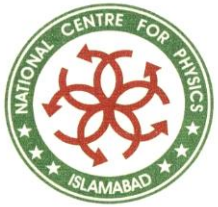


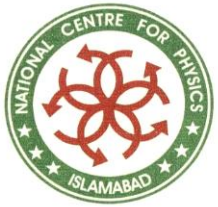
Programming Python – Lecture#2

Mr. Adeel-ur-Rehman

ASC, National Centre for Physics

Scheme of Lecture

- More on Functions
- More on Lists
- Comparing different Data Structures
- Using Packages
- Formatting I/O
- Pickle Module



More on Functions...

- Default Argument Values
- Lambda Forms
- Documentation Strings

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.

More on Lists...

- Important Built-in Functions of Lists
- Using Lists as Stacks and Queues
- The del Statement

List's Important Built-in Functions

- Here are some important functions of lists:
  - **append**(*x*)
    - Add an item to the end of the list
  - **extend**(*L*)
    - Extend the list by appending all the items in the given list
  - **insert**(*i, x*)
    - Insert an item at a given position.
    - The first argument is the index of the element before which to insert,
    - so a.insert(0, *x*) inserts at the front of the list,
    - and a.insert(len(a), *x*) is equivalent to a.append(*x*).
  - **remove**(*x*)
    - Remove the first item from the list whose value is *x*.
    - It is an error if there is no such item.

List's Important Built-in Functions

- **pop**([*i*])
  - Remove the item at the given position in the list, and return it.
  - If no index is specified, a.pop() returns the last item in the list. The item is also removed from the list.
  - (The square brackets around the *i* in the method signature denote that the parameter is optional, not that we should not type square brackets at that position.)
- **index**(*x*)
  - Return the index in the list of the first item whose value is *x*.
  - It is an error if there is no such item.

List's Important Built-in Functions

- **count**(*x*)
  - Return the number of times *x* appears in the list.
- **sort**()
  - Sort the items of the list, in place.
- **reverse**()
  - Reverse the elements of the list, in place.

List's Important Built-in Functions

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
```

List's Important Built-in Functions

```
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

Using Lists as Stacks

- The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved i.e., ("last-in, first-out").
- To add an item to the top of the stack, use append().
- To retrieve an item from the top of the stack, use pop() without an explicit index.
- For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
```

Using Lists as Stacks

```
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Using Lists as Queues

- We can also use a list conveniently as a queue, where the first element added is the first element retrieved i.e., (“first-in, first-out”).
- To add an item to the back of the queue, use append().
- To retrieve an item from the front of the queue, use pop() with 0 as the index.
- For example:

```
>>> queue = ["Eric", "John", "Michael"]
```

Using Lists as Queues

```
>>> queue.append("Terry") # Terry arrives
>>> queue.append("Graham") # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

The del Statement

- There is a way to remove an item from a list given its index instead of its value:
  - i.e., the del statement.
- This can also be used to remove slices from a list (which we did earlier by assignment of an empty list to the slice).
- For example:

```
>>> a = [-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
```

The del Statement

```
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

- del can also be used to delete entire variables:

```
>>> del a
```

- Referencing the name ***a*** hereafter is an error (at least until another value is assigned to it).

Comparing Data Structures

- Sequence objects may be compared to other objects with the same sequence type.
- The comparison uses *lexicographical* ordering:
  - first the first two items are compared, and
    - if they differ this determines the outcome of the comparison;
      - if they are equal, the next two items are compared, and so on, until either sequence is exhausted.
- If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively.
- If all items of two sequences compare equal, the sequences are considered equal.
- If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one.
- Lexicographical ordering for strings uses the ASCII ordering for individual characters.

Comparing Data Structures

- Some examples of comparisons between sequences with the same types:

  (1, 2, 3) < (1, 2, 4)
  [1, 2, 3] < [1, 2, 4]
  'ABC' < 'C' < 'Pascal' < 'Python'
  (1, 2, 3, 4) < (1, 2, 4)
  (1, 2) < (1, 2, -1)
  (1, 2, 3) == (1.0, 2.0, 3.0)
  (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
- Note that comparing objects of different types is legal.
- The outcome is deterministic but arbitrary: the types are ordered by their name.
  - Thus, a list is always smaller than a string
  - A string is always smaller than a tuple, etc.
- Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

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.

