

# Python Design Patterns - Quick Guide



## Beyond Basic Programming - Intermediate Python

36 Lectures   3 hours

 Mohammad Nauman

[More Detail](#)

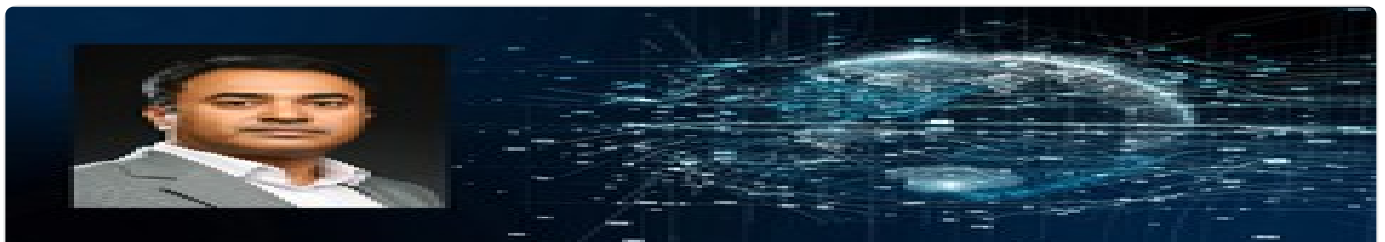


## Practical Machine Learning Using Python

91 Lectures   23.5 hours

 MANAS DASGUPTA

[More Detail](#)



## Practical Data Science Using Python

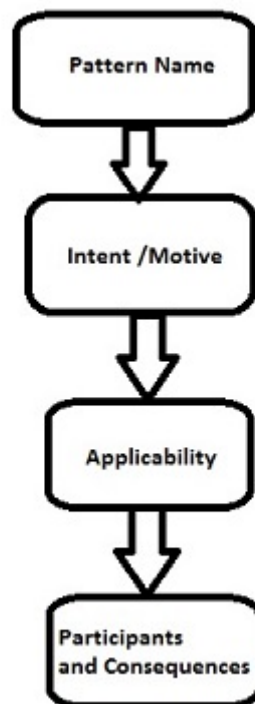
22 Lectures   6 hours

# Python Design Patterns - Introduction

Design patterns are used to represent the pattern used by developers to create software or web application. These patterns are selected based on the requirement analysis. The patterns describe the solution to the problem, when and where to apply the solution and the consequences of the implementation.

## Structure of a design pattern

The documentation of design pattern is maintained in a way that focuses more on the technology that is used and in what ways. The following diagram explains the basic structure of design pattern documentation.



### Pattern Name

It describes the pattern in short and effective manner.

### Intent/Motive

It describes what the pattern does.

### Applicability

It describes the list of situations where pattern is applicable.

## Participants and consequences

Participants include classes and objects that participate in the design pattern with a list of consequences that exist with the pattern.

## Why Python?

Python is an open source scripting language. It has libraries that support a variety of design patterns. The syntax of python is easy to understand and uses English keywords.

Python provides support for the list of design patterns that are mentioned below. These design patterns will be used throughout this tutorial –

- Model View Controller Pattern
- Singleton pattern
- Factory pattern
- Builder Pattern
- Prototype Pattern
- Facade Pattern
- Command Pattern
- Adapter Pattern
- Prototype Pattern
- Decorator Pattern
- Proxy Pattern
- Chain of Responsibility Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern
- Flyweight Pattern
- Abstract Factory Pattern
- Object Oriented Pattern

## Benefits of using design pattern

Following are the different benefits of design pattern –

- Patterns provide developer a selection of tried and tested solutions for the specified problems.
- All design patterns are language neutral.
- Patterns help to achieve communication and maintain well documentation.
- It includes a record of accomplishment to reduce any technical risk to the project.
- Design patterns are highly flexible to use and easy to understand.

# Python Design Patterns - Gist

Python is an open source scripting language, which is high-level, interpreted, interactive and object-oriented. It is designed to be highly readable. The syntax of Python language is easy to understand and uses English keywords frequently.

## Features of Python Language

In this section, we will learn about the different features of Python language.

### Interpreted

Python is processed at runtime using the interpreter. There is no need to compile program before execution. It is similar to PERL and PHP.

### Object-Oriented

Python follows object-oriented style and design patterns. It includes class definition with various features like encapsulation, polymorphism and many more.

### Portable

Python code written in Windows operating system and can be used in Mac operating system. The code can be reused and portable as per the requirements.

### Easy to code

Python syntax is easy to understand and code. Any developer can understand the syntax of Python within few hours. Python can be described as “programmer-friendly”

### Extensible

If needed, a user can write some of Python code in C language as well. It is also possible to put python code in source code in different languages like C++. This makes Python an extensible language.

## Important Points

Consider the following important points related to Python programming language –

- It includes functional and structured programming methods as well as object-oriented programming methods.
- It can be used as scripting language or as a programming language.
- It includes automatic garbage collection.
- It includes high-level dynamic data types and supports various dynamic type checking.
- Python includes a feature of integration with C, C++ and languages like Java.

## How to download python language in your system?

To download Python language in your system, follow this link –

<https://www.python.org/downloads/>



It includes packages for various operating systems like Windows, MacOS and Linux distributions.

## The Important Tools in Python

In this section, we will learn in brief about a few important tools in Python.

### Python Strings

The basic declaration of strings is as follows –

```
str = 'Hello World!'
```

### Python Lists

The lists of python can be declared as compound data types separated by commas and enclosed within square brackets ([]).

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  
tinylist = [123, 'john']
```

### Python Tuples

A tuple is dynamic data type of Python, which consists of number of values separated by commas. Tuples are enclosed with parentheses.

```
tinytuple = (123, 'john')
```

### Python Dictionary

Python dictionary is a type of hash table. A dictionary key can be almost any data type of Python. The data types are usually numbers or strings.

```
tinydict = {'name': 'omkar', 'code': 6734, 'dept': 'sales'}
```

## What constitutes a design pattern in Python?

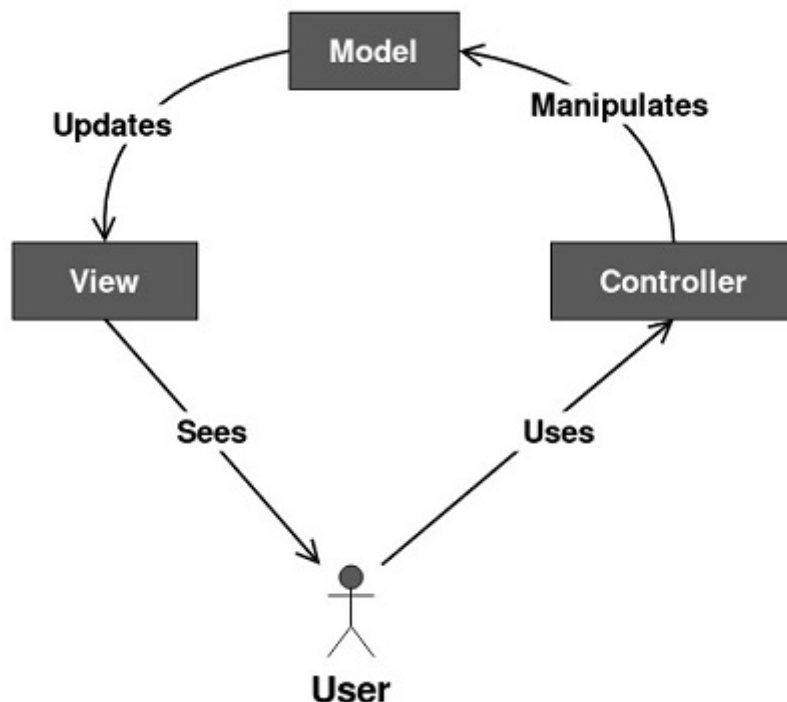
Python helps in constituting a design pattern using the following parameters –

- Pattern Name
- Intent
- Aliases
- Motivation
- Problem
- Solution
- Structure
- Participants
- Constraints
- Sample Code

## Model View Controller Pattern

Model View Controller is the most commonly used design pattern. Developers find it easy to implement this design pattern.

Following is a basic architecture of the Model View Controller –



Let us now see how the structure works.

## Model

It consists of pure application logic, which interacts with the database. It includes all the information to represent data to the end user.

## View

View represents the HTML files, which interact with the end user. It represents the model's data to user.

## Controller

It acts as an intermediary between view and model. It listens to the events triggered by view and queries model for the same.

## Python code

Let us consider a basic object called "Person" and create an MVC design pattern.

### Model.py

```
import json

class Person(object):
    def __init__(self, first_name = None, last_name = None):
        self.first_name = first_name
        self.last_name = last_name
        #returns Person name, ex: John Doe
    def name(self):
        return ("%s %s" % (self.first_name, self.last_name))

    @classmethod
    #returns all people inside db.txt as list of Person objects
    def getAll(self):
        database = open('db.txt', 'r')
        result = []
        json_list = json.loads(database.read())
        for item in json_list:
            item = json.loads(item)
            person = Person(item['first_name'], item['last_name'])
            result.append(person)
        return result
```

It calls for a method, which fetches all the records of the Person table in database. The records are presented in JSON format.

## View

It displays all the records fetched within the model. View never interacts with model; controller does this work (communicating with model and view).

