Advanced Programming with Python

25/04/2016

Module 1. Introduction to Object Oriented Concepts

A history of object oriented programming.

The first object oriented language developed in the 1960's was SIMULA. This language was developed at the Norweigian Computing center at Oslo, Norway. Many of the features that are common to OO languages, such as classes, inheritance, encapsulation, and garbage collection were first introduced in SIMULA,

SIMULA inspired Bjarne Stroustrup at AT&T Bell Labs to invent an OO language, C++ that was built on top of the already popular C Programming Language implementation, also invented there. C++ became the first widely used OO language in production.

In the early 1990's, James Gosling and others at Sun Microsystems developed *Java*. Java was an OO language inspired by and took many features from, C++.

Around the same time, Guido Van Rossum developed another OO Language, *Python*. Python is a dynamic, or scripting language that is developed from the ground up as an OO language. Python has gown in popularity and is now one of the most popular languages in use around the world.

Why use Object Oriented Programming?

1. OOP provides a clear modular structure for programs which makes it good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.
2. OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
3. OOP provides a good framework for code libraries where supplied software

components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.

What does this mean in practice? Let's consider the following example:

You work as a software engineer for a large bank. You are tasked to add functionality to an existing application written in a procedural language such as C or COBOL. In order to complete this task, not only must you write the software for the new functionality, you must integrated it into the existing code base, perform regression testing to make sure that you haven't broken any existing code and you must also test that you haven't introduced new bugs in to the code base.

Using procedural type programming, this cycle can quickly devolve into a nighmarish scenario where more and more programmers spend more of their time trying to fix existing errors rather than creating new functionality and features. Additionally, without strong process controls (which most companies do not have) existing software can suffer *bit rot* which is a term used to describe increasingly unwieldy and buggy software applications.

As we'll see later, OOP provides ways to solve these existing problems in a clean, well designed fashion,.

Object Oriented Concepts

Before we dive into OOP programming, it is useful to understand some of the concepts behind OO as well as discuss how to design an OO application.

Some of the basic concepts behind Object Oriented Programming include:

1. Objects and Classes. These are related artifacts but not necessarily the same thing. A Class is a design or representation of an object, much like the blueprint for a house or other structure. Classes show how data and functions (called *methods* in OO) are coupled together. Each object, if properly designed, serves one function. For example, a Person object may handle all relevant details of a Person, i.e. setting such data attributes as the person's name, date of birth, address and other relevant details. Objects are *instantiations* of a Class, much like a house is an instantiation of the blueprints. The class shows how an object is constructed. An object is memory is the result of being built according to the class specifications,.
2. Information hiding. More commonly known as *encapsulation*. One of the main principles of OO is that a programmer should not need to know how an object performs its tasks, moreover the programmer using the object should not have access to the data or internal functions of the object. The object should provide a set of methods that allow the programmer to perform operations on the object's data.
3. Inheritance. The programmer should be able to create new objects that inherit properties of another object. For instance, we discussed a Person class earlier. We should be able to create a new class, *Employee,* which inherits attributes from the Person class, such as the name, and date of birth, but can also add new attributes, such as an employee ID. Inheritance is one of the key properties of OOP.,
4. Interfaces. The ability to define methods and data that can be implemented by the programmer without doing so beforehand. Not all OO languages have or need this concept. Java uses interfaces extensively, however this is implemented as abstract virtual classes in C++ and Python. We'll discuss this more later on.
5. Polymorphism. This is another key concept in OOP. This allows the programmer to treat multiple subclasses as if they are the base class. For example, The programmer may have a base class, called *Animal*, and various inherited classes, called *Cat, Dog, and Horse*. Using polymorphism it is possible to treat instantiated objects of those classes as *Animals.*

Object Oriented Design. The S.O.L.I.D. principles

In the early years of this century, Robert C, Martin, known as “Uncle Bob”, formulated the common principles of Object Oriented design into a mnemonic termed SOLID. These five principles are as follows:

1. S. - Single Responsibility principle. An class design should only have one responsibility. Or as Martin puts it: “A class should only have one reason to change”. For example, a Person object should only focus on managing the attributes of a Person, and not, for example, worry about printing the data in a formatted way. This responsibility should lie in another class, for example, a FormattedPrint class.
2. O. - Open/Closed principle. A class should be open to extension, but never to modification, What this means in principle is that once a class is created, assuming that it is designed correctly, future software updates should never be made to the original class. Instead using inheritance a new class should be derived from the parent with the new functionality encapsulated.

3. L. - Liskov's Substitution principle. This principle says that any derived class can substitute for its base class without changes. Often this runs counterintuitive to what would seem like common sense. For example, say you had a class Square() that inherits from class Rectangle(). In the real world, a square is a type of rectangle, so this seems to make sense, however, because a square has both the height and the width as identical, setting the height in the square also should set the width as well. This is not true for rectangles. Thus, this design fails the Liskov substitution principle.

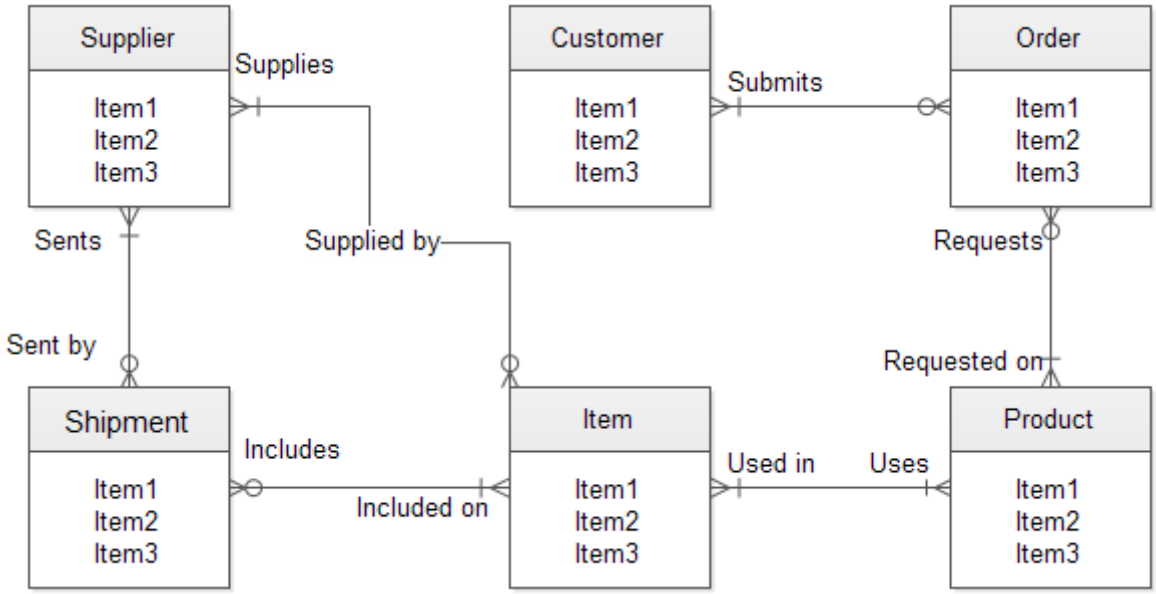
Figure 1.

