Python IV

Introductions

- Name
- Role/Job
- Length of tenure
- Programming experience
  - Python specifically
- Why did you sign up?
- Define success for you in this course

What this class is

- Intermediate / Advanced topics in Python
- Lots of hands-on lab work
- Lots of class participation
  - Share your war stories and experiences
- A walk through some real-world usages for the language

What this class isn't

- In depth information on specific Python modules
- Every intermediate/advanced topic in Python
- Rigidly structured

Class Materials

- All the materials for the class are available on GitHub:
  https://github.com/jeremyprice/RU_Python_IV

Pre-Work

Get a copy of the course materials

- Make sure git is installed on your machine
- Clone the GitHub project
- https://github.com/jeremyprice/RU_Python_IV

```
# optional step
sudo yum install git

# clone the repo
git clone https://github.com/jeremyprice/RU_Python_IV
```

System Config

- Make sure Python is installed
  - Python 2 for this course
- Lab machines have CentOS 7 with Python 2 already installed

```
python --version
```

Editor Config

- Install and setup your favorite editor
  - vim, atom.io, emacs, nano, etc.

Virtualenv

- virtualenv allows us to setup isolated python environments
  - customized to your version and dependency requirements
- Comes with distribute, easy_install, and pip
- Allows control of environments where root authority is lacking

```
#install on CentOS 7
sudo yum install -y python-virtualenv

# create a new virtual env
virtualenv <directory_name>

# work in the new virtual env
cd <directory_name>
source bin/activate

# stop working on the virtual env
deactivate
```

PIP

- PIP is a Python package manager
  - built on setuptools package
  - finds packages in the cheese shop
    - https://pypi.python.org/pypi
- Standard package commands:
  - install, uninstall, list, search
- Installation instructions:
  - Already installed by virtualenv!

IPython

- We will use IPython in this class for a lot of our shell interaction
- You can create a virtualenv if desired
  - IPython, and dependencies, won't be the only packages we will install for the course

```
# make sure you are in the virtualenv
pip install ipython
```

Lab

- Grab the materials for the class
- Verify Python version info
- Install virtualenv
- Create a virtualenv
  - activate it
- Install IPython in the virtualenv

Python Review

Anatomy of a Module

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
# Line #1 = shebang
# Line #2 = encoding declaration

"""Docstrings"""
# Inline documentation

Import(s)

Statements
```

Python Keywords

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Tuple Type

- Tuples are sequences of other objects that cannot be changed
  - they are "immutable"
- Tuples can contain any type of object, and can be sliced
  - they're ordered
- A one element tuple is formed by (element,) where the comma makes it a tuple
- Examples:

```
tup = (1, "a", 3)
tup = tuple()
tup = (1,) # Comma needed for single element
gen = (x**2 for x in range(1,4)) # Not a tuple!
```

List Type

- Lists are sequences that can be changed (“mutable”)
- Lists can contain other object types and be sliced
- A list is a set of objects separated by commas and enclosed in square brackets
- Examples:

```
lst = [1, "a", 3]
lst = list()
lst = [x for x in xrange(5)]
```

Dictionary Type

- A dictionary (“dict”) type is a set of expressions that are associated 1 to 1 with other expressions
- A dict provides a handy “mapping” between a “key” expression and its associated “value” expression
- A dict is not an ordered sequence, it’s a mapping.
- Examples:

```
dic = {"a": 1}
dic = dict(a=1)
dic = {x: chr(x) for x in range(97, 107)}
```

Set Type

- A set is a mutable group of unordered immutable objects with unique values
  - no duplicates
  - the set is mutable, but not the objects within it
- Uses the '{}' symbols just like a dictionary, but doesn't have ':' i.e. no key/value pairs
- Slicing is not allowed
  - but iterators work
- Set operations are powerful, but perhaps not the most efficient
- Examples:

```
st = {1,2,2,3}
st = set([1,2,2,3])
st = {x for x in range(1,4)}
```

Yield Statement

- yield is similar to return, but suspends execution of the called function instead of ending the function
  - Saves the state
- On the next call to the function:
  - yield picks up where it left off
  - with all identifier values still holding the same values
- A function with a yield statement is a generator
- Example:

```
def foo(bar=10):
    i = 1
    while i < bar:
        yield i
        i = i + 1
```

With Statement

- The with statement is used to run a suite of other statements under a “context manager”
- Special methods __enter__() and __exit__() are called to setup and takedown a “context”
- Common for doing i/o (which auto-closes the file handle) and threading (which auto-acquires and releases lock)

```
with file("foo.txt", "r") as infile:
    all_txt = infile.read()
```

Import and From Statements

- To make a set of identifiers in another module available for use by your module, you must first use import or from
- import pulls in identifiers from module(s) but the module name must be used as a prefix
  - only module name is added to local namespace
- "from" pulls in identifiers from modules but avoids need to prefix with the module name
  - identifier is added to local namespace

Documentation

- All the docs for version 2:
  - https://docs.python.org/2/
- Standard library:
  - https://docs.python.org/2/library/index.html

Lab - Pyreview

- Follow the instructions in:

`lab_pyreview.py`

REST

REST Principles

- Representational State Transfer
- A simple, client/server web services API currently in favor
- Way of locating and manipulating “resources” (usually XML or JSON documents) on a network
- Commonly based HTTP protocol (GET, POST, PUT, DELETE)
- Stateless (all state on client or server)
- Simple, predictable resource pathing scheme based on URL

More RESTing

- ReSTful services generally map a CRUD interface (Create, Read, Update, Delete) by URL mappings that embed data
  - e.g. GET v2/{tenant_id}/servers
  - HTTP POST -> Create
  - GET -> Read
  - PUT -> Update
  - DELETE -> Delete
- HTTP GET calls to a ReSTful service should not change state i.e. read only

Accessing Web Data: urllib2

- Client functions to access URL's
- `import urllib2`
- urllib2.urlopen(url[,data]) is a common call signature
  - url is the URL of the target resource
  - data is to be sent to the server
- Only HTTP uses data currently
- If data exists
  - it must be URL encoded
  - HTTP GET becomes a POST

urllib2

- A “file-like object” is returned from urlopen() which can be accessed with file semantics (read, readlines, etc.)
- `Raises:`
  `URLError (subclass of IOError)`
  `HTTPError (subclass of URLError)`
- Also takes urllib2.Request objects, useful for including HTTP headers in a dict (or use Request add_header method)
- Can handle redirections, HTTP error responses, cookies, proxies, HTTP authentication

Serialization

JSON

- Javascript Object Notation
- Part of Javascript language definition
- Easy for parse for humans and machines
- Language independent
- “lightweight data interchange format”

```
Name/value pair collection (dict in Python)
Ordered list of values (list in Python)
```

JSON Details

- An “object” is an unordered set of name:value pairs (called “members”) separated by commas and surrounded by curly braces
  - `{ Name1:Value1, Name2:Value1,… }`
- An “array” is a set of “elements” surrounded by square brackets
  - `[ element1, element2,… ]`
- Elements are strings, numbers, true, false, null, object, array

More JSON

- Strings are made of:
  - Unicode chars (\uhhhh)
  - \,\\,/,\b,\f,\n,\r,\t
- Numbers are signed integer, decimal, or exponent (“e”) flavors only
- Whitespace is fine

Python to JSON

- import the json module
- Serialize a Python object to JSON

```
import json
json.dump(python_obj, fo, **kwargs)
json.dumps(python_obj, **kwargs)
```

Note: JSON is not a "framed" protocol
i.e. can't append multiple JSON objects to same file

JSON to Python

- import the json module
- Deserialize a JSON object to Python

```
import json
python_obj = json.load(fo, **kwargs)
python_obj = json.loads(string, **kwargs)
```

JSON Module Caveats

- JSON keys in key/value pairs are always strings. Unlike Python.
- Default json module encoder only encodes “ASCII-compatible” strings. Use u for other encodings
- Same name in name:value pairs uses the last one
- Out of range floats are handled correctly (nan, inf, -inf)

Python to JSON Translations