```
from model import Person
def showAllView(list):
    print 'In our db we have %i users. Here they are:' % len(list)
    for item in list:
        print item.name()
def startView():
    print 'MVC - the simplest example'
    print 'Do you want to see everyone in my db?[y/n]'
def endView():
    print 'Goodbye!'
```

## Controller

Controller interacts with model through the **getAll()** method which fetches all the records displayed to the end user.

```
from model import Person
import view

def showAll():
    #gets list of all Person objects
    people_in_db = Person.getAll()
    #calls view
    return view.showAllView(people_in_db)

def start():
    view.startView()
    input = raw_input()
    if input == 'y':
        return showAll()
    else:
        return view.endView()

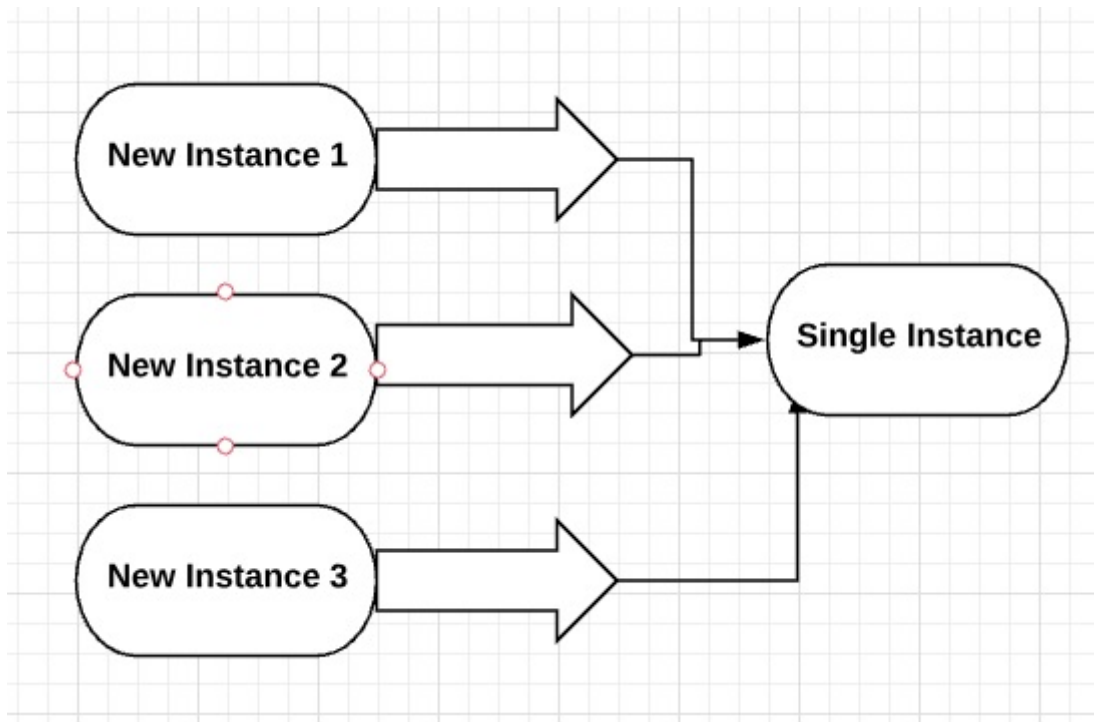
if __name__ == "__main__":
    #running controller function
    start()
```

# Python Design Patterns - Singleton



This pattern restricts the instantiation of a class to one object. It is a type of creational pattern and involves only one class to create methods and specified objects.

It provides a global point of access to the instance created.



## How to implement a singleton class?

The following program demonstrates the implementation of singleton class where it prints the instances created multiple times.

```

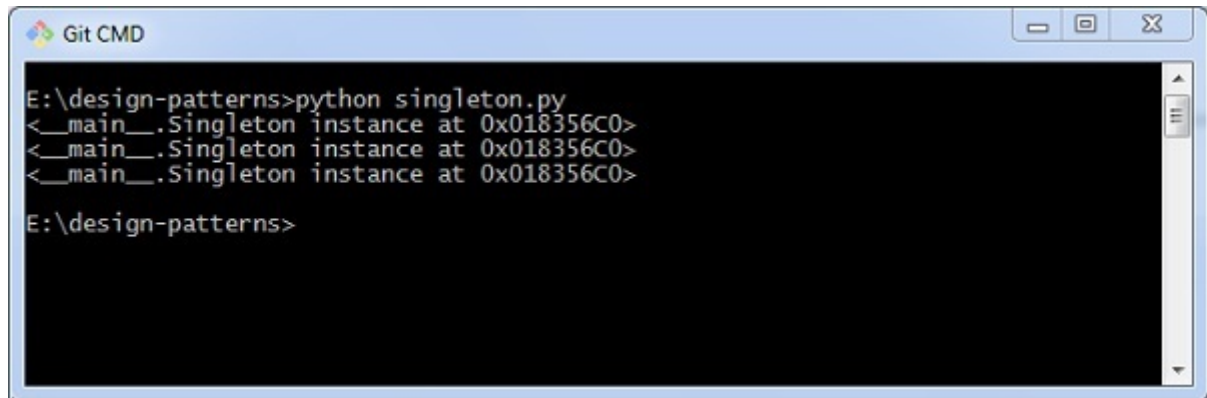
class Singleton:
    __instance = None
    @staticmethod
    def getInstance():
        """ Static access method. """
        if Singleton.__instance == None:
            Singleton()
        return Singleton.__instance
    def __init__(self):
        """ Virtually private constructor. """
        if Singleton.__instance != None:
            raise Exception("This class is a singleton!")
        else:
            Singleton.__instance = self
s = Singleton()
print s

s = Singleton.getInstance()
print s
  
```

```
s = Singleton.getInstance()  
print s
```

## Output

The above program generates the following output –



```
Git CMD  
E:\design-patterns>python singleton.py  
<__main__.Singleton instance at 0x018356C0>  
<__main__.Singleton instance at 0x018356C0>  
<__main__.Singleton instance at 0x018356C0>  
E:\design-patterns>
```

The number of instances created are same and there is no difference in the objects listed in output.

## Python Design Patterns - Factory

The factory pattern comes under the creational patterns list category. It provides one of the best ways to create an object. In factory pattern, objects are created without exposing the logic to client and referring to the newly created object using a common interface.

Factory patterns are implemented in Python using factory method. When a user calls a method such that we pass in a string and the return value as a new object is implemented through factory method. The type of object used in factory method is determined by string which is passed through method.

In the example below, every method includes object as a parameter, which is implemented through factory method.

### How to implement a factory pattern?

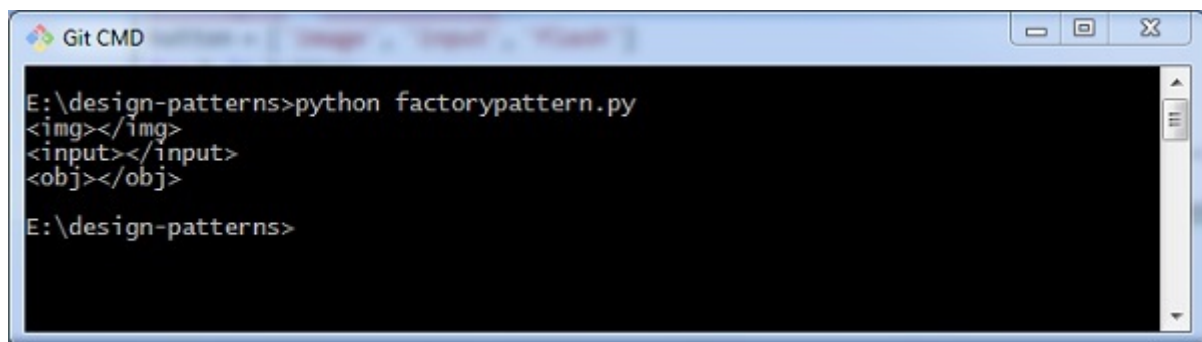
Let us now see how to implement a factory pattern.

```
class Button(object):  
    html = ""  
    def get_html(self):  
        return self.html  
  
class Image(Button):  
    html = "<img></img>"
```

```
class Input(Button):  
    html = "<input></input>"  
  
class Flash(Button):  
    html = "<obj></obj>"  
  
class ButtonFactory():  
    def create_button(self, typ):  
        targetclass = typ.capitalize()  
        return globals()[targetclass]()  
  
button_obj = ButtonFactory()  
button = ['image', 'input', 'flash']  
for b in button:  
    print button_obj.create_button(b).get_html()
```

The button class helps to create the html tags and the associated html page. The client will not have access to the logic of code and the output represents the creation of html page.

## Output



```
Git CMD  
E:\design-patterns>python factorypattern.py  
<img></img>  
<input></input>  
<obj></obj>  
E:\design-patterns>
```

## Explanation

The python code includes the logic of html tags, which specified value. The end user can have a look on the HTML file created by the Python code.

# Python Design Patterns - Builder

Builder Pattern is a unique design pattern which helps in building complex object using simple objects and uses an algorithmic approach. This design pattern comes under the category of creational pattern. In this design pattern, a builder class builds the final object in step-by-step procedure. This builder is independent of other objects.

## Advantages of Builder Pattern

- It provides clear separation and a unique layer between construction and representation of a specified object created by class.
- It provides better control over construction process of the pattern created.

- It gives the perfect scenario to change the internal representation of objects.

## How to implement builder pattern?

In this section, we will learn how to implement the builder pattern.

