	Return smallest k such that s[k] == x and i <= k < j.
s.insert(i, x)	Same as s[i:i] = [x].
s.pop([i])	Same as x = s[i]; del s[i]; return x.
s.remove(x)	Same as del s[s.index(x)].
s.reverse()	Reverses the items of s in place.
s.sort([cmp[, key[, reverse]]])	Sort the items of s in place.

BASIC BUILT-IN DATA TYPES

- Set
  - **set**: an unordered collection of unique objects.
  - **frozenset**: an immutable version of set.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange']
>>> fruit = set(basket)
>>> fruit
set(['orange', 'pear', 'apple'])
>>> 'orange' in fruit
True
>>> 'crabgrass' in fruit
False
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b
set(['r', 'd', 'b'])
>>> a | b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
```

BASIC BUILT-IN DATA TYPES

```
$ python
>>> gradebook = dict()
>>> gradebook['Susan Student'] = 87.0
>>> gradebook
{'Susan Student': 87.0}
>>> gradebook['Peter Pupil'] = 94.0
>>> gradebook.keys()
['Peter Pupil', 'Susan Student']
>>> gradebook.values()
[94.0, 87.0]
>>> gradebook.has_key('Tina Tenderfoot')
False
>>> gradebook['Tina Tenderfoot'] = 99.9
>>> gradebook
{'Peter Pupil': 94.0, 'Susan Student': 87.0, 'Tina Tenderfoot': 99.9}
>>> gradebook['Tina Tenderfoot'] = [99.9, 95.7]
>>> gradebook
{'Peter Pupil': 94.0, 'Susan Student': 87.0, 'Tina Tenderfoot': [99.9, 95.7]}
```

- Mapping
  - **dict**: hash tables, maps a set of keys to arbitrary objects.

PYTHON DATA TYPES

So now we’ve seen some interesting Python data types.

Notably, we’re very familiar with numeric, strings, and lists.

That’s not enough to create a useful program, so let’s get some control flow tools under our belt.

CONTROL FLOW TOOLS

While loops have the following general structure.

```
while expression:
    statements
```

Here, statements refers to one or more lines of Python code. The conditional expression may be any expression, where any non-zero value is true. The loop iterates while the condition is true.

Note: All the statements indented by the same amount after a programming construct are considered to be part of a single block of code.

```
i = 1
while i < 4:
    print i
    i = i + 1
flag = True
while flag and i < 8:
    print flag, i
    i = i + 1
```

```
1
2
3
True 4
True 5
True 6
True 7
```

CONTROL FLOW TOOLS

The if statement has the following general form.

```
if expression:
    statements
```

If the boolean expression evaluates to True, the statements are executed. Otherwise, they are skipped entirely.

```
a = 1
b = 0
if a:
    print "a is true!"
if not b:
    print "b is false!"
if a and b:
    print "a and b are true!"
if a or b:
    print "a or b is true!"
```

```
a is true!
b is false!
a or b is true!
```

CONTROL FLOW TOOLS

You can also pair an else with an if statement.

```
if expression:
    statements
else:
    statements
```

The elif keyword can be used to specify an else if statement.

Furthermore, if statements may be nested within eachother.

```
a = 1
b = 0
c = 2
if a > b:
    if a > c:
        print "a is greatest‟
    else:
        print "c is greatest‟
elif b > c:
    print "b is greatest‟
else:
    print "c is greatest"
```

```
c is greatest
```

CONTROL FLOW TOOLS

The for loop has the following general form.

```
for var in sequence:
    statements
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable var. Next, the statements are executed. Each item in the list is assigned to var, and the statements are executed until the entire sequence is exhausted.

For loops may be nested with other control flow tools such as while loops and if statements.

```
for letter in "aeiou":
    print "vowel: ", letter
for i in [1,2,3]:
    print i
for i in range(0,3):
    print i
```

```
vowel: a
vowel: e
vowel: i
vowel: o
vowel: u
1
2
3
0
1
2
```

CONTROL FLOW TOOLS

Python has two handy functions for creating a range of integers, typically used in for loops. These functions are range() and xrange().

They both create a sequence of integers, but range() creates a list while xrange() creates an xrange object.

Essentially, range() creates the list statically while xrange() will generate items in the list as they are needed. We will explore this concept further in just a week or two.

For very large ranges – say one billion values – you should use xrange() instead. For small ranges, it doesn't matter.

```
for i in xrange(0, 4):
    print i
for i in range(0,8,2):
    print i
for i in range(20,14,-2):
    print i
```

```
0
1
2
3
0
2
4
6
20
18
16
```

CONTROL FLOW TOOLS

There are four statements provided for manipulating loop structures. These are break, continue, pass, and else.

