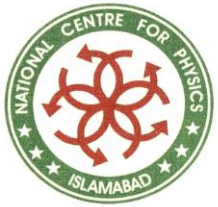


# Programming Python – Lecture#3

Mr. Adeel-ur-Rehman

ASC, National Centre for Physics

# Scheme of Lecture

- Object-Oriented Framework
- Python Scopes and Namespaces
- The self argument
- The __init__ method
- Classes
- The __getitem__ and __setitem__ methods
- Inheritance and Multiple Inheritance
- Iterators and Generators
- Exception Handling
- Gui Tkinter Programming Basics

# 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

- In a sense the set of attributes of an object also form a namespace.

- The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces;
  - for instance, two different modules may both define a function "maximize" without confusion — users of the modules must prefix it with the module name.

# Python Scopes and Namespaces

- In the expression modname.funcname, modname is a module object and funcname is an attribute of it.
- In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module:
  - they share the same namespace!




# Python Scopes and Namespaces

- Namespaces are created at different moments and have different lifetimes.
- The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted.
- The global namespace for a module is created when the module definition is read in;
  - normally, module namespaces also last until the interpreter quits.

# Python Scopes and Namespaces

- The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called __main__,
  - so they have their own global namespace.
- The built-in names actually also live in a module;
  - this is called __builtin__.

# Python Scopes and Namespaces

- The local namespace for a function is created
  - when the function is called
- And deleted
  - when the function returns or raises an exception that is not handled within the function.
  - Of course, recursive invocations each have their own local namespace.

# 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.

# Python Scopes and Namespaces

- If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names.
- Otherwise, all variables found outside of the innermost scope are read-only.

# Python Scopes and Namespaces

- Usually, the local scope references the local names of the current function.
- Outside of functions, the local scope references the same namespace as the global scope:
  - the module's namespace.
- Class definitions place yet another namespace in the local scope.

# Python Scopes and Namespaces

- A special quirk of Python is that assignments always go into the innermost scope.
- Assignments do not copy data—
  - they just bind names to objects.
- The same is true for deletions:
  - the statement 'del x' removes the binding of x from the namespace referenced by the local scope.




# Python Scopes and Namespaces

- In fact, all operations that introduce new names use the local scope:
  - in particular, import statements and function definitions bind the module or function name in the local scope. (The global statement can be used to indicate that particular variables live in the global scope.)

# 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
swaroop = Person('Swaroop')
swaroop.sayHi()
swaroop.howMany()
kalam = Person('Abdul Kalam')
kalam.sayHi()
kalam.howMany()
swaroop.sayHi()
swaroop.howMany()
```

# Special Class Methods

- In addition to normal class methods, there are a number of special methods which Python classes can define.
- Instead of being called directly by our code (like normal methods), special methods are called for you by Python in particular circumstances or when specific syntax is used.
- We can get and set items with a syntax that doesn't include explicitly invoking methods.

# The __getitem__ Special Method

```
def __getitem__(self, key): return self.data[key]
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__getitem__("name")
'/music/_singles/kairo.mp3'
>>> f["name"] (2)
'/music/_singles/kairo.mp3'
```

- The __getitem__ special method looks simple enough. Like the normal methods clear, keys, and values, it just redirects to the dictionary to return its value. But how does it get called?

# The __getitem__ Special Method

- Well, we can call __getitem__ directly, but in practice we wouldn't actually do that;
- The right way to use __getitem__ is to get Python to call it for us.
- This looks just like the syntax we would use to get a dictionary value, and in fact it returns the value we would expect.
- But here's the missing link: under the covers, Python has converted this syntax to the method call:
  - f.__getitem__("name").
- That's why __getitem__ is a special class method; not only can we call it ourself, we can get Python to call it for us by using the right syntax.

# The __setitem__ Special Method

```
def __setitem__(self, key, item):self.data[key] = item
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__setitem__("genre", 31)
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':31}
>>> f["genre"] = 32
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':32}
```

# The __setitem__ Special Method

- Like the __getitem__ method, __setitem__ simply redirects to the real dictionary self.data to do its work.
- And like __getitem__, we wouldn't ordinarily call it directly like this.
- Python calls __setitem__ for us when we use the right syntax.
- This looks like regular dictionary syntax, except of course that **f** is really a class that's trying very hard to masquerade as a dictionary, and __setitem__ is an essential part of that masquerade.
- This second last line of code actually calls f.__setitem__("genre", 32) under the covers.

# Inheritance

- One of the major benefits of object oriented programming is **reuse** of code
- One of the ways this is achieved is through the inheritance mechanism.
- Inheritance can be best imagined as implementing a *type and subtype* relationship between classes.
- Consider this example:

# Using Inheritance

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print '(Initialized SchoolMember: %s)' %
          self.name
    def tell(self):
        print 'Name:"%s" Age:"%s" ' % (self.name,
          self.age),
```