```
class Director:
    __builder = None

    def setBuilder(self, builder):
        self.__builder = builder

    def getCar(self):
        car = Car()

        # First goes the body
        body = self.__builder.getBody()
        car.setBody(body)

        # Then engine
        engine = self.__builder.getEngine()
        car.setEngine(engine)

        # And four wheels
        i = 0
        while i < 4:
            wheel = self.__builder.getWheel()
            car.attachWheel(wheel)

            i += 1
        return car

# The whole product
class Car:
    def __init__(self):
        self.__wheels = list()
        self.__engine = None
        self.__body = None

    def setBody(self, body):
        self.__body = body

    def attachWheel(self, wheel):
        self.__wheels.append(wheel)

    def setEngine(self, engine):
        self.__engine = engine
```

```

def specification(self):
    print "body: %s" % self.__body.shape
    print "engine horsepower: %d" % self.__engine.horsepower
    print "tire size: %d\'" % self.__wheels[0].size

class Builder:
    def getWheel(self): pass
    def getEngine(self): pass
    def getBody(self): pass

class JeepBuilder(Builder):

    def getWheel(self):
        wheel = Wheel()
        wheel.size = 22
        return wheel

    def getEngine(self):
        engine = Engine()
        engine.horsepower = 400
        return engine

    def getBody(self):
        body = Body()
        body.shape = "SUV"
        return body

# Car parts
class Wheel:
    size = None

class Engine:
    horsepower = None

class Body:
    shape = None

def main():
    jeepBuilder = JeepBuilder() # initializing the class

    director = Director()

    # Build Jeep
    print "Jeep"
    director.setBuilder(jeepBuilder)

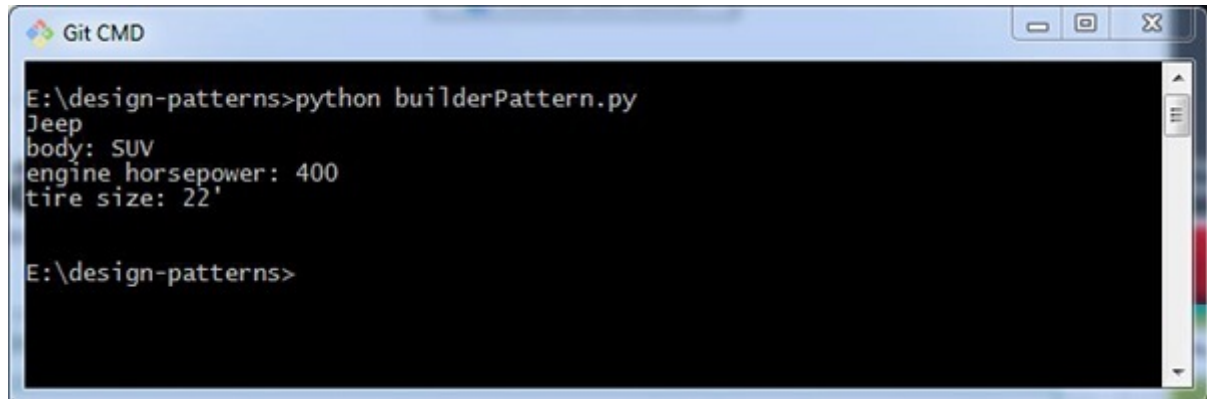
```

```
jeep = director.getCar()
jeep.specification()
print ""

if __name__ == "__main__":
    main()
```

## Output

The above program generates the following output –



```
Git CMD
E:\design-patterns>python builderPattern.py
Jeep
body: SUV
engine horsepower: 400
tire size: 22'

E:\design-patterns>
```

## Python Design Patterns - Prototype

Prototype design pattern helps to hide the complexity of the instances created by the class. The concept of the existing object will differ with that of the new object, which is created from scratch.

The newly copied object may have some changes in the properties if required. This approach saves time and resources that go in for the development of a product.

### How to implement a prototype pattern?

Let us now see how to implement a prototype pattern.

```
import copy

class Prototype:

    _type = None
    _value = None

    def clone(self):
        pass

    def getType(self):
        return self._type
```

```
def getValue(self):
    return self._value

class Type1(Prototype):

    def __init__(self, number):
        self._type = "Type1"
        self._value = number

    def clone(self):
        return copy.copy(self)

class Type2(Prototype):

    """ Concrete prototype. """

    def __init__(self, number):
        self._type = "Type2"
        self._value = number

    def clone(self):
        return copy.copy(self)

class ObjectFactory:

    """ Manages prototypes.
    Static factory, that encapsulates prototype
    initialization and then allows instantiation
    of the classes from these prototypes.
    """

    __type1Value1 = None
    __type1Value2 = None
    __type2Value1 = None
    __type2Value2 = None

    @staticmethod
    def initialize():
        ObjectFactory.__type1Value1 = Type1(1)
        ObjectFactory.__type1Value2 = Type1(2)
        ObjectFactory.__type2Value1 = Type2(1)
        ObjectFactory.__type2Value2 = Type2(2)

    @staticmethod
    def getType1Value1():
        return ObjectFactory.__type1Value1.clone()
```

```
@staticmethod
def getType1Value2():
    return ObjectFactory.__type1Value2.clone()

@staticmethod
def getType2Value1():
    return ObjectFactory.__type2Value1.clone()

@staticmethod
def getType2Value2():
    return ObjectFactory.__type2Value2.clone()

def main():
    ObjectFactory.initialize()

    instance = ObjectFactory.getType1Value1()
    print "%s: %s" % (instance.getType(), instance.getValue())

    instance = ObjectFactory.getType1Value2()
    print "%s: %s" % (instance.getType(), instance.getValue())

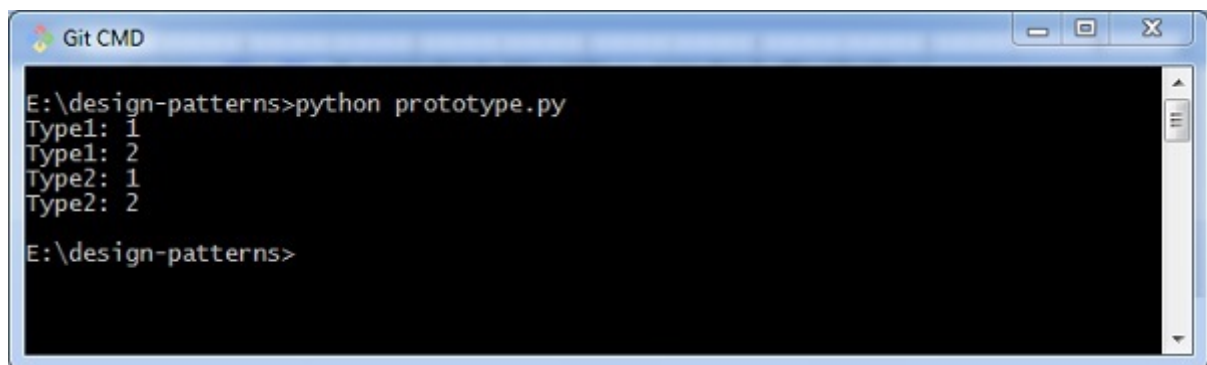
    instance = ObjectFactory.getType2Value1()
    print "%s: %s" % (instance.getType(), instance.getValue())

    instance = ObjectFactory.getType2Value2()
    print "%s: %s" % (instance.getType(), instance.getValue())

if __name__ == "__main__":
    main()
```

## Output

The above program will generate the following output –



```
Git CMD
E:\design-patterns>python prototype.py
Type1: 1
Type1: 2
Type2: 1
Type2: 2
E:\design-patterns>
```

The output helps in creating new objects with the existing ones and it is clearly visible in the output mentioned above.



# Python Design Patterns - Facade

Facade design pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that any subsystem can use.

A facade class knows which subsystem is responsible for a request.

## How to design a facade pattern?

Let us now see how to design a facade pattern.

