Python Training

A **basic** overview

Functions, Modules & Packages

Functions

- Built-in functions
- Lambda functions

Modules

- What are modules?
- Import statements

Packages

Functions… cont.

Call by value for primitive data types

- **Call by reference for derived data types**
  - Q: Why?
  - A: Reference Semantics

Functions: Parameter passing

```
def hello(greeting='Hello', name='world'):
     print ('%s, %s!' % (greeting, name))

hello('Greetings')
```

Adding default values to parameters

```
def hello_1(greeting, name):
     print ('%s, %s!' % (greeting, name))
# The order here doesn't matter at all:
hello_1(name='world', greeting='Hello')
```

Using named parameters. In this case the order of the arguments does not matter.

```
def print_params(*params):
     print (params)

print_params('Testing')
print_params(1, 2, 3)
```

The variable length function parameters allow us to create a function which can accept any number of parameters. *params is treated as tuple

```
def print_params_3(**params):
     print (params)

print_params_3(x=1, y=2, z=3)
```

Variable named parameters

```
def print_params_4(x, y, z=3, *pospar, **keypar):
     print (x, y, z)
     print (pospar)
     print (keypar)

print_params_4(1, 2, 3, 5, 6, 7, foo=1, bar=2)
print_params_4(1, 2)
```

A combination of all of above cases

Built-in functions

abs()	divmod()	***input()***	open()	staticmethod()
all()	***enumerate()***	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	***filter()***	len()	***range()***	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	***reduce()***	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	***map()***	repr()	xrange()
cmp()	globals()	max()	reversed()	***zip()***
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

Lambda functions

- **Unnamed functions**
- **Mechanism to handle function objects**
- **To write inline simple functions**
- **Generally used along with maps, filters on lists, sets etc.**
- **Not as powerful as in C++11, Haskell etc. e.g. no looping etc.**
- **Example: lambda x,y : x+y to add two values**

Example – Lambda functions

```
>>> def make_incrementor(n):
 ... return lambda x: x + n
... >>> f = make_incrementor(42)
>>> f(0)(42) >>> f(1)(43)
```

The above example uses a lambda expression to return a function.
Another use is to pass a small function as an argument:

```
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Modules

- **A module is a file containing Python definitions and statements intended for use in other Python programs.**
- **It is just like any other python program file with extension .py**
- **Use the "import <module>" statement to make the definitions in <module> available for use in current program.**
- **A new file appears in this case \path\<module>.pyc. The file with the .pyc extension is a compiled Python file for fast loading.**
- **Python will look for modules in its system path. So either put the modules in the right place or tell python where to look!**

```
import sys
sys.path.append('c:/python')
```

Modules

- **Three import statement variants**

`import math`
`x = math.sqrt(10)`
 	
`import math as m`
`print  m.pi`
	Here just the single identifier math is added to the current namespace. If you want to access one of the functions in the module, you need to use the dot notation to get to it.

`from math import cos, sin, sqrt`
`x = sqrt(10)`
	The names are added directly to the current namespace, and can be used without qualification.

`from math import *`
`x = sqrt(10)`
	This will import currentall the identifiers from module into the namespace, and can be used without qualification.

Packages

- **Packages are used to organize modules. While a module is stored in a file with the file name extension .py, a package is a directory.**

- **To make Python treat it as a package, the folder must contain a file (module) named __init__.py**

File/Directory	Description
~/python/	Directory in PYTHONPATH
~/python/drawing/	Package directory (drawing package)
~/python/drawing/__init__.py	Package code ("drawing module")
~/python/drawing/colors.py	colors module
~/python/drawing/shapes.py	shapes module
~/python/drawing/gradient.py	gradient module
~/python/drawing/text.py	text module
~/python/drawing/image.py	image module

Scope of variable

- **Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:**
  - **the innermost scope, which is searched first, contains the local names**
  - **the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names**
  - **the next-to-last scope contains the current module's global names**
  - **the outermost scope (searched last) is the namespace containing built-in names**

Scope of variable

- If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. To rebind variables found outside of the innermost scope, the nonlocal statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Working with Files

- **Python supports both free form and fixed form files – text and binary**
- **open() returns a file object, and is most commonly used with two arguments: open(filename, mode)**
- **Modes:**

Value	Description
'r'	Read mode
'w'	Write mode
'a'	Append mode
'b'	Binary mode (added to other mode)
'+'	Read/write mode (added to other mode)

- **f = open(r'C:\text\somefile.txt')**
- **For Input/Output: read(), readline(), write() and writeline()**

Working with Files

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

File handling in Python

- **With statement**
  - The advantage is that the file will be automatically closed after the indented block after the with has finished execution

- **with statement for reading and writing files.**

```
with open("example.txt", "w") as fh:
    fh.write("To write or not to write\nthat is the question!\n")

with open("ad_lesbiam.txt") as fobj:
        for line in fobj:
                print(line.rstrip())
```

File handling

- Copy a file

```
fobj_in = open("example.txt")
fobj_out = open("excopy.txt","w")
i = 1
for line in fobj_in:
    print(line.rstrip())
    fobj_out.write(str(i) + ": " + line)
    i = i + 1
fobj_in.close()
fobj_out.close()
```

File Handling

- Read entire file at one go using readlines
  - The complete file is read into the list flist.
  - We can acces e.g. the 3rd line with flist[2].