4. I. - Interface/Segregation Principle. This principle dictates that no client should need to know about methods that it won't use. A well known example of this is the experience of Robert Martin at Xerox. Xerox had designed new software to control both printing and stapling. There was a single Job class that had methods for both of these operations. A client that implemented a stapling job had to know about the print methods as well, even though it didn't use them. This made the Job class code grow large and complex, which made it very difficult to maintain. The proper way to implement this with this principle was to have a Print class and a Staple class with methods only relevant for their function.
5. D. - Dependency/Inversion principle. This principle says that class that high level classes, i.e. classes that do high level logic shouldn't depend on low level classes directly, instead, an abstraction should be created between the. This is also called *decoupling*. For example, if a class is created that requires a connection to a database, it shouldn't call the database connection class directly. Why? Well, for example, what if the company is using MySQL for its database. Then the high level class would call the low level class that connects to the MySQL database. Why is this a problem? What if the company decides instead to replace MySQL with PostGresSQL? Then the high level class has to be re-written to accomodate the new database. Rather the right approach is to create a new class, DatabaseConnection, which abstracts out the connection. Then, it is a much simpler matter to add a new interface for a PostGresSQL connection than it is to re-write the high level class to handle the new database.

Object Oriented Design. Data Modeling with Entity-Relationship diagrams

An entity/relationship diagram is a graphical representation that shows the relationships between people, places, objects, concepts or events in a system. Here is an example of an ERD.

Figure 2.

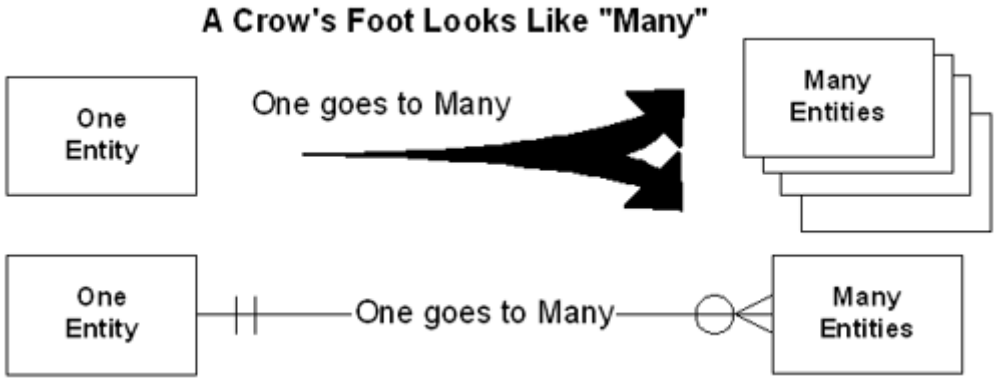


This diagram uses the “Crow's foot” notation that we will be discussing in this class.

The boxes represent the classes, with the blue shaded rectangle title representing the class name and the other parts being the class attributes (data and methods).

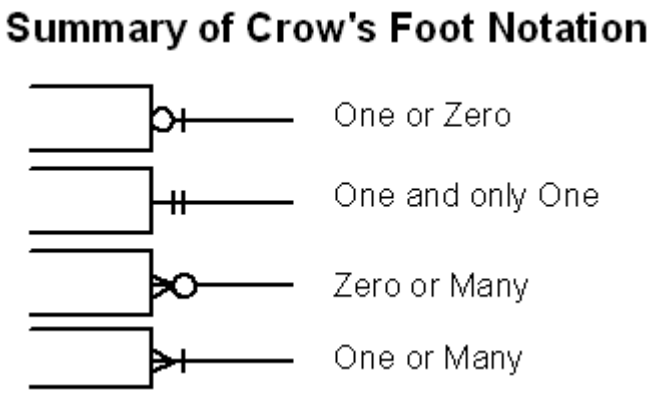
Crow's foot notation looks like this:

Figure 3.



Here is a diagram showing the Crow's foot notation scheme.

Figure 4.



Object Oriented Design. Inheritance, Aggregation and Composition

When designing an OO application, there are three criteria that should be applied when creating classes. “Is-a”, “Has-a” and “Has-an”.

For example,

“A house is a building” (is-a). This is an example of an implementation of inheritance.

“A house has a room” (has-a). This is an example of an implementation of composition.

“A house has-an occupant” (has-an). This is an example of an implementation of aggregation.

Note that the differences between composition and aggregation are subtle, yet important. Another example of the difference between the two is the example of a text editor with a buffer (object) and a file (object). When the text editor is shut down, the buffer object is destroyed, but the file object (hopefully) is not.

These three criteria are commonly used to determine the relationship between classes. For example, an Employee “is-a” Person, and an Employee “has-a” Name. A Corporation “has-an” employee. Note that in the third example, the employee may leave, but the corporation remains.

Lab 1. Given the diagram in figure 2, Decipher the crow's foot notation to understand the relationship cardinality between the different entities.

Lab 2. You are designing software for a bookstore. Design an ERD that clearly shows the relationships between the following entities,

1. Author
2. Title
3. Supplier
4. Subject

Use the concepts of inheritance, composition and aggregation when designing the ERD for the bookstore application.

Module 2. Writing Object Oriented Programs in Python

Terminology

Before we start on OOP, let's define some terminology.

1. **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
2. **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
3. **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
4. **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
5. **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
6. **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
7. **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
8. **Instantiation:** The creation of an instance of a class.
9. **Method :** A special kind of function that is defined in a class definition.
10. **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
11. **Operator overloading:** The assignment of more than one function to a particular operator.

Writing classes

It is important to understand that Python is a completely object oriented language. All intrinsic data types of Python are objects. This is different from other languages like Java

or C++. So, the following:

mynum = 123

creates an object called 'mynum' which is a number object and assigns the value of 123 to a data attribute within that object.

When writing a Python program you are using objects constantly, whether you realize it or not. Python uses the '.' syntax to access data and methods associated with the object. So, for example:

```
#!/usr/bin/python3

mystr = “foobar”
print(mystr.upper())

Output:
FOOBAR
```

So, mystr is a python string object with its own methods, including the one shown in the example, 'upper()'. Python has many such classes available as part of its distribution, including strings, lists, dictionaries and others.

However, at some point, you will want to create your own classes. The way to do this is to create a class like so:

```
class myclass(object):
    def __init__(self):
        # some code here
    def method1(self):
      # some code here
    def method2(self):
     # some code here
```

Let's analyze the code fragment above.

The line class myclass(object) tells Python that will be defining a new class called 'myclass'. Myclass inherits from the default base class 'object'.

The def __init__(self): function is the initializer method. Note that this isn't quite identical to the concept of a constructor in languages like C++ or Java. Python has a __new__()

method that is a proper constructor, however it isn't used for other than specialized requirements. Generally, when you create a new class, use the __init__() method to initialize any instance variables with data. Note that you define all methods with a required first parameter, 'self'. The self variable contains the specific instance of the class that has been created. All object methods must have this explicitly declared parameter in the parameter list.

Note that 'self' is **not** a keyword in Python, but a convention. Any non-reserved word will work for the variable name.

Here's an example of a more detailed class definition

```
class Employee(object):
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)
```

One new statement here is 'Employee.empCount +=1' empCount is a *class* variable. That is, it is a variable that is the same between all instances of the class, while instance variables are different for each instantiation of the object.

Accessing class attributes

The preferred method for accessing class attributes in Python is to use the '.' operator.