```
dict -> object
list, tuple -> array
str, unicode -> string
int, float -> number
True, False -> true, false
None -> null
```

Performance

- The builtin json module can be slow
- Other pypi packages have C extensions to speed them up
  - simplejson, yajl, python-cjson, UltraJSON
- Opinion:
  - Don't load/save huge json objects
  - profile your app to make sure it is the json encode/decode

Pickle and Shelve

- The pickle module does a similar job to JSON, but is Python specific
  - Not good for machine data interchange
  - Allows multiple pickled objects to be dumped to the same file
    - must be loaded in same order on way back
- The shelve module essentially provided a persistent dict for pickled objects in a database
- Opinion: Only use these for interchange within your own app

Lab JSON

- Follow the instructions in:
  - `lab_json.py`

Subprocessing

Subprocessing in Python

- Python can spawn and control entire processes using the subprocess module
- Generally means redirecting the basic file descriptors (stdin, stdout, stderr) to gain programmatic access
- Forks a new process and uses pipes for redirection
- As usual, beware of invoking a process based on direct or indirect user input

Subprocess Module

- Contains many convenience functions for creating and managing subprocesses
- subprocess.Popen class is the most full-featured interface
- subprocess.call() is the simplest interface

Subprocess - Details

- call() - spawn a subprocess, wait for completion, return exit code
- check_call() - same as call(), but raises CalledProcessError exception if exit code isn't 0
- check_output() - same as call(), but returns process output as a byte string
- Caveats:
  - Don’t use pipes for stdout or stderr
  - Only use shell=True if need shell features

```
subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False)
subprocess.check_call(args, *, stdin=None, stdout=None, stderr=None, shell=Fals
subprocess.check_output(args, *, stdin=None, stdout=None, stderr=None, shell=Fa

# examples
exit_code = subprocess.call(['ls', '-al'])
exit_code = subprocess.check_call('false')
exit_code = subprocess.check_output(['ls', '-al'])
```

Subprocess - POpen

- Swiss army knife of subprocess spawning
- Can map stderr to stdout using subprocess.STDOUT
- Use subprocess.PIPE in stdin, stdout, stderr
- Check out all the args/kwargs in the docs:
  - https://docs.python.org/2/library/subprocess.html#subprocess.Popen
  - ```
    - poll() sees if child is terminated
    - wait() waits for child to terminate
    - communicate(input=None) sends input to child stdin. Use instead c
    - send_signal(signal) sends given signal to child
    - terminate() and kill() send respective signals to child
    ```

```
# example
subproc = subprocess.POpen(['ls', '-al'])
(stdoutdata, stderrdata) = subproc.communicate(
```
```

Subprocess - Caveats

- Note: if args is a string, must use shell=True to specify args
- Once again, don't use shell=True with user entered data
- shell=False does not call a shell either directly or indirectly, allowing all characters (even metacharacters) to be passed safely

Subprocess - POpen Interaction

- Popen.std* are file objects if std*=PIPE was used in Popen constructor
- Popen.pid is child process ID
- Popen.returncode is child exit code. Set by poll(), wait(). None if child not terminated. If negative, indicates child was terminated by that signal number.

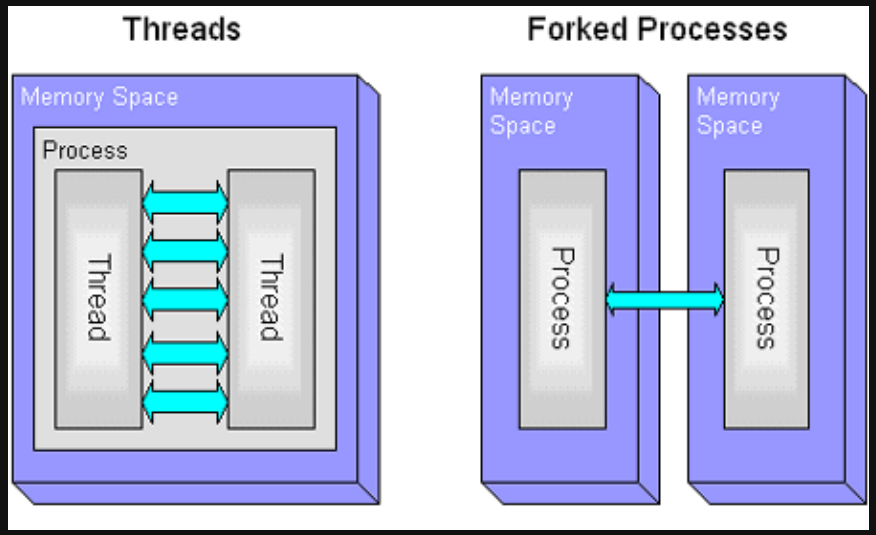
Lab Subprocess

- Follow the instructions in:

`lab_subprocess.py`

Multiprocessing / Multithreading

Threads Vs. Processes



Threads

- Threads are independently scheduled streams of instructions that run within a single OS process
- Threads have their own stack, registers, scheduling attributes, priorities, signal mask, and thread local memory
- Threads allow logical concurrency of execution (and possibly parallel execution if configured)
- Threads introduce the need for synchronization!

Threads

- Threads share the same instructions (bytecodes) same identifier bindings, same open files, and other resources
- Threads cannot exist outside of an OS process
- Threads are “lightweight” – the overhead of creating them is much less than creating a real process
- Threads are used for capturing a higher % of available cycles on a single CPU, realtime systems, an asynchronous event handling.

Threading Models

- Some common threading models exist:
- Boss/worker
  - Boss thread creates worker threads, then loops receiving work requests from a queue or socket
  - Worker threads loop on work assignments from boss
- Peer
  - All threads work on the backlog without a “manager” thread
  - All threads work on a protected/synchronized queue
- Pipeline
  - Each thread does a stage of a work pipeline
    - Each thread accepts work from “previous” thread, passes to “next” thread

Threading Module

- Contains all the necessary primitives to handle threading:
  - threading.Thread() - create a thread from a callable
  - threading.Condition() - create a condition to use for waiting
  - threading.Event() - create an event to use for waiting
  - threading.Lock()/RLock() - Lock a critical section or variable
  - threading.Semaphore() - Classic acquire/release semaphore
  - threading.Timer() - run a callable after timer expires

Running a Thread