```
flist = open(“example.txt").readlines()
 print(flist)
```

- Another way to read in a -   using read().
  - With this method we can read the complete file into a string

```
fstr = open(“example.txt").read()
```

To display characters from 10 to 20

```
      print(fstr[10:20])
```

File Handling

- To set - or reset - a file's position to a certain position

```
>>> fh = open(“myfile.txt")
>>> fh.tell()
0
>>> fh.read(7)
>>> fh.tell()
7
fh.read()
', abcde fghjklmn
>>> fh.tell()
22
>>> fh.seek(9)
9
>>> fh.read(5)
abcde
```

File Handling

- **To set the current position 6 characters to the left:**

```
>>> fh.seek(fh.tell() -6)
```

- **To set 4 characters to right**

```
>>> fh.seek(fh.tell() + 4)
```

Classes & Objects

- Python is an object-oriented programming language, which means that it provides features that support object-oriented programming (OOP).

- Sample class definition

```
class Point:
        """ Point class represents and manipulates x,y coords. """
        def __init__(self):
            """ Create a new point at the origin """
            self.x = 0
            self.y = 0
p = Point()
print p.x, p.y
```

- Constructor: In Python we use __init__ as the constructor name

```
def __init__(self):                 # a = Point()
def __init__(self, x=0, y=0):       # a = Point(5, 6)
```

Classes & Objects

- **Methods**

```
class Point:
        """ Point class represents and manipulates x,y coords. """
        def __init__(self, x=0): self.x = x
        def x_square(self): return self.x ** 2

p = Point(2)
print p.x_square()
```

- **Objects are mutable.**

Classes & Objects

- **Operator Overloading**

```
class Point:
   def __init__(self, x=0, y=0):
      self.x = x
      self.y = y
   def __add__(self, other):
      return Point(self.x + other.x, self.y + other.y)
   def __mul__(self, other):
      if isinstance(other, Point):
         return Point(self.x * other.x, self.y * other.y)
      else:
         return Point(self.x * other, self.y * other)
   def __rmul__(self, other):
      return Point(self.x * other, self.y * other)
   def __repr__(self):
      return "({0}, {1})".format(self.x, self.y)

p1 = Point(2,3)
p2 = Point(3,4)
print p1 + p2                  #prints (5, 7)
print p1 * p2                  #prints (6, 12)
print p1 * 2  #prints (4, 6)
print 2 * p2  #prints (6, 8)
```

- class Dog:

-     kind = 'canine'   # class variable shared by all instances

-     def __init__(self, name):
-         self.name = name   # instance variable unique to each instance

- >>> d = Dog('Fido')
- >>> e = Dog('Buddy')
- >>> d.kind   # shared by all dogs
- 'canine'
- >>> e.kind   # shared by all dogs
- 'canine'
- >>> d.name   # unique to d
- 'Fido'
- >>> e.name   # unique to e
- 'Buddy'

```
class Dog:

    tricks = []         # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks            # unexpectedly shared by all dogs
['roll over', 'play dead']
```

- class Dog:

-     def __init__(self, name):
-         self.name = name
-         self.tricks = []   # creates a new empty list for each dog

-     def add_trick(self, trick):
-         self.tricks.append(trick)

- >>> d = Dog('Fido')
- >>> e = Dog('Buddy')
- >>> d.add_trick('roll over')
- >>> e.add_trick('play dead')
- >>> d.tricks
- ['roll over']
- >>> e.tricks
- ['play dead‘]

Classes & Objects: Operator Overloading

Operator	Special method	Operator	Special method
self + other	__add__(self, other)	+self	__pos__(self)
self - other	__sub__(self, other)	abs(self)	__abs__(self)
self * other	__mul__(self, other)	~self	__invert__(self) (bitwise)
self / other	__div__(self, other) or __truediv__(self,other) if __future__.division is active.	self += other	__iadd__(self, other)
self // other	__floordiv__(self, other)	self -= other	__isub__(self, other)
self % other	__mod__(self, other)	self *= other	__imul__(self, other)
divmod(self,other)	__divmod__(self, other)	self /= other	__idiv__(self, other) or __itruediv__(self,other) if __future__.division is in effect.
self ** other	__pow__(self, other)	self //= other	__ifloordiv__(self, other)
self & other	__and__(self, other)	self %= other	__imod__(self, other)
self ^ other	__xor__(self, other)	self **= other	__ipow__(self, other)
self | other	__or__(self, other)	self &= other	__iand__(self, other)
self << other	__lshift__(self, other)	self ^= other	__ixor__(self, other)
self >> other	__rshift__(self, other)	self |= other	__ior__(self, other)
bool(self)	__nonzero__(self) (used in boolean testing)	self <<= other	__ilshift__(self, other)
-self	__neg__(self)	self >>= other	__irshift__(self, other)

Right-hand-side equivalents for all **binary** operators exist (__**r**add__, __**r**sub__, __**r**mul__, __**r**div__, ...).
They are called when class instance is on r-h-s of operator:
-- a + 3 calls __add__(a, 3)   -- 3 + a calls __radd__(a, 3)

Classes & Objects: Special methods for any class

Method	Description
__init__(self, args)	Instance initialization (on construction)
__del__(self)	Called on object demise (refcount becomes 0)
__repr__(self)	repr() and \`...\` conversions
__str__(self)	str() and print statement
__sizeof__(self)	Returns amount of memory used by object, in bytes (called by sys.getsizeof()).
__format__(self, format_spec)	format() and str.format() conversions
__cmp__(self,other)	Compares self to other and returns <0, 0, or >0. Implements >, <, == etc...
__index__(self)	Allows using any object as integer index (e.g. for slicing). Must return a single integer or long integer value.
__lt__(self, other)	Called for self < other comparisons. Can return anything, or can raise an exception.
__le__(self, other)	Called for self <= other comparisons. Can return anything, or can raise an exception.
__gt__(self, other)	Called for self > other comparisons. Can return anything, or can raise an exception.
__ge__(self, other)	Called for self >= other comparisons. Can return anything, or can raise an exception.
__eq__(self, other)	Called for self == other comparisons. Can return anything, or can raise an exception.
__ne__(self, other)	Called for self != other (and self <> other) comparisons. Can return anything, or can raise an exception.

Classes & Objects: Special methods for any class (contd…)

Method	Description
__hash__(self)	Compute a 32 bit hash code; hash() and dictionary ops. Since 2.5 can also return a long integer, in which case the hash of that value will be taken.Since 2.6 can set __hash__ = None to void class inherited hashability.
__nonzero__(self)	Returns 0 or 1 for truth value testing. when this method is not defined, __len__() is called if defined; otherwise all class instances are considered "true".
__getattr__(self,name)	Called when attribute lookup doesn't find name. See also __getattribute__.
__getattribute__(self, name)	Same as __getattr__ but always called whenever the attribute name is accessed.
__dir__(self)	Returns the list of names of valid attributes for the object. Called by builtin function dir(), but ignored unless __getattr__or __getattribute__ is defined.
__setattr__(self, name, value)	Called when setting an attribute (inside, don't use "self.name = value", use instead "self.__dict__[name] = value")
__delattr__(self, name)	Called to delete attribute <name>.
__call__(self, *args, **kwargs)	Called when an instance is called as function: obj(arg1, arg2, ...) is a shorthand for obj.__call__(arg1, arg2, ...).
__enter__(self)	For use with context managers, i.e. when entering the block in a with-statement. The with statement binds this method's return value to the as object.
__exit__(self, type, value, traceback)	When exiting the block of a with-statement. If no errors occured, type, value, traceback are None. If an error occured, they will contain information about the class of the exception, the exception object and a traceback object, respectively. If the exception is handled properly, return True. If it returns False, the with-block re-raises the exception.

Classes & Objects

- **Inheritance / Sub-classing**
  - We can create a class by inheriting all features from another class.

The "hello" method defined in class A will be inherited by class B.
 	
The output will be:
Hello, I'm A.
Hello, I'm A.
	`class A:`
	`    def hello(self):`
	`        print "Hello, I'm A."`
	`class B(A):`
	`    pass`
	`a = A()`
	`b = B()`
	`a.hello()`
	`b.hello()`

  - Python supports a limited form of multiple inheritance as well.
    - class DerivedClassName(Base1, Base2, Base3):
  - Derived classes may **override methods** of their base classes.

Exception Handling

- **Whenever a runtime error occurs, it creates an exception object. For example:**

```
  >>> print(55/0)
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- **In python, the basic syntax of exception handling is**

```
  try:
some code to raise exception
  except ExceptionClassName:
exception handler statements
```

- **Example**

```
try:
   1/0
except ZeroDivisionError:
   print "Can't divide anything by zero."
```

Exception Handling

➢ **Below is a list of some of the built-in exceptions**

Class Name	Description
Exception	The root class for all exceptions
AttributeError	Raised when attribute reference or assignment fails
IOError	Raised when trying to open a nonexistent file (among other things)
IndexError	Raised when using a nonexistent index on a sequence
KeyError	Raised when using a nonexistent key on a mapping
NameError	Raised when a name (variable) is not found
SyntaxError	Raised when the code is ill-formed
TypeError	Raised when a built-in operation or function is applied to an object of the wrong type
ValueError	Raised when a built-in operation or function is applied to an object with correct type, but with an inappropriate value
ZeroDivisionError	Raised when the second argument of a division or modulo operation is zero

Exception Handling

- **Catch more than one exception**
  - except (ExceptionType1, ExceptionType2, ExceptionType3):
- **Handle multiple exceptions one-by-one**
  - except ExceptionType1: <code>
  - except ExceptionType2: <code>
- **Catch all exceptions**
  - except:
- **Capture the exception object**
  - except ExceptionType as e:

- **Use the raise statement to throw an exception**

  raise ValueError(“You’ve entered an incorrect value”)

- **The finally clause of try is used to perform cleanup activities**