For example:

```
class foo(object):
    def __init__(self):
        self.var1 = “foo”
        self.var2 = “bar”

f = foo()
print(f.var1)
print (f.var2)
```

Note that we first *instantiate* the foo object with the line f=foo(). This tells Python to create an object using the foo class blueprint in memory and return a reference to this object into the variable called f. Now that we have this object reference, we can access elements of this object with the '.' operator, namely f.var1 and f.var2.

Having said this, can anyone spot what the problem with this code example is?

The above code example breaks a fundamental concept of OOP, namely encapsulation. You shouldn't be able to access individual variable elements of a class directly. Rather, this is a better form.

```
Class foo(object):
    def __init__(self):
        self.var1 = “foo”
        self.var2 = “bar”

    def getvar1(self):
        return(self.var1)
    def getvar2(self):
        return(self.var2)

f = foo()
print(f.getvar1())
print (f.getvar2())
```

There is another way of accessing class attributes, using the following functions:

Function	Description
hasattr(object,name)	Returns True if the object has the attribute in question
getattr(object,name)	Returns the value of the named attribute
setattr(object,name,value):	Sets the value of the named attribute
delattr(object,name)	Deletes the named attribute from the object.

All Python classes have some built-in attributes. Some of these attributes are as follows:

Attribute	Description
__dict__	A dictionary that contains all of the objects attributes.
__doc__	Contains the document string of the class, if any.
__name__	Contains the class name
__module__	The module name in which the class is defined.
__bases__	A tuple containing the base classes from which this class is derived (if any).

Object creation and garbage collection in Python

The way Python handles object creation is somewhat different than it is in C++ or Java.

Writing this code:

f=foo()

binds the instantiation of the foo() object to name *f.* A name is just the pythonic definition of a variable.

What does this mean in practice?

Let's look at an example in C++ vs. Python.

C++

std::string foo = “bar”;

foo = “baz”

This code instantiates a string object. Foo contains a pointer to an area of memory which now contains the text 'bar'. The next line overwrites that area of memory with the text 'baz'.

Python

foo = “bar”

foo = “baz”

This binds the foo name to the string object containing 'bar'. It then binds foo to a string object containing “baz”. Note that the “bar” object may in fact still exist until the Python garbage collection routine destroys the object.

Python handles garbage collection internally. An object will be deleted when all references to it are gone. Note that this isn't necessarily deterministic. The object is only deleted when the garbage collection routine runs.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope.

Since garbage collection is done automatically, the need for a *destructor* is substantially lessened. However, you can create a destructor by overriding the __del__() in your class.

Inheritance

One of the main features of object oriented programming is the ability for a class to *inherit* properties from other classes. As we saw earlier, creating a class *Employee* that inherits properties from a class *Person* is an example of inheritance.

Additionally the derived or child class can also override the data and methods of the parent class. Why is this interesting? Let's consider the following:

You are responsible for maintaining a banking application. This banking application has a class *BankAcct* that has all the properties needed for a real bank account, such as a person's name, contact details, etc. Additionally, it contains methods relating to bank accounts, such as interest calculation. However, we have a problem. This structure works for a single type of bank account, but of course, a bank has many types of accounts, checking, current, savings, pension, credit card, etc.

If we had only a single bank account class, then we would have to write code like so:

```
if accttype == 'Savings':
    do_some_savings_acct_stuff()
elif accttype == 'Current'
    do_some_current_acct_stuff()
```

and so on. With this approach, we quickly lose any advantage of OOP as the class would become so large and unwieldly that effectively, we're going back to the paradigm of procedural programming.

How do we fix this problem? We can change our architecture so that we have the following:

base class *BankAcct.* This class has all of the properties common to all bank accounts, such as a person's name and contact details.

Derived classes such as *SavingsAcct*. This class inherits all of the BankAcct base class attributes, but may extend the class to add new attributes, or override existing attributes of the base class.

Now, we can create new classes, such as *CurrentAcct, CreditCardAcct,* etc. that extend the base BankAcct class.

This is an example of the Open/Closed principle in SOLID. You don't modify the base class, you extend it by creating new children classes.

In Python, we can override base class methods in the derived classes. Here is a simple example.

```
class Parent(object):
    def __init__(self):
        pass
    def Method1(self):
        print(“In parent\n”)

class Child(Parent):
    def __init__(self):
         pass
    def Method1(self):
       print (“In child\n”)

foo = Child()
foo.Method1()
```

Calling foo.Method1() should result in the output string: In child

There are also instances when you may want to call the parent's method either before the child's method runs or in place of the child method. To do this we use the 'super' key word.

Consider the following code:

```
class Parent(object):
    def __init__(self):
        print (“In parent\n”)

class Child(Parent):
    def__init__(self):
        super.__init__()
        print (“In child\n”)

c = Child()
```

When this code instantiates a child object, the first thing that the __init__() function does is call the Parent's __init__() function using the super() function, it then continues on with it's own code.

Composition

While inheritance is a powerful feature of OOP, sometimes it isn't the appropriate way to design a system. Inheritance can run the risk of being overly complex or not quite fitting the reality of the system that you are implementing. Let's look at an example:

```
class Car(object):
    def__init__(self):
        some_attributes here
    def carmethod1(self):
        pass
    def carmethod2(self):

class Toyota(car):
    def __init__(self):
        some_Toyota_specific attributes here
    def carmethod1(self):
        override some car method here
```

The above code is a classic example of inheritance. This example works well when you don't have a very large collection of classes, however, once you start adding classes, for example a new car type 'Nissan', or a new mark 'Toyota Land Cruiser', then you can start getting a very long and complex chain of inheritance which can make your code large, inefficient and difficult to maintain. Here's another way of doing it.

```
from abc import ABCMeta,abstractmethodf
class Transmission(metaclass = ABCMeta):
    def __init__(self):
        pass
    @abstractmethod
    def transmissionMethod1(self):
        return

<other interrfaces here>

class Car(object):
   def __init__(self):
       self.Transmission = Transmission()
       self.Engine = Engine()
       self.Door = Door()

  def base_car_method(self):
       pass

class Toyota(Car):
   def __init__(self):
       pass
   def transmission1(self):
       return("Transmission1")

Transmission.register(Toyota)
```

With composition, we can *flatten* the class hiearchy. We no longer have to have multiple classes inheriting from other classes. Instead, each sub-class will implement an *Abstract Base Class*. An abstract base class is simply a class that declares empty methods. It is then up to the class using it to actually define the methods. In Java, this would be called an *interface*.

When should you use composition vs. Inheritance?

Avoid multiple inheritance at all costs, as it's too complex to be reliable. If you're stuck with it, then be prepared to know the class hierarchy and spend time finding where everything is coming from.
Use composition to package code into modules that are used in many different unrelated places and situations.
Use inheritance only when are are clearly relatled reusable pieces of code that fit under a single common concept, or if you have to because of some other constraint.

Mixins

Another feature supported by Python is the *mixin*. Mixin is a limited type of multiple inheritance where your class can inherit *Mixin* classes. Mixin classes are support classes that are theoretically not stand-alone objects. For example

```
Import Book
from Index import IndexMixin
TextBook(IndexMixin,Book):
    <some stuff here>
```

Mixins should be used in a “has-a” format. That is, a Book “has-a” index. Also note that Python reads class inheritance heirarchy from right to left. So, if you have a method with the same name in both Book and IndexMixin, make sure that you have the proper order of the classes, otherwise you will get unexpected results.