```
import threading
import time

def xyz():
    for i in range(10):
        print "thread"
        time.sleep(1)

th = threading.Thread(target=xyz)
th.start()
for i in range():
    print "main"
    time.sleep(1)
```

Threading Module

- Calling another threads join() causes the caller to wait for thread to end
- Pass arguments to new thread using the args and kwargs keywords
- Locks have acquire() and release() methods that can be automatically invoked on the with statement

Threading Module

- Condition variables have a wait() to wait for the condition to be reached
- Condition variables also have notify() and notify_all() that can be used to indicate to other threads that the condition variable has changed
- Semaphore objects also have acquire() and release() for easy use with with
- Event objects have set(), clear(), and wait() which indicate if event has happened or not

Bad Bits about Threads

- The CPython interpreter uses something called the Global Interpreter Lock (GIL) to protect access to the inner workings of the interpreter
- One Python thread at a time can have that lock, thus only one thread at a time can run
- Many threads can be waiting (on data, cpu, etc.)
  - Only one can be active regardless of the number of CPUs/cores/etc.
- There are ways around this, but there is peril therein

Threading Worker Example

```
import Queue
import threading

num_worker_threads = 5

def worker():
    thread_id = threading.current_thread().name
    while True:
        item = q.get()
        print "Worker %s:" % thread_id, item
        q.task_done()

q = Queue.Queue()
for i in range(num_worker_threads):
     t = threading.Thread(target=worker)
     t.daemon = True
     t.start()

work_items = xrange(20)
for item in work_items:
    q.put(item)

q.join()       # block until all tasks are done
```

Multiprocessing

- Multiprocessing module has same interface as threading, but spawns processes instead of threads
- Doesn't suffer from GIL problems of threads
- Higher overhead because it spawns a Python interpreter for each process

Multiprocessing

- Processes can exchange information via Pipes and Queues
- Can synchronize like Threads
- Can share memory amongst processes
- Can create a pool of workers to map tasks to

Running a Process

```
import multiprocessing
import time

def xyz():
    for i in range(10):
        print "thread"
        time.sleep(1)

proc = multiprocessing.Process(target=xyz)
proc.start()
for i in range(10):
    print "main"
    time.sleep(1)
```

Process Worker Example