```
class _IgnitionSystem(object):

    @staticmethod
    def produce_spark():
        return True

class _Engine(object):

    def __init__(self):
        self.revs_per_minute = 0

    def turnon(self):
        self.revs_per_minute = 2000

    def turnoff(self):
        self.revs_per_minute = 0

class _FuelTank(object):

    def __init__(self, level=30):
        self._level = level

    @property
    def level(self):
        return self._level

    @level.setter
    def level(self, level):
        self._level = level

class _DashBoardLight(object):

    def __init__(self, is_on=False):
        self._is_on = is_on
```

```

def __str__(self):
    return self.__class__.__name__

@property
def is_on(self):
    return self._is_on

@is_on.setter
def is_on(self, status):
    self._is_on = status

def status_check(self):
    if self._is_on:
        print("{}: ON".format(str(self)))
    else:
        print("{}: OFF".format(str(self)))

class _HandBrakeLight(_DashboardLight):
    pass

class _FogLampLight(_DashboardLight):
    pass

class _Dashboard(object):

    def __init__(self):
        self.lights = {"handbreak": _HandBrakeLight(), "fog": _FogLampLight()}

    def show(self):
        for light in self.lights.values():
            light.status_check()

# Facade
class Car(object):

    def __init__(self):
        self.ignition_system = _IgnitionSystem()
        self.engine = _Engine()
        self.fuel_tank = _FuelTank()
        self.dashboard = _Dashboard()

    @property
    def km_per_litre(self):
        return 17.0

```

```

def consume_fuel(self, km):
    litres = min(self.fuel_tank.level, km / self.km_per_litre)
    self.fuel_tank.level -= litres

def start(self):
    print("\nStarting...")
    self.dashboard.show()
    if self.ignition_system.produce_spark():
        self.engine.turnon()
    else:
        print("Can't start. Faulty ignition system")

def has_enough_fuel(self, km, km_per_litre):
    litres_needed = km / km_per_litre
    if self.fuel_tank.level > litres_needed:
        return True
    else:
        return False

def drive(self, km = 100):
    print("\n")
    if self.engine.revs_per_minute > 0:
        while self.has_enough_fuel(km, self.km_per_litre):
            self.consume_fuel(km)
            print("Drove {}km".format(km))
            print("{:.2f}l of fuel still left".format(self.fuel_tank.level))
        else:
            print("Can't drive. The Engine is turned off!")

def park(self):
    print("\nParking...")
    self.dashboard.lights["handbreak"].is_on = True
    self.dashboard.show()
    self.engine.turnoff()

def switch_fog_lights(self, status):
    print("\nSwitching {} fog lights...".format(status))
    boolean = True if status == "ON" else False
    self.dashboard.lights["fog"].is_on = boolean
    self.dashboard.show()

def fill_up_tank(self):
    print("\nFuel tank filled up!")
    self.fuel_tank.level = 100

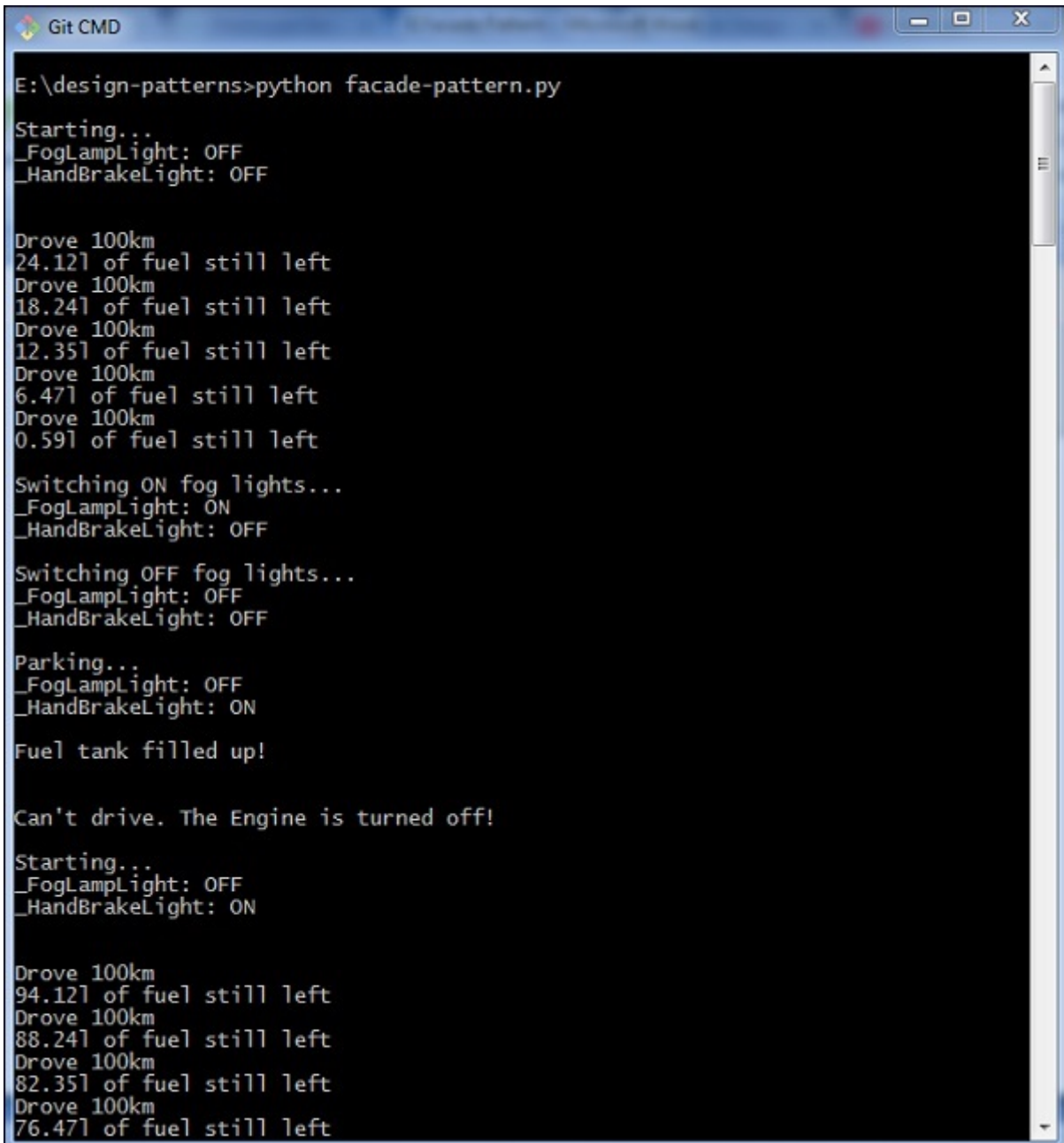
```

*# the main function is the Client*

```
def main():  
    car = Car()  
    car.start()  
    car.drive()  
    car.switch_fog_lights("ON")  
    car.switch_fog_lights("OFF")  
    car.park()  
    car.fill_up_tank()  
    car.drive()  
    car.start()  
    car.drive()  
  
if __name__ == "__main__":  
    main()
```

## Output

The above program generates the following output –



```
Git CMD
E:\design-patterns>python facade-pattern.py

Starting...
_FogLampLight: OFF
_HandBrakeLight: OFF

Drove 100km
24.12l of fuel still left
Drove 100km
18.24l of fuel still left
Drove 100km
12.35l of fuel still left
Drove 100km
6.47l of fuel still left
Drove 100km
0.59l of fuel still left

Switching ON fog lights...
_FogLampLight: ON
_HandBrakeLight: OFF

Switching OFF fog lights...
_FogLampLight: OFF
_HandBrakeLight: OFF

Parking...
_FogLampLight: OFF
_HandBrakeLight: ON

Fuel tank filled up!

Can't drive. The Engine is turned off!

Starting...
_FogLampLight: OFF
_HandBrakeLight: ON

Drove 100km
94.12l of fuel still left
Drove 100km
88.24l of fuel still left
Drove 100km
82.35l of fuel still left
Drove 100km
76.47l of fuel still left
```

## Explanation

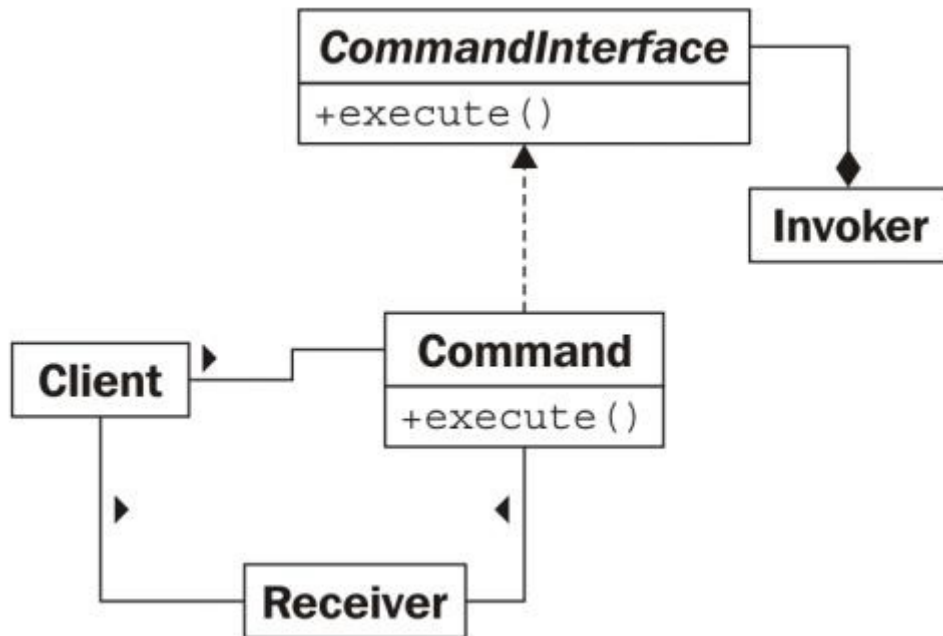
This program is designed with a scenario. It is that of starting the engine of a car or any driving vehicle. If you observe the code, it includes the associated functions to drive, to park and to consume fuel as well.

## Python Design Patterns - Command

Command Pattern adds a level of abstraction between actions and includes an object, which invokes these actions.

In this design pattern, client creates a command object that includes a list of commands to be executed. The command object created implements a specific interface.

Following is the basic architecture of the command pattern –



## How to implement the command pattern?

We will now see how to implement the design pattern.

