Programming in Python – Lecture#2

Adeel-ur-Rehman

Scheme of Lecture

- Modules and Packages
- Default Arguments and Keyword Arguments
- Lambda Functions and Documentation Strings
- Object Oriented Framework
- Python Namespaces and Scopes
- Classes, Objects, Methods
- Iterators

Modules

- We can reuse code in our program by defining functions once.
- What if we want to reuse a number of functions in other programs we write?
- The solution is modules.
- A module is basically a file containing all our functions and variables that we have defined.
- The filename of the module **must** have a **.py** extension.
- A module can be *imported* by another program to make use of its functionality.

The sys Module

- sys stands for **Sys**tem
- The sys module contains system-level information, like:
  - The version of Python we are running.
  - i.e., (sys.version or sys.version_info),
  - And system-level options like the maximum allowed recursion
  - i.e., depth (sys.getrecursionlimit() and sys.setrecursionlimit()).

Using sys module

```
# A use of sys module
import sys
print 'The command line arguments used are:'
for i in sys.argv: # list of command-line args
    print i
print '\n\nThe PYTHONPATH is', sys.path, '\n'
```

The os module

- os stands for **O**perating **S**ystem.
- The os module has lots of useful functions for manipulating files and processes – the core of an operating system.
- And os.path has functions for manipulating file and directory paths.

Using os module

```
# A use of os module
import os
print os.getcwd()
os.chdir("/dev")
print os.listdir(os.getcwd())
print os.getpid()
print os.getppid()
print os.getuid()
print os.getgid()
```

The Module Search Path

- When a module named **os** is imported, the interpreter searches for a file named 'os.py' in the:
  - Current directory
  - In the list of directories specified by the environment variable PYTHONPATH.
  - In an installation-dependent default path;
    - on UNIX, this is usually '.:/usr/local/lib/python'.

The Module Search Path

- Actually, modules are searched in the list of directories given by the variable sys.path
  - Which is initialized from the directory containing the input script (or the current directory).
  - PYTHONPATH
  - Installation-dependent default.

Byte-Compiled .pyc files

- Importing a module is a relatively costly affair.
- So Python does some optimizations to create byte-compiled files with the extension .pyc
- If you import a module such as, say, module.py, then Python creates a corresponding byte-compiled module.pyc
- This file is useful when you import the module the next time (even from a different program)
  - i.e., it will be much faster.
  - These byte-compiled files are platform-independent.

The from.. import statement

- If we want to directly import the argv variable into our program, then we can use the from sys import argv statement.
- If we want to import all the functions, classes and variables in the sys module, then we can use the from sys import * statement.
- This works for any module.
- In general, avoid using the from..import statement and use the import statement instead since our program will be much more readable that way.

Packages

- Packages are a way of structuring Python's module namespace by using "dotted module names".
- For example, the module name A.B designates a submodule named 'B' in a package named 'A'.
- Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.
- Suppose we want to design a collection of modules (a "package") for the uniform handling of sound files and sound data.
- There are many different sound file formats usually recognized by their extensions. For example:
- '.wav', '.aiff', '.au', so we may need to create and maintain a growing collection of modules for the conversion between the various file formats.

Packages

- There are also many different operations we might want to perform on sound data:
  - Mixing
  - Adding echo
  - Applying an equalizer function
  - Creating an artificial stereo effect,
- We will be writing a never-ending stream of modules to perform these operations.
- Here's a possible structure for our package (expressed in terms of a hierarchical filesystem):

Packages

- Sound/ — Top-level package
  - Formats/ — Subpackage for file format conversions
    - wavread.py
    - wavwrite.py
    - aiffread.py
    - aiffwrite.py
    - auread.py
    - auwrite.py
    - ...
  - Effects/ — Subpackage for sound effects
    - echo.py
    - surround.py
    - reverse.py
    - ...

Packages

- Filters/ Subpackage for filters
  - equalizer.py
  - vocoder.py
  - karaoke.py
  - ...
- When importing the package, Python searches through the directories on sys.path looking for the package subdirectory.

Packages

- Users of the package can import individual modules from the package, for example:
- import Sound.Effects.echo
- This loads the submodule Sound.Effects.echo
- It must be referenced with its full name.
  - Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
- An alternative way of importing the submodule is:
  - from Sound.Effects import echo

Packages

- This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:
- echo.echofilter(input, output, delay=0.7, atten=4)
- Yet another variation is to import the desired function or variable directly:
  - from Sound.Effects.echo import echofilter
- Again, this loads the submodule echo, but this makes its function echofilter() directly available:
- echofilter(input, output, delay=0.7, atten=4)

Packages

- Note that when using from *package* import *item*, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable.
- The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it.
- If it fails to find it, an ImportError exception is raised.
- Contrarily, when using syntax like import *item.subitem.subsubitem*, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

Intra-Package References

- The submodules often need to refer to each other.
- For example, the surround module might use the echo module.
- In fact, such references are so common that the import statement first looks in the containing package before looking in the standard module search path.
- Thus, the surround module can simply use import echo or from echo import echofilter.
- If the imported module is not found in the current package (the package of which the current module is a submodule), the import statement looks for a top-level module with the given name.