```
import multiprocessing
import os

def worker(q):
    process_id = os.getpid()
    while True:
        item = q.get()
        print "Worker %s:" % process_id, item
        q.task_done()

if __name__ == "__main__":
    num_worker_processes = 5
    num_work_items = 20
    q = multiprocessing.JoinableQueue(num_work_items+1)
    for i in range(num_worker_processes):
        p = multiprocessing.Process(target=worker, args=[q]
        p.daemon = True
        p.start()

    work_items = xrange(num_work_items)
    for item in work_items:
        q.put(item)

    q.join()       # block until all tasks are done
    print "All done"
```

Lab multiprocessing / multithreading

- Follow the instructions in:

`lab_multi.py`

Objects

Python and Objects

- Everything in Python is an object
- For instance, try the following code:

```
class FakeClass:
    pass

a = 1
b = 'hello'
c = FakeClass()
print a, b, c
```

"Variables" in Python

- In most programming languages, variables are a spot to put values
- In Python, we don't create variables and assign values to them
  - We create names and bind them to objects
- for instance, c = FakeClass() tells Python:
  - "I want to refer that FakeClass instance as c"

More about binding

- Does the following code do what you expect?

```
class FakeClass:
    pass

a = FakeClass()
b = a

print b.hello

a.hello = 'world'

print b.hello
```

Everything is an Object (Including integers)

```
x = 123
print x.__add__
print dir(x)
```

Since everything is just names bound to objects...

```
from __future__ import print_function
import datetime
import imp

print(datetime.datetime.now())
print(datetime.datetime.max, datetime.datetime.min)

class PartyTime():
    def __call__(self, *args):
        imp.reload(datetime)
        value = datetime.datetime(*args)
        datetime.datetime = self
        return value

    def __getattr__(self, value):
        if value == 'now':
            return lambda: print('Party Time!')
        else:
            imp.reload(datetime)
            value = getattr(datetime.datetime, value)
            datetime.datetime = self
            return value

datetime.datetime = PartyTime()
print(datetime.datetime.now())
print(datetime.datetime.max, datetime.datetime.min)
```

Mutable vs. Immutable

- mutable objects **can** be changed after they are created
  - lists, dictionaries
- immutable objects **cannot** be changed after they are created
  - strings, tuples

Immutable Strings?

```
x = 'abc'
print id(x)

x += 'def'
print id(x)
```

Immutable Tuples?

```
class myInt():
    def __init__(self):
        self.value = 0
    def __str__(self):
        return str(self.value)
    def __repr__(self):
        return str(self.value)

x = myInt()
print x, id(x)

x_t = (x, x)
print x_t, id(x_t), id(x_t[0]), id(x_t[1])

x_t[0] = 999 # exception!

x.value = 999
print x, id(x)
print x_t, id(x_t), id(x_t[0]), id(x_t[1])
```

More Names and Binding

```
def list_widget(in_list):
    in_list[0] = 10

    in_list = range(1, 10)
    print in_list
    in_list[0] = 10
    print in_list

my_list = [9, 9, 9, 9]
list_widget(my_list)
print my_list
```

Even Functions are Objects

```
def myFunc():
    x = 1
    y = 2
    z = 'abc'
    return x + y

print myFunc.func_name
print myFunc.func_code.co_varnames
print myFunc.func_code.co_consts
print myFunc.func_code.co_code

import dis
dis.disassemble(myFunc.func_code)
```

Let's Explore Objects

```
# get all the attributes of an object
dir(obj)

# get a specific attribute of an object
# equivalent to: obj.y
x = getattr(obj, 'y')

# check if an object has an attribute named 'y'
hasattr(obj, 'y')
```

Lab

- Follow the instructions in:

`lab_objects.py`

Decorators

Functions

- We all know about functions:

```
def func():
    return "hi"

print func()
```

Scope (namespace)

- Identifiers (names) are looked up in the namespace
- Local to global in steps
- First occurrence found wins

```
info = "hello world"

def localonly():
    print locals()

print globals()
localonly()
```

Scope (globals)

- We can still access global variables, but we have to explicitly reference them

```
info = "hello world"

def localonly():
    print info

localonly()
```

Scope (global shadowing)

- If we try to assign the variable it won't do what we think

```
info = "hello world"

def localonly():
   info = "inside localonly"
   print info

localonly()
print info
```

Variable lifetime

- Variables exist as long as they are in scope

```
def localonly():
   info = "inside localonly"
   print info

localonly()
print info
```

Function parameters and args

- Become local variables to the function