```

def demo(a,b,c):
    print 'a:',a
    print 'b:',b
    print 'c:',c

class Command:
    def __init__(self, cmd, *args):
        self._cmd=cmd
        self._args=args

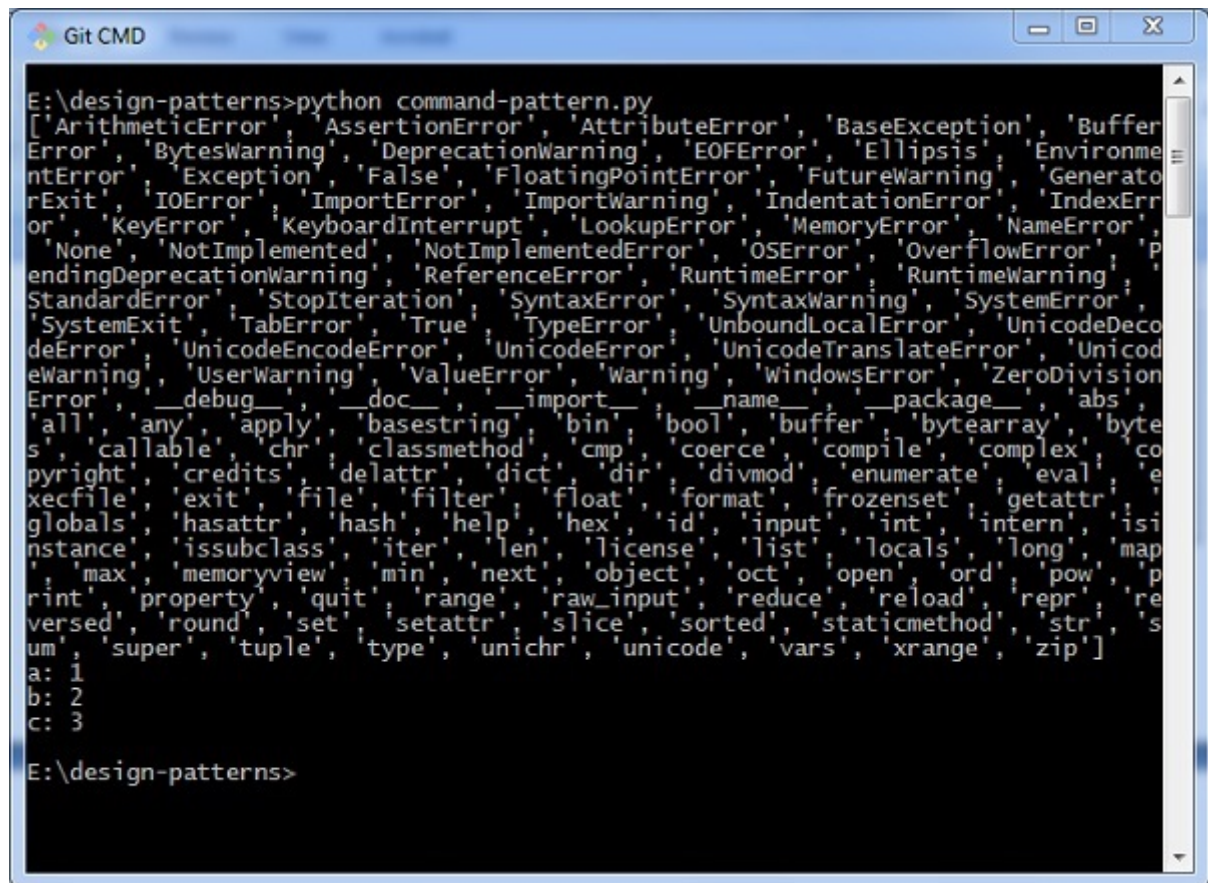
    def __call__(self, *args):
        return apply(self._cmd, self._args+args)

cmd = Command(dir,__builtins__)
print cmd()

cmd = Command(demo,1,2)
cmd(3)
  
```

## Output

The above program generates the following output –



```

E:\design-patterns>python command-pattern.py
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'Buffer
Error', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'Environme
ntError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'Generato
rExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexErr
or', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'P
endingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning',
StandardError', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDeco
deError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'Unicod
eWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivision
Error', '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'apply', 'basestring', 'bin', 'bool', 'buffer', 'bytearray', 'byte
s', 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'co
pyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'e
xecfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset', 'getattr', 'e
globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'intern', 'isi
nstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map
', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'p
rint', 'property', 'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr', 're
versed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 's
um', 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
a: 1
b: 2
c: 3
E:\design-patterns>

```

## Explanation

The output implements all the commands and keywords listed in Python language. It prints the necessary values of the variables.

# Python Design Patterns - Adapter

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class, which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be the case of a card reader, which acts as an adapter between memory card and a laptop. You plug in the memory card into the card reader and the card reader into the laptop so that memory card can be read via the laptop.

The adapter design pattern helps to work classes together. It converts the interface of a class into another interface based on requirement. The pattern includes a speciation a polymorphism which names one name and multiple forms. Say for a shape class which can use as per the requirements gathered.

There are two types of adapter pattern –

## Object Adapter Pattern

This design pattern relies on object implementation. Hence, it is called the Object Adapter Pattern.

## Class Adapter Pattern

This is an alternative way to implement the adapter design pattern. The pattern can be implemented using multiple inheritances.

## How to implement the adapter pattern?

Let us now see how to implement the adapter pattern.

```
class EuropeanSocketInterface:
    def voltage(self): pass

    def live(self): pass
    def neutral(self): pass
    def earth(self): pass

# Adaptee
class Socket(EuropeanSocketInterface):
    def voltage(self):
        return 230

    def live(self):
        return 1

    def neutral(self):
        return -1

    def earth(self):
        return 0

# Target interface
class USASocketInterface:
    def voltage(self): pass
    def live(self): pass
    def neutral(self): pass

# The Adapter
class Adapter(USASocketInterface):
    __socket = None
    def __init__(self, socket):
        self.__socket = socket

    def voltage(self):
```



```
        return 110

    def live(self):
        return self.__socket.live()

    def neutral(self):
        return self.__socket.neutral()

# Client
class ElectricKettle:
    __power = None

    def __init__(self, power):
        self.__power = power

    def boil(self):
        if self.__power.voltage() > 110:
            print "Kettle on fire!"
        else:
            if self.__power.live() == 1 and \
               self.__power.neutral() == -1:
                print "Coffee time!"
            else:
                print "No power."

def main():
    # Plug in
    socket = Socket()
    adapter = Adapter(socket)
    kettle = ElectricKettle(adapter)

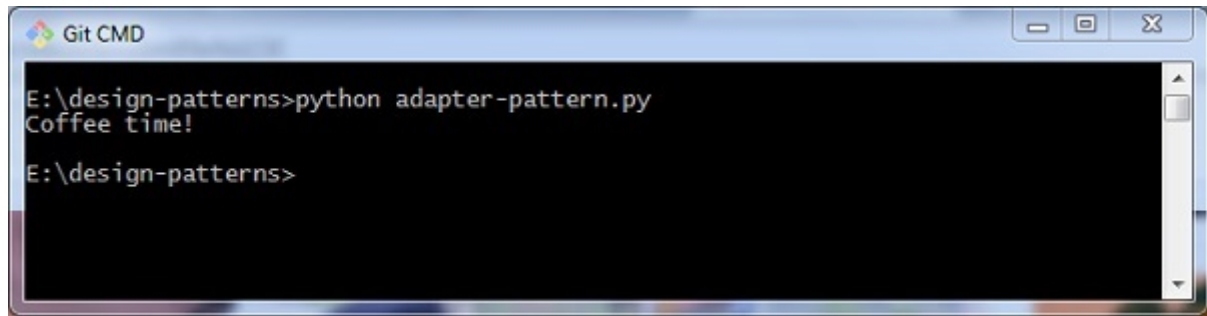
    # Make coffee
    kettle.boil()

    return 0

if __name__ == "__main__":
    main()
```

## Output

The above program generates the following output –

A screenshot of a Git CMD terminal window. The window title is "Git CMD". The command prompt shows the directory "E:\design-patterns". The user has run the command "python adapter-pattern.py", and the output is "Coffee time!". The prompt is now "E:\design-patterns>".