Packages in Multiple Directories

- Packages support one more special attribute, __path__.
- This is initialized to be a list containing the name of the directory holding the package's '__init__.py' before the code in that file is executed.
- This variable can be modified;
  - doing so affects future searches for modules and subpackages contained in the package.
- While this feature is not often needed, it can be used to extend the set of modules found in a package.

Formatting Input and Output

- There are several ways to present the output of a program;
  - data can be printed in a human-readable form
  - or written to a file for future use.
- So far we've encountered two ways of writing values: expression statements and the print statement.
- A third way is using the write() method of file objects;
  - the standard output file can be referenced as sys.stdout.

Formatting Input and Output

- Often we'll want more control over the formatting of our output than simply printing space-separated values.
- There are two ways to format our output;
  - the first way is to do all the string handling ourselves; using string slicing and concatenation operations we can create any lay-out we can imagine. The standard module string contains some useful operations for padding strings to a given column width.
  - The second way is to use the % operator with a string as the left argument. The % operator interprets the left argument much like a sprintf()-style format string to be applied to the right argument, and returns the string resulting from this formatting operation.

Formatting Input and Output

- Python has ways to convert any value to a string:
  - pass it to the repr() or str() functions.
  - Reverse quotes (‘) are equivalent to repr(), but their use is discouraged.
  - The str() function is meant to return representations of values which are fairly human-readable, while repr() is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is not equivalent syntax).
  - For objects which don’t have a particular representation for human consumption, str() will return the same value as repr().
  - Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function.
  - Strings and floating point numbers, in particular, have two distinct representations.

Formatting Input and Output

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(0.1)
```

Formatting Input and Output

```
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ',
        and y is ' + repr(y) + '...'
```

Formatting Input and Output

```
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string
        quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
```

Formatting Input and Output

```
'hello, world\n'
>>> # The argument to repr() may be any
        Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
>>> # reverse quotes are convenient in
   interactive sessions:
... `x, y, ('spam', 'eggs')`
```

Formatting Input and Output

"(32.5, 40000, (’spam’, ’eggs’))"

- Here are two ways to write a table of squares and cubes:

```
>>> import string
>>> for x in range(1, 11):
        print string.rjust(repr(x), 2),
        string.rjust(repr(x*x), 3),
        # Note trailing comma on previous line
```

Formatting Input and Output

```
print string.rjust(repr(x*x*x), 4)
1 1 1
2 4 8
3 9 27
4 16 64
```

Formatting Input and Output

5 25 125

6 36 216

7 49 343

8 64 512

9 81 729

10 100 1000

Formatting Input and Output

- >>> for x in range(1,11)
- print '%2d %3d %4d' % (x, x*x, x*x*x)

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
```

Formatting Input and Output

5 25 125

6 36 216

7 49 343

8 64 512

9 81 729

10 100 1000

Formatting Input and Output

- Note that one space between each column was added by the way print works:
  - it always adds spaces between its arguments.
- This example demonstrates the function:
  - string.rjust(), which right-justifies a string in a field of a given width by padding it with spaces on the left.
  - There are similar functions string.ljust() and string.center().
  - These functions do not write anything, they just return a new string.
  - If the input string is too long, they don't truncate it, but return it unchanged; this will mess up our column lay-out but that's usually better than the alternative, which would be lying about a value.
  - (If we really want truncation, we can always add a slice operation, as in 'string.ljust(x, n)[0:n]'.)

Formatting Input and Output

- There is another function, string.zfill(), which pads a numeric string on the left with zeros.
- It understands about plus and minus signs:

```
>>> import string
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
```

Formatting Input and Output

- Using the % operator looks like this:

```
>>> import math
>>> print 'The value of PI is approximately
        %5.3f.' % math.pi
```

- The value of PI is approximately 3.142.