# Using Inheritance

```
class Teacher(SchoolMember):
'''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print '(Initialized Teacher: %s)' % self.name
    def tell(self):
        SchoolMember.tell(self)
        print 'Salary:"%d"' % self.salary
```

# Using Inheritance

```
class Student(SchoolMember):
'''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print '(Initialized Student: %s)' % self.name
    def tell(self):
        SchoolMember.tell(self)
        print 'Marks:"%d"' % self.marks
```

# Using Inheritance

```
t = Teacher('Mrs. Abraham', 40, 30000)
s = Student('Swaroop', 21, 75)
print # prints a blank line
members = [t, s]
for member in members:
    member.tell() # Works for instances of
      Student as well as Teacher
```

# Multiple Inheritance

- Python supports a limited form of multiple inheritance as well.
- A class definition with multiple base classes looks as follows:

```
class DerivedClassName(Base1, Base2, Base3):
<statement-1>
.
<statement-N>
```

- The only rule necessary to explain the semantics is the resolution rule used for class attribute references.

# Multiple Inheritance

- This is depth-first, left-to-right. Thus, if an attribute is not found in DerivedClassName, it is searched in Base1, then (recursively) in the base classes of Base1, and only if it is not found there, it is searched in Base2, and so on.
- A well-known problem with multiple inheritance is a class derived from two classes that happen to have a common base class. While it is easy enough to figure out what happens in this case (the instance will have a single copy of “instance variables” or data attributes used by the common base class).

# 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
```

# Iterators

```
for char in "123":
        print char
for line in open("myfile.txt"):
        print line
```

- 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
```

# Iterators

- Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to our classes.
- Define a __iter__() method which returns an object with a next() method.
- If the class defines next(), then __iter__() can just return self:

# Iterators

```
>>> class Reverse:
"Iterator for looping over a sequence
   backwards"
def __init__(self, data):
   self.data = data
   self.index = len(data)
def __iter__(self):
   return self
```

# Iterators

```
def next(self):
    if self.index == 0:
        raise StopIteration
    self.index = self.index - 1
    return self.data[self.index]
```

# Iterators

```
>>> for char in Reverse('spam'):
print char
m
a
p
s
```

# Generators

- Generators are a simple and powerful tool for creating iterators.
- They are written like regular functions but use the yield statement whenever they want to return data.
- Each time the next() is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed).
- An example shows that generators can be trivially easy to create:

# Generators

```
>>> def reverse(data):
for index in range(len(data)-1, -1, -1):
yield data[index]
>>> for char in reverse('golf'):
print char
f
l
o
g
```

# Generators

- Anything that can be done with generators can also be done with class based iterators as described in the previous section.
- What makes generators so compact is that the __iter__() and next() methods are created automatically.
- Another key feature is that the local variables and execution state are automatically saved between calls.
- This made the function easier to write and much more clear than an approach using class variables like **self.index** and **self.data**.

# Generators

- In addition to automatic method creation and saving program state, when generators terminate, they automatically raise StopIteration.
- In combination, these features make it easy to create iterators with no more effort than writing a regular function.

# Exception Handling

- Exceptions occur when certain *exceptional* situations occur in our program.
- For example, what if we are reading a file and we accidentally deleted it in another window or some other error occurred? Such situations are handled using exceptions.
- What if our program had some invalid statements?
- This is handled by Python which **raises** its hands and tells you there is an **error**.

# Exception Handling

- Consider a simple print statement.
- What if we misspelt print as Print?
- Note the capitalization.
- 
- In this case, Python *raises* a syntax error.

```
>>> Print 'Hello, World' File "<stdin>", line 1 Print 'Hello, World' ^ SyntaxError: invalid syntax
>>> print 'Hello, World'
Hello, World
>>>
```

- Observe that a SyntaxError is raised and also the location where the error was detected, is printed. This is what a *handler* for the error does.

# Exception Handling

- To show the usage of exceptions, we will **try** to read input from the user and see what happens.
- >>> s = raw_input('Enter something --> ')
- Enter something --> Traceback (most recent call last): File "<stdin>", line 1, in ? EOFError
- >>>
- Here, we ask the user for input and if he/she presses **Ctrl-d** i.e. the EOF (end of file) character, then Python raises an error called EOFError.
- Next, we will see how to handle such errors.

# Exception Handling

- We can handle exceptions using the try..except statement.
- We basically put our usual statements within the **try-block**.
- And we put all the error handlers in the **except-block**.

# Exception Handling

```
import sys
try:
    s = raw_input('Enter something --> ')
except EOFError:
    print '\nWhy did you do an EOF on me?' sys.exit() #
       Exit the program
except:
    print '\nSome error/exception occurred.'
    # Here, we are not exiting the program
print 'Done'
```

# Exception Handling

- We put all the statements that might raise an error in the try block
- And then handle all errors and exceptions in the except clause/block.
- The except clause can handle a single specified error or exception or a parenthesized list of errors/exceptions.
- If no names of errors or exceptions are supplied, it will handle *all* errors and exceptions. There has to be at least one except clause associated with every try clause.