Mixins are yet another way to do composition with objects.

Polymorphism

Polymorphism is a key concept of OOP. Polymorphism allows a programmer to treat all derived types of a class as a base class. What do we mean by this?

Say we have a class called Gem(). We can also have derived classes from Gem() such as Ruby(), Emerald(), Diamond(), Opal() and so forth...

We can then treat each member of the derived classes as a Gem() object. Why is this interesting?

Well, say we want to calculate the value of the gem given things like number of carets, degree of flawlessness, etc. We can override each method in the given derived class. I.e. Ruby.calculateValue(), Diamond.calculateValue(), etc. Each derived class inherits the calculateValue() method from the base Gem() class. Additionally, we can collect all of the different gem types in a single collection and treat them the same.

Operator Overloading

Python allows the programmer to *overload* or assign a different meaning to many operators, such as the '+', and '-' operators. This can be done by overriding some of the default methods available to all classes.

To override the '+' operator, override the __add__() method that comes with the class by default.

When should we do this? Let's consider a class to model imginary numbers. We know that imginary numbers have the format *a + bi* where a and b are real numbers and i is the square root of negative 1. If we declare a class Complex() and instantiates some objects like so:

```
Class Complex(object):
    def __init(self,a,b):
        self.a = a
        self.b = b
    def __add__(self,other):
        return(Complex(a+other.a,b+other.b)

class C1 = complex(1,2)
class C2=complex(3,4)
class C3 = C1 +_C2
```

Note that since we've overiden the __add__ method, when calling C3 + C1 + C2. C3 will be the returned object that contains the sum of both *a* ane *b.*

This overriding is how Python is able to concatenate two strings together with the '+' operator.

Other popular methods to override are:

1. *Str().* This method tells Python how to print out the object.

2. *Cmp()*. This method tells Python how to compare two objects.

Module 3. Advanced Topics in Python

List comprehensions

List comprehensions are a very fast way to transform the contents of one list into another.

Let's consider the following problem:

We have a list consisting of the first ten even numbers like so:

[2,4,6,8,10,12,14,16,18,20]

We'd like to construct a new list that square the values in the original list. We could do the following:

```
originalList=[2,4,,8,10,12,14,16,18,20]
newList = []
for number in originalList:
    newlist.append(number * number(
```

This would give us a new list where each number is a square of the original. However, Python gives us a much easier way to write this:

```
originalList = [2,4,6,8,10,12,14,16,18,20]
newList = [number * number  for number in OriginalList]
```

This new construct is a *List Comprehension*. This is a fast, and powerful way to create new lists. Additionally, it produces faster python code than the original code above.

Here's a list comprehension that takes a list of temperatures in Celsius and converts them to Fahrenheit.

```
Celsius = (37.4,23.2,11.5,2.7)
Fahrenheit= [ ((float(9) / 5) * temp + 32) for temp in Celsius]
```

In list comprehensions you can use the *for* and *if* keywords to help you construct lists. For example:

```
noprimes = [j for i in range(2,8) for j in range(i:*2,100,i)]
primes = [x for x in range(2,100) if x not in noprimes]
```

The above code implements the *Sieve of Erastosthenes* to calculate prime numbers.

Functional programming in Python using *lambda's, map(), filter() and reduce()*

Lambda's allow programmers the ability to create anonymous functions in code. This is a very powerful tool, especially when combined with other builtin functions such as map() and filter().

Let's look at an example.

```
def f(x,n):
    return(x+n)

g = lambda x,n: x+n

print (f(1,2))
print (g(1,2))
```

In the above code example, both print functions return the same value. f and g both do the same thing. However g is defined as an anonymous function using a lambda expression.

Lambda's can be embedded in other function calls. For example:

As we have seen, list comprehensions are a powerful way to create new lists by transforming old lists, however, list comprehensions have some limitations. They only allow the use of *if* and *for* keywords. This may not be enough for your purposes. Also, very complex list comprehensions quickly become unreadable and cumbersome. Because of this, Python offers another built-in function called *map().*

The map function looks like this:

```
NewList = list(map(func,oldlist))
```

Where func is a user supplied function and oldlist is the list passed to the function, one element at a time. Note that map returns an iterator. If we want to get a list, we need to convert it to a list using the list typecast.

Let's re-write our Celsius to Fahrenheit conversion to use the map() function like so:

```
def convert(x):
    return (float(9) * 5) /(x + 32)

CelsiusList = [32.3,27.5,2.3,11.1]
FahrenheitList = list(map(convert,CelsiusList))
```

We can also use map with multiple lists.

Let's consider the following problem.

We have two lists

a = [1,5,11,14,19]

b = [2.4,9,15,35]

We'd like a new list that compare the values of each index of the two original list and returns the maximum of the two compared values. We can use a combination of a lambda, the zip() and map() functions to do this as follows:

```
a= [1,5,11,14,19]
b = [2,4,9,15,35]
print(list(map(lambda pair:max(pair),zip(a,b))))
```

The filter function allows the programmer to *filter out* elements of a list that are unwanted for any reason. For example, let's say that in a list of ten elements from 1 to 10, I want a new list that only contains odd elements. I can do the following:

```
Mylist = range(1,11)
myfilteredlist = filter(lambda x: x%2 != 0, mylist)
print (list(myfilteredlist))
```

Note that filtering can also be done using a list comprehension like so:

```
Myfilteredlist = [x for x in mylist if x%2 != 0]
```

The third function we will look at is the *reduce()* function. The reduce function is a bit more complex and will require some explanation.

The reduce function takes the parameters (func and seq) similar to map and filter. However, what reduce does is a bit more complex. Here are the steps.

Given a sequence (s_1, s_2, s_3, s_4,s_5...s_n) reduce will send the first two elements in the sequence to the func. The function will return the result like so:

(func(S_1,S_2),S ,S_4,S_5...S_n). Next reduce will take the next element in the sequence and apply the function to the results of the first function and the element like so:

(func(func(S_1,S_2),S $_3$),S_4 ,S_5..S_n) and so on until there is only one element left in the list.

Iterators

As we may have noted from experience, it is possible to *iterate* over a number of different data types in Python, including lists, dictionaries, strings, tuples and other objects. For example:

```
elements = [1,2,3,4,5]
for element in elements:
    print element
```

But, how is this implemented? How does the for loop know to go from the first to the last element of the list? In this case, the for statement calls the *iter()* function. This function returns a special object called an *iterator*. The iterator object defines a function called *__next__()* (Note that in Python 2 this function is just called next().) Using the built-in function *next()* and passing in the iterator object will invoke the __next__ function to get the next element of the list (or whatever iterable object you pass in). For example, we can now re-write the above code as follows:

```
elements = [1,2,3,4,5]
it = iter(elements)
while (True):
   try:
      print (next(it))
   except StopIteration
      break
```

Creating your own iterators is relatively straight forward. When creating an iterable object, override the __iter__ method and supply your own.

So, what is an iterable object? An iterable object is anything that can be defined as follows:

1. Anything that can be looped over. For example a list or a string.
2. Anything that can appear on the right of a for loop. For example: for x in iterable_object:

3. Anything that you can call with the iter() function that returns an iterator
4. Any object that defines the __iter__ or __getitems__ methods.