```
Git CMD
E:\design-patterns>python adapter-pattern.py
Coffee time!
E:\design-patterns>
```

## Explanation

The code includes adapter interface with various parameters and attributes. It includes Adaptee along with Target interface that implements all the attributes and displays the output as visible.

# Python Design Patterns - Decorator

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class, which wraps the original class and provides additional functionality keeping the class methods signature intact.

The motive of a decorator pattern is to attach additional responsibilities of an object dynamically.

## How to implement decorator design pattern

The code mentioned below is a simple demonstration of how to implement decorator design pattern in Python. The illustration involves demonstration of a coffee shop in the format of class. The coffee class created is an abstract, which means that it cannot be instantiated.

```
import six
from abc import ABCMeta

@six.add_metaclass(ABCMeta)
class Abstract_Coffee(object):

    def get_cost(self):
        pass

    def get_ingredients(self):
        pass

    def get_tax(self):
        return 0.1*self.get_cost()

class Concrete_Coffee(Abstract_Coffee):
```

```
def get_cost(self):  
    return 1.00
```

```
def get_ingredients(self):  
    return 'coffee'
```

```
@six.add_metaclass(ABCMeta)
```

```
class Abstract_Coffee_Decorator(Abstract_Coffee):
```

```
    def __init__(self, decorated_coffee):  
        self.decorated_coffee = decorated_coffee
```

```
    def get_cost(self):  
        return self.decorated_coffee.get_cost()
```

```
    def get_ingredients(self):  
        return self.decorated_coffee.get_ingredients()
```

```
class Sugar(Abstract_Coffee_Decorator):
```

```
    def __init__(self, decorated_coffee):  
        Abstract_Coffee_Decorator.__init__(self, decorated_coffee)
```

```
    def get_cost(self):  
        return self.decorated_coffee.get_cost()
```

```
    def get_ingredients(self):  
        return self.decorated_coffee.get_ingredients() + ', sugar'
```

```
class Milk(Abstract_Coffee_Decorator):
```

```
    def __init__(self, decorated_coffee):  
        Abstract_Coffee_Decorator.__init__(self, decorated_coffee)
```

```
    def get_cost(self):  
        return self.decorated_coffee.get_cost() + 0.25
```

```
    def get_ingredients(self):  
        return self.decorated_coffee.get_ingredients() + ', milk'
```

```
class Vanilla(Abstract_Coffee_Decorator):
```

```
    def __init__(self, decorated_coffee):  
        Abstract_Coffee_Decorator.__init__(self, decorated_coffee)
```

```
    def get_cost(self):
```

```

    return self.decorated_coffee.get_cost() + 0.75

def get_ingredients(self):
    return self.decorated_coffee.get_ingredients() + ', vanilla'

```

The implementation of the abstract class of the coffee shop is done with a separate file as mentioned below –

```

import coffeeshop

myCoffee = coffeeshop.Concrete_Coffee()
print('Ingredients: '+myCoffee.get_ingredients()+
      '; Cost: '+str(myCoffee.get_cost())+'; sales tax = '+str(myCoffee.get_tax()))

myCoffee = coffeeshop.Milk(myCoffee)
print('Ingredients: '+myCoffee.get_ingredients()+
      '; Cost: '+str(myCoffee.get_cost())+'; sales tax = '+str(myCoffee.get_tax()))

myCoffee = coffeeshop.Vanilla(myCoffee)
print('Ingredients: '+myCoffee.get_ingredients()+
      '; Cost: '+str(myCoffee.get_cost())+'; sales tax = '+str(myCoffee.get_tax()))

myCoffee = coffeeshop.Sugar(myCoffee)
print('Ingredients: '+myCoffee.get_ingredients()+
      '; Cost: '+str(myCoffee.get_cost())+'; sales tax = '+str(myCoffee.get_tax()))

```



## Output

The above program generates the following output –

```

Git CMD
E:\design-patterns>python make_coffee.py
Ingredients: coffee; Cost: 1.0; sales tax = 0.1
Ingredients: coffee, milk; Cost: 1.25; sales tax = 0.125
Ingredients: coffee, milk, vanilla; Cost: 2.0; sales tax = 0.2
Ingredients: coffee, milk, vanilla, sugar; Cost: 2.0; sales tax = 0.2
E:\design-patterns>

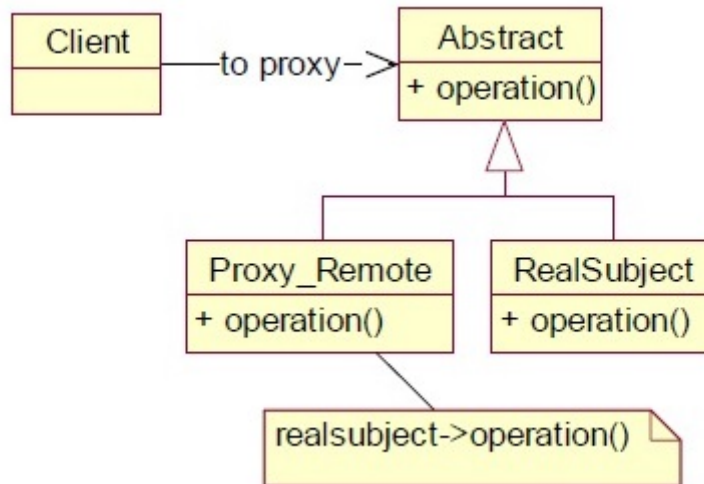
```

## Python Design Patterns - Proxy

The proxy design pattern includes a new object, which is called “Proxy” in place of an existing object which is called the “Real Subject”. The proxy object created of the real subject must be on the same interface in such a way that the client should not get any idea that proxy is used

place of the real object. Requests generated by the client to the proxy are passed through the real subject.

The UML representation of proxy pattern is as follows –



## How to implement the proxy pattern?

Let us now see how to implement the proxy pattern.

```

class Image:
    def __init__( self, filename ):
        self._filename = filename

    def load_image_from_disk( self ):
        print("loading " + self._filename )

    def display_image( self ):
        print("display " + self._filename)

class Proxy:
    def __init__( self, subject ):
        self._subject = subject
        self._proxystate = None

class ProxyImage( Proxy ):
    def display_image( self ):
        if self._proxystate == None:
            self._subject.load_image_from_disk()
            self._proxystate = 1
        print("display " + self._subject._filename )

proxy_image1 = ProxyImage ( Image("HiRes_10Mb_Photo1") )
proxy_image2 = ProxyImage ( Image("HiRes_10Mb_Photo2") )

proxy_image1.display_image() # Loading necessary
  
```

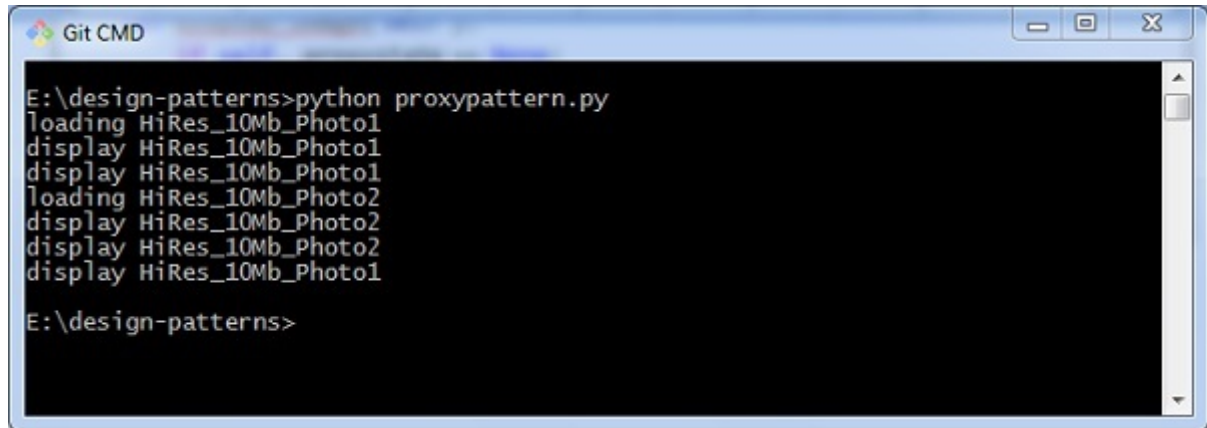
```

proxy_image1.display_image() # Loading unnecessary
proxy_image2.display_image() # Loading necessary
proxy_image2.display_image() # Loading unnecessary
proxy_image1.display_image() # Loading unnecessary

```

## Output

The above program generates the following output –



```

Git CMD
E:\design-patterns>python proxypattern.py
loading HiRes_10Mb_Photo1
display HiRes_10Mb_Photo1
display HiRes_10Mb_Photo1
loading HiRes_10Mb_Photo2
display HiRes_10Mb_Photo2
display HiRes_10Mb_Photo2
display HiRes_10Mb_Photo1
E:\design-patterns>

```

The proxy pattern design helps in replicating the images that we created. The `display_image()` function helps to check if the values are getting printed in the command prompt.

## Chain of Responsibility

The chain of responsibility pattern is used to achieve loose coupling in software where a specified request from the client is passed through a chain of objects included in it. It helps in building a chain of objects. The request enters from one end and moves from one object to another.

This pattern allows an object to send a command without knowing which object will handle the request.

## How to implement the chain of responsibility pattern?

We will now see how to implement the chain of responsibility pattern.