Formatting Input and Output

- Most formats work exactly as in C and require that you pass the proper type; however, if you don't you get an exception, not a core dump.
- The %s format is more relaxed:
  - if the corresponding argument is not a string object, it is converted to string using the str() built-in function.

Formatting Input and Output

- If we have a really long format string that we don't want to split up,
  - it would be nice if we could reference the variables to be formatted by name instead of by position.

Formatting Input and Output

- This can be done by using form %(name) format, as shown here:
- >>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
- >>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
- Jack: 4098; Sjoerd: 4127; Dcab: 8637678

Reading and Writing Files

- We can open and use files for reading or writing by:
  - first creating an object of the file class
  - then we use the read, readline, or write methods of the file object to read from or write to the file depending on which mode you opened the file in
  - then finally, when we are finished the file, we call the close method of the file object.

Reading and Writing Files

- In the open function,
  - the first argument is a string containing the filename.
  - The second argument is another string containing a few characters describing the way in which the file will be used.
- *mode* can be 'r' when the file will only be read,
- 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending;
- any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing.
- The *mode* argument is optional; 'r' will be assumed if it's omitted.

Reading and Writing Files

```
poem = '''\
Programming is fun
When the work is done
if (you wanna make your work also fun):
use Python!
'''
f = file('poem.txt', 'w')
f.write(poem)
f.close()
f = file('poem.txt') # the file is opened in 'r'ead mode by default
while True:
    line = f.readline()
    if len(line) == 0: # Length 0 indicates EOF
            break
    print line, # So that extra newline is not added
f.close()
```

Reading and Writing Files

- f.tell() returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file.
- To change the file object's position,
  - use 'f.seek(*offset*, *from_what*)'.
- The position is computed from adding *offset* to a reference point;
  - the reference point is selected by the *from_what* argument.
- A *from_what* value of 0 measures from the beginning of the file,
  - 1 uses the current file position,
  - 2 uses the end of the file as the reference point.
  - *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

Reading and Writing Files

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5) # Go to the 6th byte in
      the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte
   before the end
```

Reading and Writing Files

- >>> f.read(1)
- 'd'
- When we're done with a file,
  - call f.close() to close it and free up any system resources taken up by the open file.
  - After calling f.close(), attempts to use the file object will automatically fail.
- >>> f.close()
- >>> f.read()

Reading and Writing Files

- Traceback (most recent call last):
- File "<stdin>", line 1, in ?
- ValueError: I/O operation on closed file

pickle Module

- Strings can easily be written to and read from a file. Numbers take a bit more effort, since the read() method only returns strings, which will have to be passed to a function like string.atoi(), which takes a string like '123' and returns its numeric value 123.
- However, when we want to save more complex data types like lists, dictionaries, or class instances, things get a lot more complicated.
- Rather than have users be constantly writing and debugging code to save complicated data types, Python provides a standard module called pickle.

pickle Module

- This is an amazing module that can take almost any Python object (even some forms of Python code!), and convert it to a string representation; this process is called *pickling*. Reconstructing the
- object from the string representation is called *unpickling*. Between pickling and unpickling, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.
- If you have an object 'x', and a file object 'f' that's been opened for writing, the simplest way to pickle the object takes only one line of code:

pickle Module

- pickle.dump(x, f)
- To unpickle the object again, if 'f' is a file object which has been opened for reading:
- x = pickle.load(f)
- (There are other variants of this, used when pickling many objects or when we don't want to write the pickled data to a file.

pickle Module

- pickle is the standard way to make Python objects which can be stored and reused by other programs or by a future invocation of the same program; the technical term for this is a persistent object.
- Because pickle is so widely used, many authors who write Python extensions take care to ensure that new data types such as matrices can be properly pickled and unpickled.

Using pickle module

- import cPickle
- shoplistfile = 'shoplist.data' # The name of the file we will use
- shoplist = ['apple', 'mango', 'carrot'] # Write to the storage
- f = file(shoplistfile, 'w')
- cPickle.dump(shoplist, f) # dump the data to the file f.close()
- del shoplist # Remove shoplist
- # Read back from storage
- f = file(shoplistfile)
- storedlist = cPickle.load(f) print storedlist