An iterable object is not quite the same as an *iterator*. Which is defined as follows:

1. Any object with a state that remembers where it is during iteration.
2. Any object with a __next__ method defined that:

- returns the next value in the collection.
- Updates the state to point to the next value.
- Signals when it is finished iteration by raising the StopIteration exception.

```
elements = [1,2,3,4,5] # This is an iterable object
it = iter(elements) # it is the iterator object.
```

While the iterator and iterable object can be defined as two separate entities, in practice most programmers combine then like so:

```
Class IterableExample(object):
    def __iter__(self)L#
        return self
    define next (self):
        <some code here>
```

Generators

Generators are a special type of iterator. You can think of a generator as an iterable function. For example:

```
def my_generator():
    l = [1,2,3,4,5]
    for e in l:
        yield e
x = my_generator()
try:
    next(x)
except StopIteration:
    print (“Finished”)
```

Defining a name such as x = my_generator(), we can now call next(x) on the generator to give us the next element in the defined list *l*. Note the main difference between a generator and a normal function. Using the keyword *yield* automatically makes the function a generator. Unlike functions, generators maintain state between calls. If my_generator had return e rather than yield e, the only value that it would ever return is '1'.

However, because we use the yield keyword, every call to the generator using it as an argument to the bultin next() function give us the next element of the list, so the output would be 1 2 3 4 5 rather than just 1 if we had a normal function.

Additionally, since x is now iterable, we could also re-write the above code like this:

```
def my_generator():
    l = [1,2,3,4,5]
    for e in l:
        yield e
x = my_generator()
for i in x:
    print (i)
```

We can do even more with generators. Recall the concept of a list comprehension. Python also supports generator comprehensions.

For example we can now re-write the above code even more simply like so:

```
my_generator = (n for n in range(1,6))

for i in my_generator:

    print (i)
```

Note. In Python 3, the range function returns a generator rather than a list in Python 2. If you want a generator object in Python 2, use the built-in *xrange()* function.

We can also use the send method in generators to be able to create coroutines. Coroutines allow us to have functions that can collaboratively call co-routines, pass execution to them, and then pass execution back to the calling function without using a 'return'. The real key here is that the 'control' state is saved between calls. For example:

```
def my_coroutine(s):
    while True:
        p = yield
        print (pow(s,p))

x = my_coroutine(2)
x.send(None)
x.send(2)
x.send(5)
```

The s parameter is the base number. The p received by the yield is the power.

We initialize the co-routine by sending x.send(None). Alternatively, next(x) would also work. Now we can send values to the co-routine using the send method to the generator.

Therefore, x.send(2) returns 2 to the 2th power, i.e. 4

x.send(5) returns 2 to the 5^{th} power, i.e. 32

Co-routines are primarily consumers of data. Generators are producers of data.

Using these tools make it easy to create producer/consumer patterns, where the generator produces data for the consumer co-routine to process.

Remember, all coroutines must be “primed” by calling either the next function or the send method with the None parameter.

Closures and Decorators

A closure is a function that is defined inside another function. Like so:

```
def f(a):
    def g(b,c):
        return a * (b+c)
    return g
x = f (1)
print (x(2,3))
```

In this case, we have created a function f and defined another function g inside of f. Note that even though the a name is not defined in g, it is still usable as the scope of f is readable from g.

We then create a function instance x and pass it the a parameter value of 1.

We then call that instance and then pass the b and c parameter values of 2 and 3.

This is useful when we have a situation where we may have a function that takes many parameters, only some of which change on a regular basis, i.e. in the above case we assume that the a parameter changes rarely, whereas the b and c parameters change on each subsequent call to the function.

Decorators are a “syntactical sugar” for closures. Here is an example:

We note that when using coroutines, we must always initialize it with a call to __next__() or next() in Python2, or use the send method with None passed. If I'm calling a number of coroutines in code, this extra lines of code become tiresome and redundant. I.e. we don't want to constantly have to call the next/send methods every time we want to initialize the coroutine. Better to declare a function we'll call coroutine and use that as a “decorator” to our coroutine. Like so:

```
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        cr.send(None)
        return cr
    return start

@coroutine
def some_coroutine(myarg):
    some_code_here

f = some_coroutine(“foo”)
f.send(“bar”_
```

Note that the '@' symbol designates the decorator in Python. Therefore, when calling the some_coroutine coroutine, first Python will call the coroutine function, which will then call the coroutine itself, the decorator calls the send method so you don't have to do it manually.

Module 4. Design Patterns

What are design patterns? Design Patterns are a suite of “best practices” that can be applied to common problems when writing software. Most of these patterns were developed by trial and error over decades of software design and engineering.

In 1994, the seminal book on design patterns *Design Patterns – Elements of Reusable Object Oriented Software”* written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, commonly known as the “gang of four”, was released. They advocated the following:

- Program to an interface, not an implementation.
- Favor composition over inheritance.

They defined 23 different patterns comprised of three groupings, Creational, Structural and Behavorial patterns. We'll look at a small subset of these in this course.

Creational Patterns.

These patterns allow programmers to create objects whlie hiding the logic of how the object is created. This type of pattern gives programmers flexibility in deciding how an object will be created. A *factory* pattern is a type of a creational pattern.

Structural Patterns

These patterns concern class and object composition. Inheritance is used to create interfaces and define ways to compose objects in order to create new functionality. A *strategy* pattern is a type of a structural pattern.

Behavorial Patterns

These patterns are primarily concerned with communication between objects. *Model-View-Controller* is a type of behavorial pattern.