- break – terminates the current loop.
- continue – immediately begin the next iteration of the loop.
- pass – do nothing. Use when a statement is required syntactically.
- else – represents a set of statements that should execute when a loop terminates.

```
for num in range(10,20):
    if num%2 == 0:
        continue
    for i in range(3,num):
        if num%i == 0:
            break
    else:
        print num, 'is a prime number'
```

```
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
```

OUR FIRST REAL PYTHON PROGRAM

Ok, so we got some basics out of the way. Now, we can try to create a real program.

I pulled a problem off of Project Euler. Let's have some fun.

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

A SOLUTION USING BASIC PYTHON

```
from __future__ import print_function

total = 0
f1, f2 = 1, 2
while f1 < 4000000:
    if f1 % 2 == 0:
        total = total + f1
    f1, f2 = f2, f1 + f2
print(total)
```

Output: 4613732

Notice we're using
the Python 3.x version of
print here.

Python supports multiple
assignment at once.
Right hand side is fully evaluated
before setting the variables.

FUNCTIONS

A function is created with the def keyword. The statements in the block of the function must be indented.

```
def function_name(args):
    statements
```

The def keyword is followed by the function name with round brackets enclosing the arguments and a colon. The indented statements form a body of the function.

The return keyword is used to specify a list of values to be returned.

```
# Defining the function
def print_greeting():
    print "Hello!"
    print "How are you today?"

print_greeting() # Calling the function
```

```
Hello!
How are you today?
```

FUNCTIONS

All parameters in the Python language are passed by reference.

However, only mutable objects can be changed in the called function.

```
def hello_func(name, somelist):
    print "Hello,", name, "!\n"
    name = "Caitlin"
    mylist[0] = 3
    return 1, 2
myname = "Ben"
mylist = [1,2]
a,b = hello_func(myname, mylist)
print myname, mylist
print a, b
```

```
Hello, Ben !

Ben [3, 2]
1 2
```

FUNCTIONS

What is the output of the following code?

```
def hello_func(names):
    for n in names:
        print "Hello, ", n, "!"
    names[0] = 'Susie’
    names[1] = 'Pete’
    names[2] = 'Will’
names = ['Susan', 'Peter', 'William']
hello_func(names)
print "The names are now ", names, ".\n"
```

FUNCTIONS

What is the output of the following code?

```
def hello_func(names):
    for n in names:
        print "Hello,", n, "!"
    names[0] = 'Susie’
    names[1] = 'Pete’
    names[2] = 'Will’
names = ['Susan', 'Peter', 'William']
hello_func(names)
print "The names are now", names, "."
```

```
Hello, Susan !
Hello, Peter !
Hello, William !
The names are now [‘Susie’, ‘Pete’, ‘Will’] .
```

A SOLUTION WITH FUNCTIONS

The Python interpreter will set some special environmental variables when it starts executing.

If the Python interpreter is running the module (the source file) as the main program, it sets the special __name__ variable to have a value "__main__". This allows for flexibility is writing your modules.

```python
from __future__ import print_function

def even_fib():
    total = 0
    f1, f2 = 1, 2
    while f1 < 4000000:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    print(even_fib())
```

INPUT

- raw_input()
  - Asks the user for a string of input, and returns the string.
  - If you provide an argument, it will be used as a prompt.
- input()
  - Uses raw_input() to grab a string of data, but then tries to evaluate the string as if it were a Python expression.
  - Returns the value of the expression.
  - Dangerous – don’t use it.

```
>>> print(raw_input('What is your name? '))
What is your name? Caitlin
Caitlin
>>> print(input(‘Do some math: ’))
Do some math: 2+2*5
12
```

Note: In Python 3.x, input() is now just an alias for raw_input()

A SOLUTION WITH INPUT

```
from __future__ import print_function

def even_fib(n):
    total = 0
    f1, f2 = 1, 2
    while f1 < n:
        if f1 % 2 == 0:
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total

if __name__ == "__main__":
    limit = raw_input("Enter the max Fibonacci number: ")
    print(even_fib(int(limit)))
```

```
Enter the max Fibonacci number: 4000000
4613732
```

CODING STYLE

So now that we know how to write a Python program, let’s break for a bit to think about our coding style. Python has a style guide that is useful to follow, you can read about PEP 8 here.

I encourage you all to check out pylint, a Python source code analyzer that helps you maintain good coding standards.