```
def hello(arg1):
   print locals()

hello(1)
```

Function parameters and args

- Have names and positions

```
def hello(arg1, arg2=0):
   print locals()

hello(1)
hello(1, 3)
hello()
hello(arg2=15, arg1=10)
hello(arg2=99)
```

Nested functions

- We can define functions within functions
- Variable lifetime and scoping rules still apply

```
def outside():
    info = 1
    def inside():
        print info
    inside()

outside()
```

Functions are 1st class objects

- (nearly) Everything in Python is an object

```
def func():
    pass

type(func)
func.__class__
issubclass(func.__class__, object)
```

Treating functions as objects

- We can use functions just like integers and other primitives

```
def add(x, y):
    return x + y

def sub(x, y):
    return x + y

def do_math(operation, x, y):
    return operation(x, y)

do_math(sub, 10, 5)
do_math(add, 2, 2)
```

What about inner functions?

- Functions defined in other functions are still just objects

```
def outside():
    def inside():
        print "inside the inner function"
    return inside

my_func = outside()
my_func
my_func()
```

Closures

- Closures are a way to handle scoping and variable lifetimes so the function objects act as expected

```
def outside():
    info = 1
    def inside():
        print info
    return inside

my_func = outside()
my_func.func_closure
my_func()
```

Closures

- Even with function arguments!

```
def outside(arg1):
    def inside():
        print arg1
    return inside

my_func1 = outside(123)
my_func2 = outside('xyz')
my_func1()
my_func2()
```

Decorators (finally)

- Decorators rely on "functions as objects" and closures
- They wrap one function in another

```
def wrapper(passed_in_func):
    def inside():
        print "before passed_in_func"
        retval = passed_in_func()
        return retval + 1
    return inside

def constant():
    return 2

decorated = wrapper(constant)
decorated()
```

Useful Decorator

- Decorators can be used to add bounds checking or other features to our code

```
def add_two_numbers(x, y):
    return x + y

def ensure_positivity(func):
    def inner(x, y):
        return func(abs(x), abs(y))
    return inner

add_two_numbers(-10, 10)

add_two_numbers = ensure_positivity(add_two_

add_two_numbers(-10, 10)
```

Useful Decorator Syntax

- The @ symbol can be used as "syntactic sugar" to apply a decorator to a function

```
def ensure_positivity(func):
    def inner(x, y):
        return func(abs(x), abs(y))
    return inner

@ensure_positivity
def add_two_numbers(x, y):
    return x + y

add_two_numbers(-10, 10)
```

Args and Kwargs

- We can catch all the args to a function in the special * variable (usually called *args)

```
def argless(*args):
    print args

argless()
argless(1,2,3)

def twoargs(x, y, *args):
    print x, y, args

twoargs(1, 2, 3)
twoargs('a', 'b')
```

Args and Kwargs

- We can also apply a list to an argument list

```
def subtract(x, y):
    return x - y


subtract(10, 5)
lst = [10, 5]
subtract(*lst)
```

Args and Kwargs

- Kwargs can be handled the same way - but with ** (usually used as **kwargs)

```
def kwargless(**kwargs):
    print kwargs


kwargless()
kwargless(x=1, y=2)


def subtract(x, y):
    return x - y

values = {'x': 10, 'y': 5}
subtract(**values)
```

Args and Kwargs in Decorators

- We can pass along any args or kwargs we have from the decorator to the wrapped function

```
def debug_call(func):
    def inner(*args, **kwargs):
        print "{} called with {} {}".format(func.func_name, args, kwargs)
        return func(*args, **kwargs)
    return inner

@debug_call
def add(x, y=0):
    return x + y

print add(1, 2)
print add(y=3, x=5)
print add(5)
```

Lab decorators

- Follow the instructions in:

  `lab_decorators.py`

Decorators

with Parameters

Decorator without parameters

```
def debug_call(func):
    def inner(*args, **kwargs):
        print "{} called with {} {}".format(func.func_name, args, kwargs
        return func(*args, **kwargs)
    return inner

@debug_call
def add(x, y=0):
    return x + y

print add(1, 2)
print add(y=3, x=5)
print add(5)
```

What if we wanted to add a parameter to debug_call?

Example

- I want to add a parameter to debug_call that enables or disables the printing
  - (perhaps I could make this a global to turn off/on debugging)

Decorators return functions