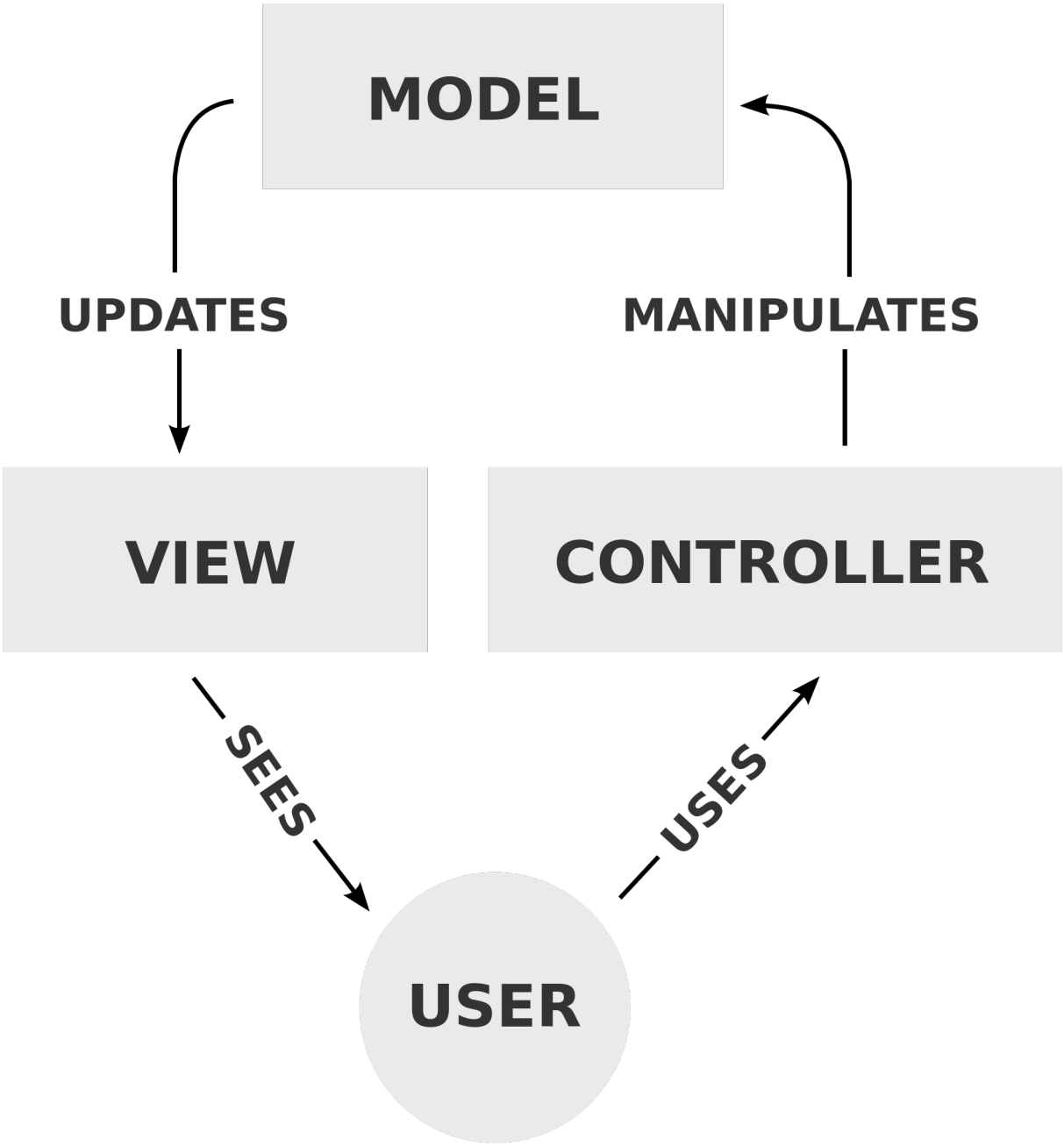
Let's now look at some specific pattern types.

Model-View-Controller Pattern.

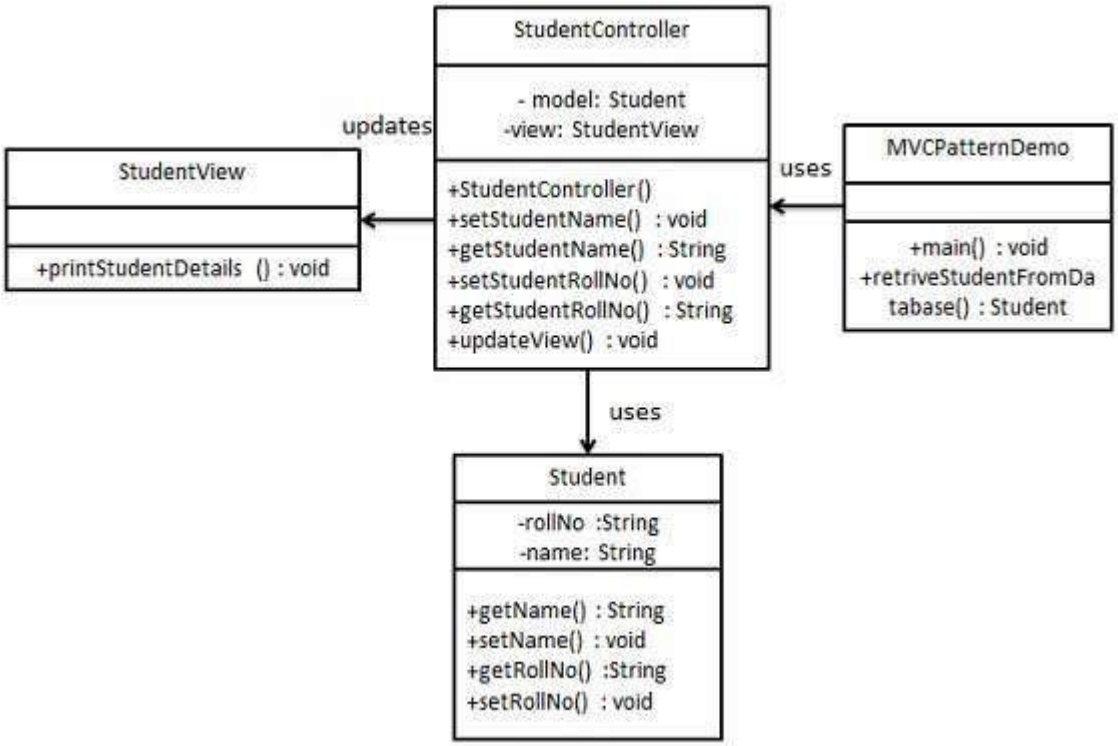
The MVC pattern is used for implementing user interfaces. For example, if a web site wants to send data to a user, instead of handling all of the aspects of this in one class, we can split it out into three distinct parts.

The Model	Concerned with access to the data that the client is requesting.
The Controller	The software logic that retrieves the data and presents it. In other words, the logic that connects the model and the view together.
The View	Displays the model's data.

Graphically, it can be represented as follows:



Here is an example of an ERD that implements an MVC pattern allowing people to reference data from a *Student*.

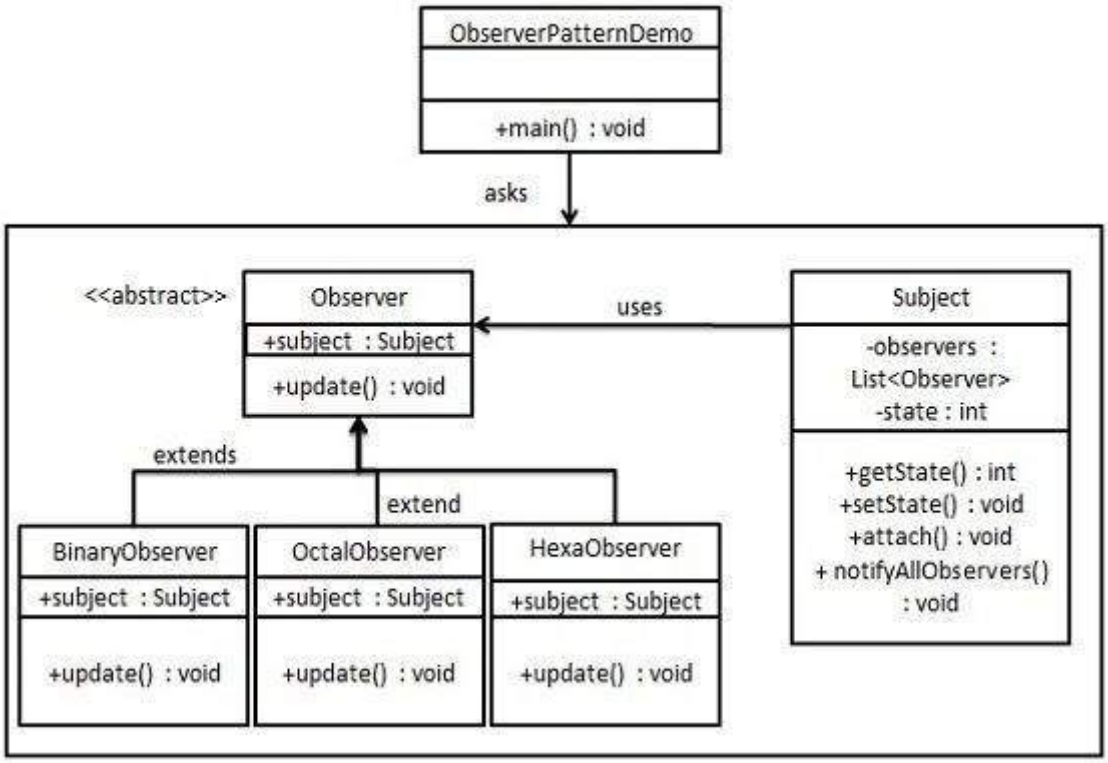


There are many applications that work on the concept of a subscriber model. For example, let's take an application that monitors weather. This application takes data from a number of sensors. The application observes each sensor and is notified whenever the sensor sends new data. We can implement this type of application using an Observer pattern.

