LECTURE 2

Python Basics

MODULES

```python
''' Module fib.py '''
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(“Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

So, we just put together our first real Python program. Let’s say we store this program in a file called fib.py.

We have just created a *module*.

Modules are simply text files containing Python definitions and statements which can be executed directly or imported by other modules.

MODULES

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Within a module, the module's name (as a string) is available as the value of the global variable __*name*__.
- If a module is executed directly however, the value of the global variable __*name*__ will be "__main__".
- Modules can contain executable statements aside from definitions. These are executed only the *first* time the module name is encountered in an import statement as well as if the file is executed as a script.

MODULES

I can run our module directly at the command line. In this case, the module's __name__ variable has the value "__main__".

```
$ python fib.py
Max Fibonacci number: 4000000
4613732
```

```
''' Module fib.py '''
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("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

I can import the module into the interpreter. In this case, the value of __name__ is simply the name of the module itself.

```
$ python
>>> import fib
>>> fib.even_fib(4000000)
4613732
```

```
''' Module fib.py '''
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(“Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

I can import the module into the interpreter. In this case, the value of __name__ is simply the name of the module itself.

```
$ python
>>> import fib
>>> fib.even_fib(4000000)
4613732
```

Note that we can only access the definitions of fib as members of the fib object.

```python
''' Module fib.py '''
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("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MODULES

I can import the definitions of the module directly into the interpreter.

```
$ python
>>> from fib import even_fib
>>> even_fib(4000000)
4613732
```

To import *everything* from a module:

```
>>> from fib import *
```

```python
''' Module fib.py '''
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("Max Fibonacci number: ")
    print(even_fib(int(limit)))
```

MINI MODULE QUIZ

I have two modules, foo.py and bar.py.

```python
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```python
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

By convention, all import statements should appear at the top of the .py file.
Let's try to guess the output for each of the following execution methods.

MINI MODULE QUIZ

```python
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```python
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python bar.py
```

What is the output when we execute the bar module directly?

MINI MODULE QUIZ

```python
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```python
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python bar.py
Hi from bar's top level!
bar's __name__ is __main__
```

MINI MODULE QUIZ

```python
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```python
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python foo.py
```

Now what happens when we execute the foo module directly?

MINI MODULE QUIZ

```python
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```python
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python foo.py
Hi from bar's top level!
Hi from foo's top level!
foo's __name__ is __main__
Hello from bar!
```

MINI MODULE QUIZ

```python
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```python
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python
>>> import foo
```

Now what happens when we import the foo module into the interpreter?

MINI MODULE QUIZ

```python
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```python
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python
>>> import foo
Hi from bar's top level!
Hi from foo's top level!
>>> import bar
```

And if we import the bar module into the interpreter?

MINI MODULE QUIZ

```python
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```python
''' Module foo.py'''
import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python
>>> import foo
Hi from bar's top level!
Hi from foo's top level!
>>> import bar
>>>
```

MODULE SEARCH PATH

When a module is imported, Python does not know where it is located so it will look for the module in the following places, in order:

- Built-in modules.
- The directories listed in the sys.path variable. The sys.path variable is initialized from these locations:
  - The current directory.
  - PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
  - The installation-dependent default.

The sys.path variable can be modified by a Python program to point elsewhere at any time.

At this point, we'll turn our attention back to Python functions. We will cover advanced module topics as they become relevant.

MODULE SEARCH PATH

The sys.path variable is available as a member of the sys module. Here is the example output when I echo my own sys.path variable.

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.7/dist-packages/D_Wave_One_Python_Client-
1.4.1-py2.6-linux-x86_64.egg', '/usr/local/lib/python2.7/dist-
packages/PyOpenGL-3.0.2a5-py2.7.egg', '/usr/local/lib/python2.7/dist-
packages/pip-1.1-py2.7.egg', '/usr/local/lib/python2.7/dist-
packages/Sphinx-....
```

FUNCTIONS

We've already know the basics of functions so let's dive a little deeper.

Let's say we write a function in Python which allows a user to connect to a remote machine using a username/password combination. Its signature might look something like this:

```
def connect(uname, pword, server, port):
    print "Connecting to", server, ":", port, "..."
    # Connecting code here ...
```

We've created a function called *connect* which accepts a username, password, server address, and port as arguments (in that order!).

FUNCTIONS

```
def connect(uname, pword, server, port):
    print "Connecting to", server, ":", port, "..."
    # Connecting code here ...
```

Here are some example ways we might call this function:

- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 9160)`
- `connect('jdoe', 'r5f0g87g5@y', 'linprog.cs.fsu.edu', 6370)`

FUNCTIONS

These calls can become a little cumbersome, especially if one of the arguments is likely to have the same value for every call.

Default argument values

- We can provide a default value for any number of arguments in a function.
- Allows functions to be called with a variable number of arguments.
- Arguments with default values must appear at the end of the arguments list!

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

FUNCTIONS

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

Now we can provide a variable number of arguments. All of the following calls are valid:

- `connect('admin', 'ilovecats')`
- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')`
- `connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 6379)`

SURPRISING BEHAVIOR

Let's say I have the following Python module. It defines the *add_item* function whose arguments are *item* and *item_list*, which defaults to an empty list.

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item) # Add item to end of list
    print item_list
```

SURPRISING BEHAVIOR

Let's say I have the following Python module. It defines the *add_item* function whose arguments are *item* and *item_list*, which defaults to an empty list.

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item)
    print item_list
```

```
$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[4, 5]
```

SURPRISING BEHAVIOR

This bizarre behavior actually gives us some insight into how Python works.

```
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item)
    print item_list
```

```
$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[4, 5]
```

Python's default arguments are evaluated *once* when the function is defined, not every time the function is called. This means that if you make changes to a mutable default argument, these changes will be reflected in future calls to the function.