```

class ReportFormat(object):
    PDF = 0
    TEXT = 1
class Report(object):
    def __init__(self, format_):
        self.title = 'Monthly report'
        self.text = ['Things are going', 'really, really well.']
        self.format_ = format_

```

```
class Handler(object):
    def __init__(self):
        self.nextHandler = None

    def handle(self, request):
        self.nextHandler.handle(request)

class PDFHandler(Handler):

    def handle(self, request):
        if request.format_ == ReportFormat.PDF:
            self.output_report(request.title, request.text)
        else:
            super(PDFHandler, self).handle(request)

    def output_report(self, title, text):
        print '<html>'
        print ' <head>'
        print ' <title>%s</title>' % title
        print ' </head>'
        print ' <body>'
        for line in text:
            print ' <p>%s'

' % line
        print ' </body>'
        print '</html>'

class TextHandler(Handler):

    def handle(self, request):
        if request.format_ == ReportFormat.TEXT:
            self.output_report(request.title, request.text)
        else:
            super(TextHandler, self).handle(request)

    def output_report(self, title, text):
        print 5*' ' + title + 5*' '
        for line in text:
            print line

class ErrorHandler(Handler):
    def handle(self, request):
        print "Invalid request"

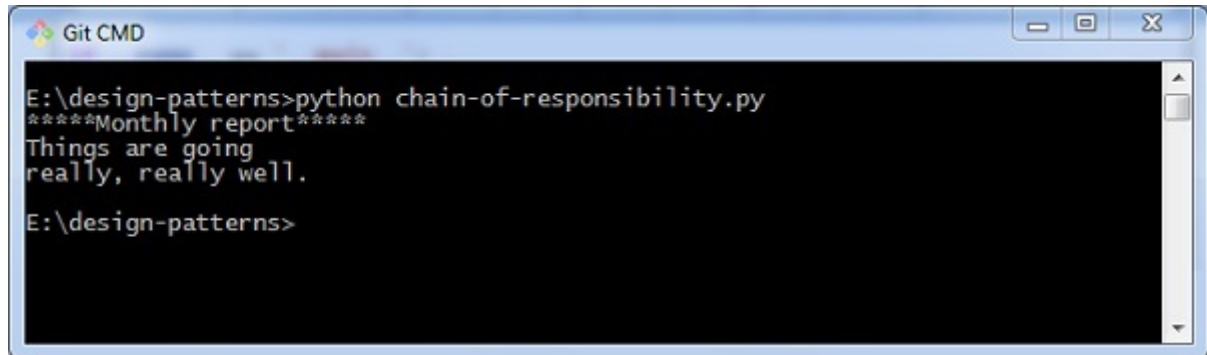
if __name__ == '__main__':
    report = Report(ReportFormat.TEXT)
```

```
pdf_handler = PDFHandler()
text_handler = TextHandler()

pdf_handler.nextHandler = text_handler
text_handler.nextHandler = ErrorHandler()
pdf_handler.handle(report)
```

## Output

The above program generates the following output –



```
Git CMD
E:\design-patterns>python chain-of-responsibility.py
*****Monthly report*****
Things are going
really, really well.
E:\design-patterns>
```

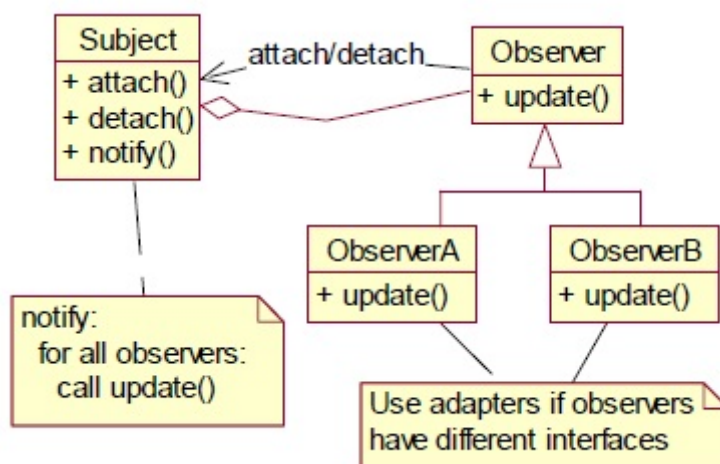
## Explanation

The above code creates a report for monthly tasks where it sends commands through each function. It takes two handlers – for PDF and for text. It prints the output once the required object executes each function.

# Python Design Patterns - Observer

In this pattern, objects are represented as observers that wait for an event to trigger. An observer attaches to the subject once the specified event occurs. As the event occurs, the subject tells the observers that it has occurred.

The following UML diagram represents the observer pattern –



## How to implement the observer pattern?



Let us now see how to implement the observer pattern.

```
import threading
import time
import pdb

class Downloader(threading.Thread):

    def run(self):
        print 'downloading'
        for i in range(1,5):
            self.i = i
            time.sleep(2)
            print 'unfunf'
        return 'hello world'

class Worker(threading.Thread):
    def run(self):
        for i in range(1,5):
            print 'worker running: %i (%i)' % (i, t.i)
            time.sleep(1)
            t.join()

            print 'done'

t = Downloader()
t.start()

time.sleep(1)

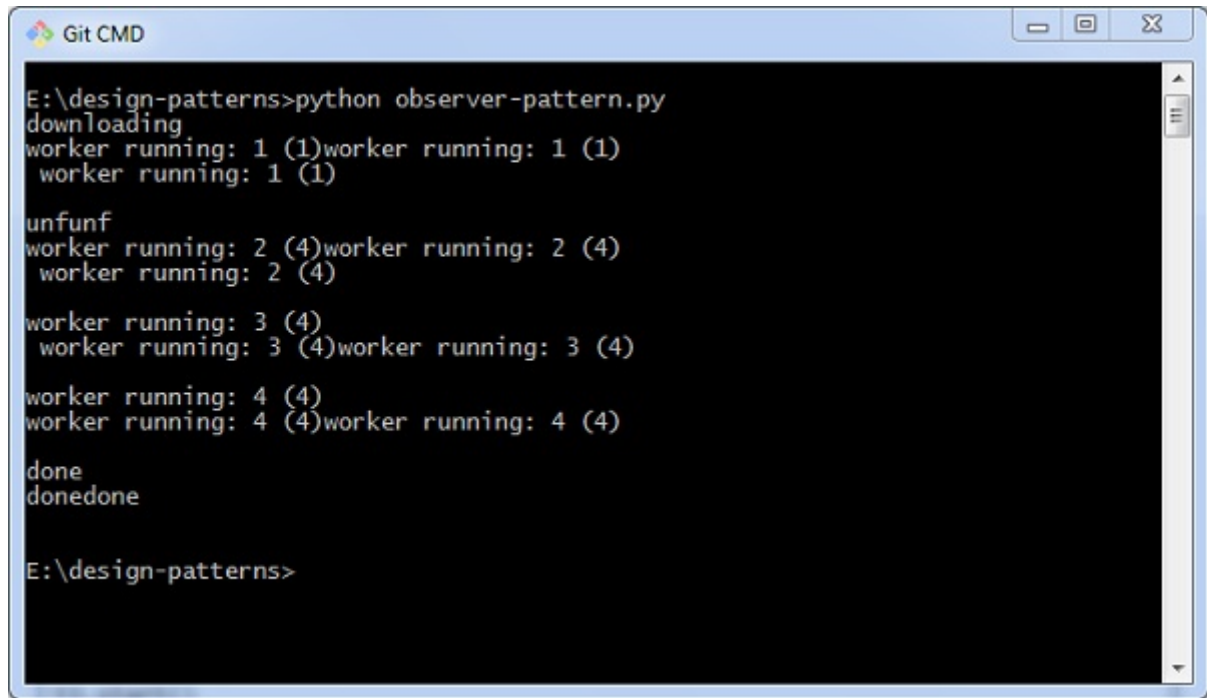
t1 = Worker()
t1.start()

t2 = Worker()
t2.start()

t3 = Worker()
t3.start()
```

## Output

The above program generates the following output –



```

E:\design-patterns>python observer-pattern.py
downloading
worker running: 1 (1)worker running: 1 (1)
worker running: 1 (1)

unfunf
worker running: 2 (4)worker running: 2 (4)
worker running: 2 (4)

worker running: 3 (4)
worker running: 3 (4)worker running: 3 (4)

worker running: 4 (4)
worker running: 4 (4)worker running: 4 (4)

done
done
done
done

E:\design-patterns>

```

## Explanation

The above code explains the procedure of downloading a particular result. As per the observer pattern logic, every object is treated as observer. It prints the output when event is triggered.

# Python Design Patterns - State

It provides a module for state machines, which are implemented using subclasses, derived from a specified state machine class. The methods are state independent and cause transitions declared using decorators.

## How to implement the state pattern?

The basic implementation of state pattern is shown below –