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We ccreate an abstract class *Observer* and a concrete class *Subject* that extends class *Observer*.

ObserverPatternDemo, our demo class, will use *Subject* and concrete class object to show observer pattern in action.

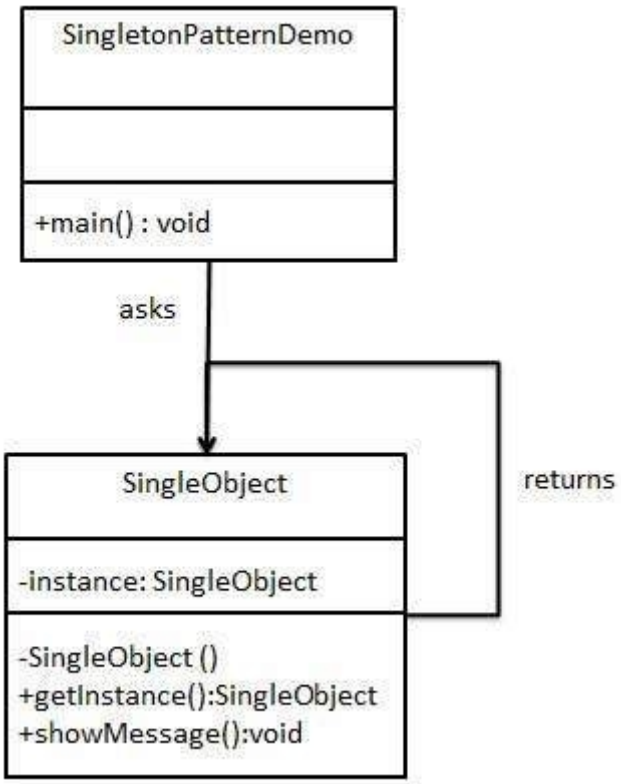
Here is an ERD of an implementation of the observer pattern.



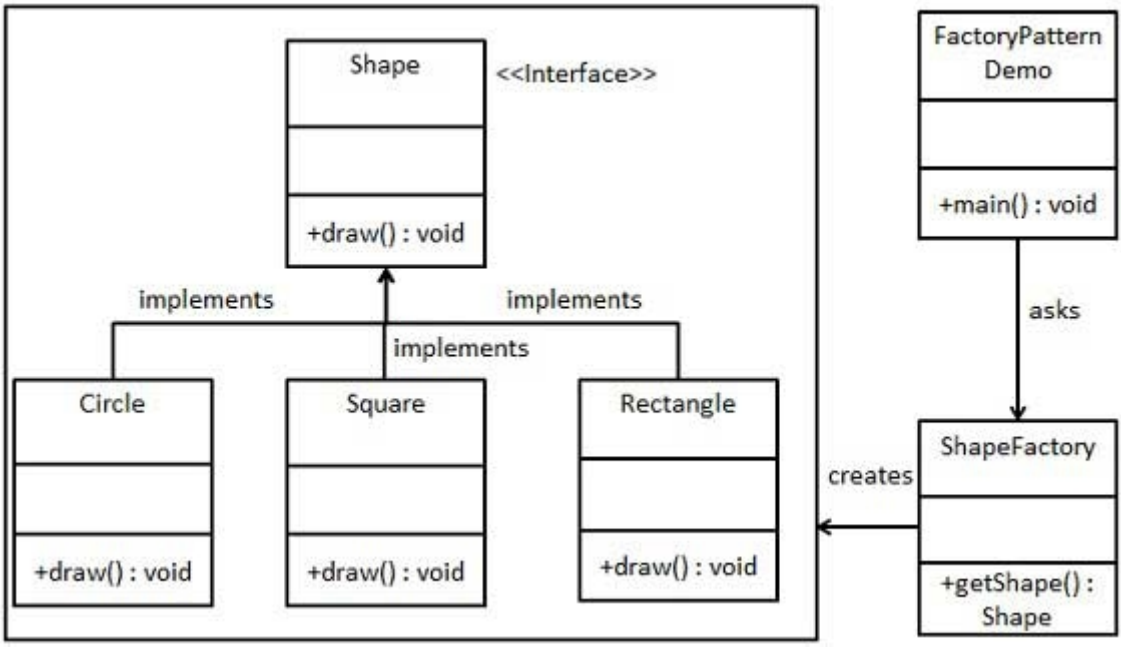
The singleton pattern is one of the simplest design patterns to create. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Here is an ERD of a singleton pattern.



The factory pattern is one of the most commonly used patterns in software design. This type of pattern is a creational pattern that allows the designer flexibility in what sort of objects to create. For example, say we have a *Shape* class with three derived classes, *Square, Rectangle, and Circle*. Instead of having the programmer specifically instantiate circles, rectangles and squares, we can now create a new factory class, *ShapeFactory* and tell it what type of shape to create. It then returns the object type requested. Here is an ERD of this type of pattern:



Module 5. Parsing XML and JSON

XML or eXtensible Markup Language, is an open source language that is generally used to represent data in a single, machine and user readable format that can be parsed by different systems. XML it self does not do anything, it simply wraps data in markup tags that can be read by other software. Primarily, we use XML to send and receive data from different systems that might maintain their data using completely different types of software without having to know the internal workings of the data formats of each different system.

There are two main types of parsers for XML

1. SAX Parsers. SAX stands for Simple Api for Xml. With SAX, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk and the entire file is never stored in memory.

2. DOM Parsers. DOM stands for Document Object Model. With this model, the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document. The parser can then go through the tree to extract data elements of interest to the programmer.

Because DOM stores its data in memory, it is always going to be faster to lookup and retrieve data than SAX. SAX, however, is preferred if you are reading many small files or you have a very large element tree that can exceed memory resources.

Additionally, SAX is read-only, while DOM allows changes to the XML file. Since these two different APIs literally complement each other, there is no reason why you cannot use them both for large projects.

Parsing with SAX.

SAX is a standard interface for event-driven XML parsing. Parsing XML with SAX generally requires you to create your own ContentHandler by subclassing xml.sax.ContentHandler.

Your *ContentHandler* handles the particular tags and attributes of your flavor(s) of XML. A ContentHandler object provides methods to handle various parsing events. Its owning parser calls ContentHandler methods as it parses the XML file.

The methods *startDocument* and *endDocument* are called at the start and the end of the XML file. The method *characters(text)* is passed character data of the XML file via the parameter text.