# Exception Handling

- If any error or exception is not handled, then the default Python handler is called which stops the execution of the program and prints a message.
- We can also have an else clause with the try..catch block.
- The else clause is executed if no exception occurs.

# Exception Handling

- We can also get the **exception object** so that we can retrieve additional information about the exception which has occurred.
- This is demonstrated in the next example.

# Exception Handling

- We can raise exceptions using the raise statement
  - - we specify the name of the error/exception and the exception object.
  - The error or exception that we can raise should be a class which directly or indirectly is a derived class of the Error or Exception class respectively.

# Exception Handling

- class ShortInputException(Exception):
  - '''A user-defined exception class.'''
  - def __init__(self, length, atleast):
    - self.length = length
    - self.atleast = atleast
- try:
  - s = raw_input('Enter something --> ')
  - if len(s) < 3:
    - raise ShortInputException(len(s), 3)

# Exception Handling

- Other work can go as usual here.
  except EOFError:
  - print '\nWhy did you do an EOF on me?‘
- except ShortInputException, x:
  - print ‘ \nThe input was of length %d, it should be at least %d'\ % (x.length, x.atleast)
- else:
  - print 'No exception was raised.'

# Exception Handling

- Other work can go as usual here.
  except EOFError:
  - print '\nWhy did you do an EOF on me?‘
- except ShortInputException, x:
  - print ‘ \nThe input was of length %d, it should be at least %d'\ % (x.length, x.atleast)
- else:
  - print 'No exception was raised.'

# Exception Handling

- What if we wanted some statements to execute after the **try block** whether or not an exception was raised?
- This is done using the **finally block**.
- Note that if we are using a **finally block**, we cannot have any except clauses for the same try block.

# Exception Handling

```
try:
    f = file('poem.txt')
    while True: # Our usual file-reading block
        l = f.readline()
        if len(l) == 0:
            break
        print l,
finally:
    print 'Cleaning up...'
    f.close()
```

# GUI – Tkinter Overview

- Of various GUI options, Tkinter is the de facto standard way to implement portable user interfaces in Python today.
- Tkinter's availability, accessibility, documentation and extensions have made it the most widely used Python GUI solution for many years running.

# Tkinter Structure

- Tkinter is the simply the name of Python's interface to Tk
  - -- a GUI library originally written for use with the Tcl programming language.
- Python's Tkinter module talks to Tk, and the Tk API in turn interfaces with the underlying window system:
  - Microsoft Windows
  - X Windows on Unix
  - or Macintosh

# Tkinter Structure

- Python's Tkinter adds a software layer on top of Tk that allows Python scripts to call out to Tk to build and configure interfaces, and routes control back to Python scripts that handle user-generated events (e.g., mouse-clicks).
- i.e., GUI calls are internally routed from Python script, to Tkinter, to Tk; GUI events are routed from Tk, to Tkinter, and back to a Python script.

# Tkinter Structure

- Luckily, Python programmers don't normally need to care about all this call routing going on internally;
  - They simply make widgets and register Python functions to handle widget events.
- Because of the overall structure, event handlers are usually known as **callback handlers** as the GUI library "calls back" to Python code when events occur.

# Tkinter Structure

- Python/Tkinter programs are entirely event-driven:
  - They build displays and register handlers for events, and then do nothing but wait for events to occur.
  - During the wait, the Tk GUI library runs an event loop that watches for mouseclicks, keyboard presses, and so on.
  - All application program processing happens in the registered callback handlers in response to events.

# A Tiny GUI example

```
 # Get a widget object
from Tkinter import Label
 # Make one
widget = Label(None, text='Hello GUI World!')
 # Arrange it
widget.pack()
 # Start event loop
widget.mainloop()
```

# A Tiny GUI Example

- The above written code is a complete Python Tkinter GUI program.
- When this script is run, we get a simple window with a label in the middle.

# Tkinter Coding Basics

- Although the last example was a trivial one but it illustrates steps common to most Tkinter programs:
  - Loads a widget class from the Tkinter module
  - Makes an instance of the imported Label class
  - Packs(arrange) the new Label in its parent widget
  - Calls mainloop to bring up the window and start the Tkinter event loop

# Tkinter Coding Basics

- The mainloop method called last puts the label on the screen and enters a Tkinter wait state, which watches for user-generated GUI events.
- Within the mainloop function, Tkinter internally monitors things like the keyboard and mouse, to detect user-generated events.
- Because of this model, the mainloop call here never returns to our script while the GUI is displayed on screen.

# Tkinter Coding Basics

- To display a GUI's window, we need to call mainloop.
- To display widgets within the window, they must be packed so that the Tkinter geometry manager knows about them.
- A mainloop without a pack shows an empty window.
- And a pack without a mainloop in a script shows nothing since the script never enters an event wait-state.