Intra-Package References

- When packages are structured into subpackages (as with the Sound package in the example), there's no shortcut to refer to submodules of sibling packages - the full name of the subpackage must be used.
- For example, if the module Sound.Filters.vocoder needs to use the echo module in the Sound.Effects package, it can use from Sound.Effects import echo.

Default Argument Values

- For some functions, we may want to make some parameters as *optional.*
- In that case, we use default values if the user does not want to provide values for such parameters.
- This is done with the help of default argument values.
- We can specify default argument values for parameters by following the parameter name in the function definition with the assignment operator (=) followed by the default argument.

Using Default Argument Values

```
# Demonstrating default arg. values
def say(s, times = 1):
    print s * times
say('Hello')
say('World', 5)
```

Using Default Argument Values

- Only those parameters which are at the end of the parameter list can be given default argument values.
- i.e. we cannot have a parameter with a default argument value before a parameter without a default argument value, in the order of parameters declared, in the function parameter list.
- This is because values are assigned to the parameters by position.
- For example:
  - def func(a, b=5) is valid
  - but def func(a=5, b) is *not valid*.

Keyword Arguments

- If we have some functions with many parameters and we want to specify only some parameters, then we can give values for such parameters by naming them.
- i.e., this is called keyword arguments. We use the name instead of the position which we have been using all along.
- This has two advantages:
  - Using the function is easier since we do not need to worry about the order of the arguments.
  - We can give values to only those parameters which we want, provided that the other parameters have default argument values.

Using Keyword Arguments

```
# Demonstrating Keyword Arguments
def func(a, b=5, c=10):
   print 'a is', a, 'and b is', b, 'and c is', c
func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Lambda Forms

- Python supports an interesting syntax that lets you define one-line mini-functions on the fly.
- Borrowed from Lisp, these so-called lambda functions can be used anywhere a function is required.
- Have a look at an example:

Using Lambda Functions

```
>>> def f(x):
... return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2
>>> g(3)
6
>>> (lambda x: x*2)(3)
6
>>> def f(n):
…return lambda x: x+n
>>> v = f(3)
>>> v(10)
13
```

Using Lambda Functions

- This is a lambda function that accomplishes the same thing as the normal function above it.
- Note the abbreviated syntax here:
  - there are no parentheses around the argument list
  - and the return keyword is missing (it is implied, since the entire function can only be one expression).
  - Also, the function has no name
  - But it can be called through the variable it is assigned to.

Using Lambda Functions

- We can use a lambda function without even assigning it to a variable.
- It just goes to show that a lambda is just an in-line function.
- To generalize, a lambda function is a function that:
  - takes any number of arguments and returns the value of a single expression
  - lambda functions can not contain commands
  - and they can not contain more than one expression.
  - Don't try to squeeze too much into a lambda function; if needed something more complex, define a normal function instead and make it as long as wanted.

Documentation Strings

- Python has a nifty feature called documentation strings which are usually referred to by their shorter name docstrings.
- DocStrings are an important tool that we should make use of since it helps to document the program better.
- We can even get back the docstring from a function at runtime i.e. when the program is running.

Using Documentation Strings

```
def printMax(x, y):
    '''Prints the maximum of the two numbers.