The ContentHandler is called at the start and end of each element. If the parser is not in namespace mode, the methods *startElement(tag, attributes)* and *endElement(tag)* are

called; otherwise, the corresponding methods *startElementNS* and *endElementNS* are called. Here, tag is the element tag, and attributes is an Attributes object.

The following table describes some of the SAX parser methods.

Method	Description
make_parser	Following method creates a new parser object and returns it. The parser object created will be of the first parser type the system finds. xml.sax.make_parser([parser_list]) **parser_list:** The optional argument consisting of a list of parsers to use which must all implement the make_parser method.
parse	Following method creates a SAX parser and uses it to parse a document. xml.sax.parse(xmlfile, contenthandler[, errorhandler]) Here is the detail of the parameters − •**xmlfile:** This is the name of the XML file to read from. •**contenthandler:** This must be a ContentHandler object. •**errorhandler:** If specified, errorhandler must be a SAX ErrorHandler object.
parse_string	There is one more method to create a SAX parser and to parse the specified **XML string**. xml.sax.parseString(xmlstring, contenthandler[, errorhandler]) Here is the detail of the parameters − •**xmlstring:** This is the name of the XML string to read from.

	•**contenthandler:** This must be a ContentHandler object. •**errorhandler:** If specified, errorhandler must be a SAX ErrorHandler object.

Parsing with DOM

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another.

Here is the easiest way to quickly load an XML document and to create a minidom object using the xml.dom module. The minidom object provides a simple parser method that quickly creates a DOM tree from the XML file.

Parsing JSON

Parsing JSON is even easier than parsing XML with Python. This is because JSON files are already in pythonic format. You can use the JSON library to simply read and write JSON files directly via the library methods.

For example, if we have some JSON data that looks like this:

```
json_string = '{"first_name": "Guido", "last_name":"Rossum"}'
```

We can use the json library to decode this like so:

```
import json

parsed_json = json.loads(json_string)
```

Note that the JSON libraries take strings and code or decode them into Python data structures.

As well, we can load JSON from data files using the json.load() method and write to files using json.dump(). Do not confuse json.load and json.loads. The first method loads from a file, the second method loads from a string.

Module 6. Database programming with Python

In this module, we will explore the ability of Python to connect to a SQL database, execute SQL statements and receive data back from SQL SELECT statements.

For this module, we will use the Postgres SQL database as well as the psycopg database API.

Here is an example of connecting to a postgressql database in Python.

```
#!/usr/bin/env python
import psycopg2
try:
    conn = psycopg2.connect("dbname='template1' user='dbuser' host='localhost'
password='dbpass'")
except:
    print “Unable to connect to the database”
```

Here we attempt to a database called 'template1' as user 'dbuser' to host 'localhost' with a password 'dbpass'.

Once this is done, we can now create a cursor variable. It is this variable that will store the results from SQL operations.

```
cur = conn.cursor()
```

Note that this will return results into memory. This may not be suitable, especially if you are returning results with hundreds or thousands of rows. In this case, add a parameter such as 'mycursor' to the conn.cursor() method and this will allow a “streaming” cursor which does not return all of the data at once.

Once we have the cursor variable defined, we can now execute SQL statements like so:

```
cur.execute(“SELECT datname from template1”)
rows = cur.fetchall()
for row in rows:
    print “    “,row
```

You must be extremely careful when passing parameters to an SQL string. This is one of the main avenues for SQL injection attacks. For example, never do this:

```
SQL = “INSERT INTO authors (name) VALUES ('%s');” #Never do this
data = (“O'Reilly”,)
cur.execute(SQL % data) # This will fail miserably
```

The correct way to pass variables in an SQL command is like this:

```
SQL = “INSERT INTO authors (name) VALUES (%s);” # Note, no quotes
data = (“O'Reilly”,)
cur.execute(SQL,data) # Note, no % operator
```

Never use the Python string concatenation (+) or string parameter interpolation to pass variables to an SQL query string.

Some database transactions, such as INSERT's, UPDATE's and DELETE's require the ability for the database to either commit or do a rollback on the transaction, the latter in case of an error where the transaction cannot be completed.

When finished with the database connection, issue the following:

```
cur.close()
conn.close()
```

With Python (from version 2.5), you can use the *with* statement to do commits and rollbacks automatically. You can use the form:

```
with psycopg2.connect (DSN) as conn:
    with conn.cursor() as curs:
        curs.execute(SQL)
```

When the connection exits the *with* block, if no exceptions have been raised, then the transaction is automatically commited, otherwise, if an exception has been raised, the transaction is automatically rolled back.

Module 7. Multithreading in Python

What is threading? Threading is similar to concurrent execution of programs. In concurrent execution, a process can create one or more *children* processes by using the *fork()* and *exec()* system calls to tell the operating system to do so. However, this tends to be very resource-intensive and can affect performance. Another way to do this is to create *threads*. Threads are also called light weight processes. They run in the same context as process creating it, i.e. they share the same resources, although they can have their own local variables. A multithreaded application, if designed properly can be substantially faster than one that is concurrent or runs as a single process.

Unfortunately for us, the reference implementation of Python, *cpython*, does not allow CPU based multithreading. I.e. the multicores available on standard modern Intel/AMD processors cannot be accessed from Python due to the *Global Interpreter Lock* (GIL) in all current versions of cpython. Note that other implementations, such as jython and IronPython, do not suffer from this problem.

Even though we can't use multithreading for CPU based computation, we can still use it for I/O processing.

```
#!/usr/bin/python

import thread
import time

# Define a function for the thread
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Create two threads as follows
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Error: unable to start thread"

while 1:
   pass
```

Let's analyze the above code. We note that we have to import the thread library first. We then create a function that will run inside the thread, *print_time().* Each thread will run an instance of this function separately.

We then create the thread with the start_new_thread() method, passing the actual function to run and any arguments to that function.

This is an example of the low-level *thread* module. However as of Python 2.4, a new library *threading* has been introduced and is now more commonly used as it abstracts much of the threading functionality away.

Let's re-write the first threading example using the newer threading library.

```
#!/usr/bin/python

import threading
import time

exitFlag =0
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            time.sleep(delay)
            print "%s: %s" % (threadName, time.ctime(time.time()))
         counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

print "Exiting Main Thread"
```

Notice that the code is a bit more complex. Here we create a new class *myThread* which a derived class of the Thread class defined in the threading module. The first thing that the __init__() method does is call the superclasses init module. It also defines a *run()* method that will call the actual function that does the work.

Calling the start method from the thread object will automatically invoke the run method inside the thread object.

Note that in the above coding examples, none of the threads actually change any variables. When a thread changes a variable, it is possible to hit a *race condition*. This is where multiple threads depend on the value of that variable. It is possible that one thread changes the variable value that will affect the behavior of other thread while it is executing.

In order to prevent this, we use the concept of a *lock*. A lock is also called a *mutex.* In order to use a lock, we need to do the following:

create the lock object like so:

myLock = threading.Lock()

Then, when changing a variable, we do the following:

myLock.acquire()

critical_variable = 5

myLock.release()

Note that inbetween the acquire and the release, only the thread that holds the lock can change the value of critical_variable. Any other thread attempting to get the lock will block until the lock is released by the first thread. Note, that if the first thread doesn't release the lock due to a programmer error, all of the threads will eventually block and the program will probably hang.