```
@debug_call
def add(x, y=0):
    return x + y

# translates into:
add = debug_call(add)
```

Decorators with Parameters return ???

```
@decorator_with_args(arg)
def foo(*args, **kwargs):
    pass

# translates into:
foo = decorator_with_args(arg)(foo)
```

- The decorator then becomes a function that:
  - accepts an argument,
  - and returns a function that:
    - accepts a function that:
      - returns another function

Decorator with Parameter

```
def debug_call(enabled=True):
    def real_debug_call(func):
        def inner(*args, **kwargs):
            if enabled:
                print "{} called with {} {}".format(func.func_name, args, kwargs
            return func(*args, **kwargs)
        return inner
    return real_debug_call

@debug_call()
def add(x, y=0):
    return x + y

@debug_call(enabled=False)
def sub(x, y):
    return x - y

print add(1, 2)
print sub(3, 4)
```

Lab decorators with parameters

- Follow the instructions in:

`lab_decorators_with_parameters.py`

UnitTest

Test Driven Development

- Best early detection of bugs:
  - Pair programming
  - Continuous Integration
  - Test Driven Development
- Test Driven Development (TDD) says essentially:
  - Write a test that for a proposed feature and verify it fails
  - Write the minimal amount of code to make the test pass
  - Refactor
  - Repeat

Unittest Module

- Module in Python used to implement TDD
  - or other unit testing needs
- Derive a class from unittest.TestCase
- For a test runner, unittest.main() is the default, or roll your own
- Simple test to see if two values were multiplied correctly:

```
import calculator.operations.arithmetic as arith
import unittest

class TestCalculations(unittest.TestCase):
    def test_multiply(self):
        """ test multiply 2 * 2 """
        testVal = arith.mult(2,2)
        self.assertEqual(testVal, 4)

if __name__ == "__main__":
    unittest.main()
```

Unittest Module

- If a test fixture is needed to setup/takedown the calculator for each test, we can add methods setUp() and tearDown()
- The naming standard is that each test method name start with test
- For unit tests, the TestCase class offers (3) primary verifiers:
  - assertEqual(): result is equal to the value we expect
  - assertTrue(): result is a True assertion
  - assertRaises(): result is the Exception we expect

Unittest Test

- There are more assert methods to make life easy:
  - assertNotEqual(): a != b
    assertTrue(): a is True
    assertFalse(): a is False
    assertIs(): a is b
    assertIsNot(): a is not b
    assertIsNone(): a is None
    assertIsNotNone(): a is not None
    assertIn(): a in b
    assertNotIn(): a not in b
    assertIsInstance(): isinstance(a, b) is True
    assertIsNotInstance(): isinstance(a, b) is False
    assertGreater(a, b): a > b
    assertRaises(ex): raises ex
    assertRegexMatches(x, re): regex.search(x) is True

Lab

- Follow the instructions in:

`lab_unittest.py`

- Register your completion in Socrative

Mock

What is a Mock?

- Let's say you have some code that talks to a database
  - If you implement unit tests you probably don't want your code to talk to the actual database
- A Mock is an object that looks like the thing your are testing against
  - It doesn't have all the underlying logic to DO anything
  - Think of it like an actor
    - "I'm not a real database, I just play one on TV"

Why use Mocks?

- Maybe your resource isn't available
- Maybe you can't force an error on the resource to test how your code will respond
- Maybe it is too costly to use a real resource
- Maybe the real resource takes too long
  - Not parallel

Drawbacks to Mocks

- Building an accurate Mock is time consuming and difficult
  - Has to track the resource changes
- You built the Mock, and the code, pollution is possible

Mock

- Mock is a library for mocking objects in Python
- It fits well with the unittest library
- Mock and MagicMock objects create attributes and methods as you use them
  - store details about how they've been used
- Mock is available in Python 3 (> 3.3) and has backports available for Python 2

```
pip install mock
```

Mock Example

```
# in Python 2
from mock import MagicMock
# in Python 3
# from unittest.mock import MagicMock
thing = ProductionClass()
thing.method = MagicMock(return_value=3)
thing.method(3, 4, 5, key='value')
3
thing.method.assert_called_with(3, 4, 5, key='value')
```

Side Effects

- You can have the mocked object create side effects

```
from mock import Mock

thing = Mock(side_effect=KeyError('bam'))
thing()
```

Meta Classes