       The two values must be integers. If they are
       floating point numbers, then they are converted to
       integers.'''
    x = int(x) # Convert to integers, if possible
    y = int(y)
    if x > y:
        print x, 'is maximum'
    else:
        print y, 'is maximum'
    printMax(3, 5)
    print printMax.__doc__
```

Using Documentation Strings

- A string on the first logical line of a function is a docstring for that function.
- The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot.
- Then the second line is blank followed by any detailed explanation starting from the third line.
- *It is strongly advised* to follow such a convention for all our docstrings for all our functions.
- We access the docstring of the printMax function using the __doc__ attribute of that function.

Object-Oriented Framework

- Two basic programming paradigms:
  - Procedural
    - Organizing programs around functions or blocks of statements which manipulate data.
  - Object-Oriented
    - combining data and functionality and wrap it inside what is called an object.

Object-Oriented Framework

- Classes and objects are the two main aspects of object oriented programming.
- A **class** creates a new *type*.
- Where **objects** are *instances* of the class.
- An analogy is that we can have variables of type int which translates to saying that variables that store integers are variables which are instances (objects) of the int class.

Object-Oriented Framework

- Objects can store data using ordinary variables that *belong* to the object.
- Variables that belong to an object or class are called as fields.
- Objects can also have functionality by using functions that *belong* to the class. Such functions are called methods.
- This terminology is important because it helps us to differentiate between a function which is separate by itself and a method which belongs to an object.

Object-Oriented Framework

- Remember, that fields are of two types
  - they can belong to each instance (object) of the class
  - or they belong to the class itself.
  - They are called instance variables and class variables respectively.
- A class is created using the class keyword.
- The fields and methods of the class are listed in an indented block.

Python Scopes and Namespaces

- A namespace is a mapping from names to objects.
- Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way.
- Examples of namespaces are:
  - the set of built-in names (functions such as abs(), and built-in exception names)
  - the global names in a module,
  - and the local names in a function invocation.

Python Scopes and Namespaces

- A scope is a textual region of a Python program where a namespace is directly accessible.
- "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

Python Scopes and Namespaces

- 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 namespaces of any enclosing functions,

Python Scopes and Namespaces

- which are searched starting with the nearest enclosing scope; the middle scope, searched next, contains the current module's global names;
- and the outermost scope (searched last) is the namespace containing built-in names.

The self

- Class methods have only one specific difference from ordinary functions
  - they have an extra variable that has to be added to the beginning of the parameter list
  - but we do **not** give a value for this parameter when we call the method.
  - this particular variable refers to the object itself,
  - and by convention, it is given the name self.

The self

- Although, we can give any name for this parameter, it is *strongly recommended* that we use the name self.
- Any other name is definitely frowned upon.
- There are many advantages to using a standard name
  - any reader of our program will immediately recognize that it is the object variable i.e. the self and even specialized IDEs (Integrated Development Environments such as Boa Constructor) can help us if we use this particular name.

The self

- Python will automatically provide this value in the function parameter list.
- For example, if we have a class called **MyClass** and an instance (object) of this class called **MyObject**, then when we call a method of this object as MyObject.method(arg1, arg2), this is automatically converted to MyClass.method(MyObject, arg1, arg2).
- This is what the special self is all about.

The __init__ method

- __init__ is called immediately after an instance of the class is created.
- It would be tempting but incorrect to call this the constructor of the class.
  - Tempting, because it looks like a constructor (by convention, __init__ is the first method defined for the class), acts like one (it's the first piece of code executed in a newly created instance of the class), and even sounds like one ("init" certainly suggests a constructor-ish nature).

The __init__ method

- Incorrect, because the object has already been constructed by the time __init__ is called, and we already have a valid reference to the new instance of the class.

- But __init__ is the closest thing we're going to get in Python to a constructor, and it fills much the same role.

Creating a Class

```
class Person:
    pass # A new block
p = Person()
print p
#<__main__.Person instance at 0x816a6cc>
```

Object Methods

```
class Person:
    def sayHi(self):
        print 'Hello, how are you?'
p = Person()
p.sayHi()
# This short example can also be
  #written as Person().sayHi()
```

Class and Object Variables

```
class Person:
    '''Represents a person.'''
    population = 0
    def __init__(self, name):
        '''Initializes the person.'''
        self.name = name
        print '(Initializing %s)' % self.name
        # When this person is created, # he/she adds to the population
        Person.population += 1
    def sayHi(self):
        '''Greets the other person. Really, that's all it does.'''
        print 'Hi, my name is %s.' % self.name
```

Class and Object Variables

```
    def howMany(self):
        '''Prints the current population.'''
         # There will always be at least one person
        if Person.population == 1:
            print 'I am the only person here.'
        else:
            print 'We have %s persons here.' % Person.population
adeel = Person('Adeel')
adeel.sayHi()
adeel.howMany()
kalam = Person('Abdul Kalam')
kalam.sayHi()
kalam.howMany()
adeel.sayHi()
adeel.howMany()
```

Iterators

- By now, you've probably noticed that most container objects can looped over using a for statement:
  - for element in [1, 2, 3]:
    print element
  - for element in (1, 2, 3):
    print element
  - for key in {'one':1, 'two':2}:
    print key
  - for char in "123":
    print char

Iterators

- This style of access is clear, concise, and convenient.
- The use of iterators pervades and unifies Python.
- Behind the scenes, the for statement calls iter() on the container object.
- The function returns an iterator object that defines the method next() which accesses elements in the container one at a time.
- When there are no more elements, next() raises a StopIteration exception which tells the for loop to terminate.
- This example shows how it all works:

Iterators

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
```

Iterators

```
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
File "<pyshell#6>", line 1, in -toplevel
  it.next()
StopIteration
```