SURPRISING BEHAVIOR

This bizarre behavior actually gives us some insight into how Python works.

```python
''' Module adder.py '''

def add_item(item, item_list = []):
    item_list.append(item)
    print item_list
```

```
$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[4, 5]
```

Arguments are evaluated at this point!

Python's default arguments are evaluated *once* when the function is defined, not every time the function is called. This means that if you make changes to a mutable default argument, these changes will be reflected in future calls to the function.

SURPRISING BEHAVIOR

An easy fix is to use a sentinel default value that tells you when to create a new mutable argument.

```
''' Module adder.py '''

def add_item(item, item_list = None):
    if item_list == None:
        item_list = []
    item_list.append(item)
    print item_list
```

```
$ python
>>> from adder import *
>>> add_item(3, [])
[3]
>>> add_item(4)
[4]
>>> add_item(5)
[5]
```

FUNCTIONS

Consider again our connecting function.

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

The following call utilizes *positional arguments*. That is, Python determines which formal parameter to bind the argument to based on its position in the list.

```
connect('admin', 'ilovecats', 'shell.cs.fsu.edu', 6379)
```

FUNCTIONS

When the formal parameter is specified, this is known as a *keyword argument*.

```
connect(uname='admin', pword='ilovecats',
        server='shell.cs.fsu.edu', port=6379)
```

By using keyword arguments, we can explicitly tell Python to which formal parameter the argument should be bound. Keyword arguments are always of the form *kwarg* = *value*.

If keyword arguments are used they must follow any positional arguments, although the relative order of keyword arguments is unimportant.

FUNCTIONS

Given the following function signature, which of the following calls are valid?

```python
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

1. `connect('admin', 'ilovecats', 'shell.cs.fsu.edu')`
2. `connect(uname='admin', pword='ilovecats', 'shell.cs.fsu.edu')`
3. `connect('admin', 'ilovecats', port=6379, server='shell.cs.fsu.edu')`

FUNCTIONS

Given the following function signature, which of the following calls are valid?

```
def connect(uname, pword, server = 'localhost', port = 9160):
    # connecting code
```

1. `connect('admin', 'ilovecats', 'shell.cs.fsu.edu') -- VALID`

2. `connect(uname='admin', pword='ilovecats', 'shell.cs.fsu.edu') -- INVALID`

3. `connect('admin', 'ilovecats', port=6379, server='shell.cs.fsu.edu') -- VALID`

FUNCTIONS

Parameters of the form *param contain a variable number of arguments within a tuple. Parameters of the form **param contain a variable number of keyword arguments.

```
def connect(uname, *args, **kwargs):
    # connecting code here
```

This is known as *packing*.

Within the function, we can treat args as a list of the positional arguments provided and kwargs as a dictionary of keyword arguments provided.

FUNCTIONS

```
def connect(uname, *args, **kwargs):
    print uname
    for arg in args:
        print arg
    for key in kwargs.keys():
        print key, ":", kwargs[key]

connect('admin', 'ilovecats', server='localhost', port=9160)
```

Output: ?

FUNCTIONS

```
def connect(uname, *args, **kwargs):
    print uname
    for arg in args:
        print arg
    for key in kwargs.keys():
        print key, ":", kwargs[key]

connect('admin', 'ilovecats', server='localhost', port=9160)
```

Output:

```
admin
ilovecats
port : 9160
server : localhost
```

FUNCTIONS

We can use *args and **kwargs not only to define a function, but also to call a function. Let's say we have the following function.

```
def func(arg1, arg2, arg3):
    print "arg1:", arg1
    print "arg2:", arg2
    print "arg3:", arg3
```

FUNCTIONS

We can use *args to pass in a tuple as a single argument to our function. This tuple should contain the arguments in the order in which they are meant to be bound to the formal parameters.

```
>>> args = ("one", 2, 3)
>>> func(*args)
arg1: one
arg2: 2
arg3: 3
```

We would say that we're *unpacking* a tuple of arguments here.

FUNCTIONS

We can use **kwargs to pass in a dictionary as a single argument to our function. This dictionary contains the formal parameters as keywords, associated with their argument values. Note that these can appear in any order.

```
>>> kwargs = {"arg3": 3, "arg1": "one", "arg2": 2}
>>> func(**kwargs)
arg1: one
arg2: 2
arg3: 3
```

LAMBDA FUNCTIONS

One can also define lambda functions within Python.

- Use the keyword *lambda* instead of *def*.
- Can be used wherever function objects are used.
- Restricted to one expression.
- Typically used with functional programming tools – we will see this next time.

```
>>> def f(x):
...     return x**2
...
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
```

LIST COMPREHENSIONS

List comprehensions provide a nice way to construct lists where the items are the result of some operation.

The simplest form of a list comprehension is

```
[expr for x in sequence]
```

Any number of additional for and/or if statements can follow the initial for statement. A simple example of creating a list of squares:

```
>>> squares = [x**2 for x in range(0,11)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

LIST COMPREHENSIONS

Here's a more complicated example which creates a list of tuples.

```
>>> squares = [(x, x**2, x**3) for x in range(0,9) if x % 2 == 0]
>>> squares
[(0, 0, 0), (2, 4, 8), (4, 16, 64), (6, 36, 216), (8, 64, 512)]
```

The initial expression in the list comprehension can be anything, even another list comprehension.

```
>>> [[x*y for x in range(1,5)] for y in range(1,5)]
[[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8, 12, 16]]
```