```

class ComputerState(object):

    name = "state"
    allowed = []

    def switch(self, state):
        """ Switch to new state """
        if state.name in self.allowed:
            print 'Current:',self,' => switched to new state',state.name
            self.__class__ = state
        else:
            print 'Current:',self,' => switching to',state.name,'not possible.'

    def __str__(self):
        return self.name

```

```

class Off(ComputerState):
    name = "off"
    allowed = ['on']

class On(ComputerState):
    """ State of being powered on and working """
    name = "on"
    allowed = ['off', 'suspend', 'hibernate']

class Suspend(ComputerState):
    """ State of being in suspended mode after switched on """
    name = "suspend"
    allowed = ['on']

class Hibernate(ComputerState):
    """ State of being in hibernation after powered on """
    name = "hibernate"
    allowed = ['on']

class Computer(object):
    """ A class representing a computer """

    def __init__(self, model='HP'):
        self.model = model
        # State of the computer - default is off.
        self.state = Off()

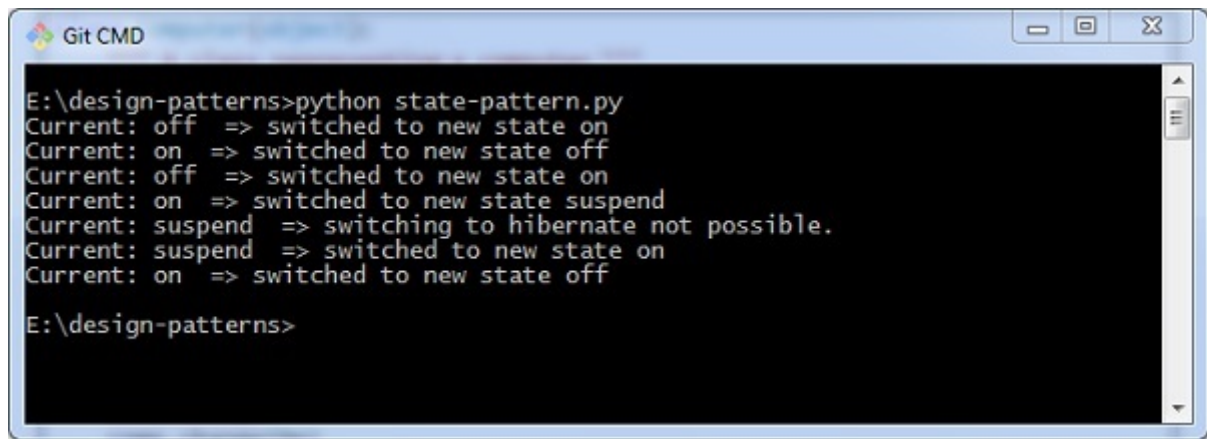
    def change(self, state):
        """ Change state """
        self.state.switch(state)

if __name__ == "__main__":
    comp = Computer()
    comp.change(On)
    comp.change(Off)
    comp.change(On)
    comp.change(Suspend)
    comp.change(Hibernate)
    comp.change(On)
    comp.change(Off)

```

## Output

The above program generates the following output –



```

E:\design-patterns>python state-pattern.py
Current: off => switched to new state on
Current: on => switched to new state off
Current: off => switched to new state on
Current: on => switched to new state suspend
Current: suspend => switching to hibernate not possible.
Current: suspend => switched to new state on
Current: on => switched to new state off

E:\design-patterns>

```

## Python Design Patterns - Strategy

The strategy pattern is a type of behavioral pattern. The main goal of strategy pattern is to enable client to choose from different algorithms or procedures to complete the specified task. Different algorithms can be swapped in and out without any complications for the mentioned task.

This pattern can be used to improve flexibility when external resources are accessed.

### How to implement the strategy pattern?

The program shown below helps in implementing the strategy pattern.

```

import types

class StrategyExample:
    def __init__(self, func = None):
        self.name = 'Strategy Example 0'
        if func is not None:
            self.execute = types.MethodType(func, self)

    def execute(self):
        print(self.name)

def execute_replacement1(self):
    print(self.name + 'from execute 1')

def execute_replacement2(self):
    print(self.name + 'from execute 2')

if __name__ == '__main__':
    strat0 = StrategyExample()
    strat1 = StrategyExample(execute_replacement1)
    strat1.name = 'Strategy Example 1'
    strat2 = StrategyExample(execute_replacement2)

```

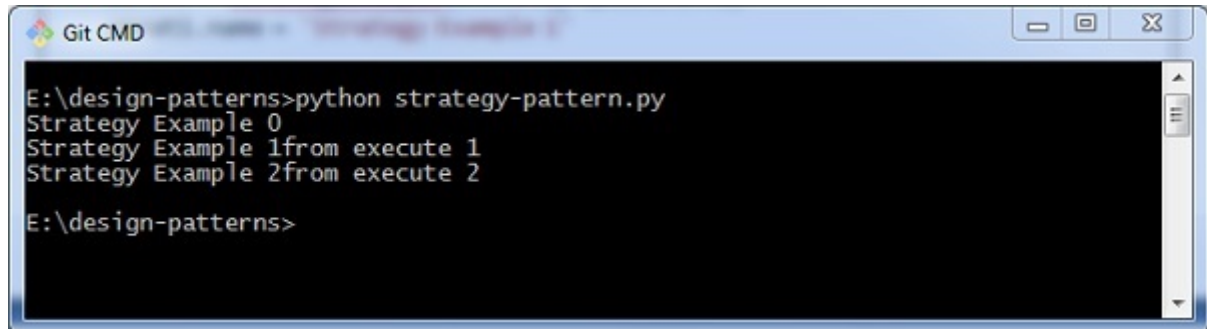
```

strat2.name = 'Strategy Example 2'
strat0.execute()
strat1.execute()
strat2.execute()

```

## Output

The above program generates the following output –



```

Git CMD
E:\design-patterns>python strategy-pattern.py
Strategy Example 0
Strategy Example 1from execute 1
Strategy Example 2from execute 2
E:\design-patterns>

```

## Explanation

It provides a list of strategies from the functions, which execute the output. The major focus of this behavior pattern is behavior.

# Python Design Patterns - Template

A template pattern defines a basic algorithm in a base class using abstract operation where subclasses override the concrete behavior. The template pattern keeps the outline of algorithm in a separate method. This method is referred as the template method.

Following are the different features of the template pattern –

- It defines the skeleton of algorithm in an operation
- It includes subclasses, which redefine certain steps of an algorithm.

```
class MakeMeal:
```

```

    def prepare(self): pass
    def cook(self): pass
    def eat(self): pass

```

```

    def go(self):
        self.prepare()
        self.cook()
        self.eat()

```

```
class MakePizza(MakeMeal):
```

```
def prepare(self):
    print "Prepare Pizza"

def cook(self):
    print "Cook Pizza"

def eat(self):
    print "Eat Pizza"

class MakeTea(MakeMeal):
    def prepare(self):
        print "Prepare Tea"

    def cook(self):
        print "Cook Tea"

    def eat(self):
        print "Eat Tea"

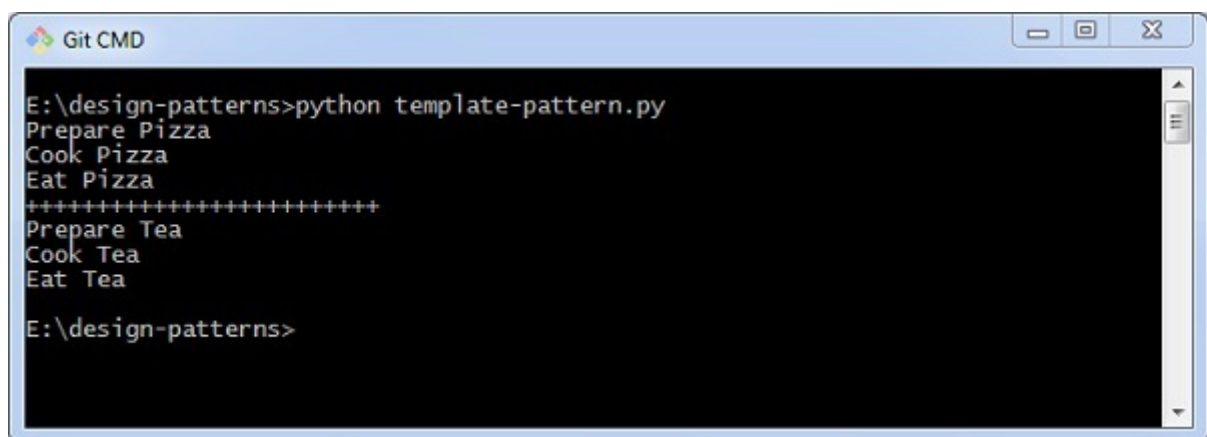
makePizza = MakePizza()
makePizza.go()

print 25*"+"

makeTea = MakeTea()
makeTea.go()
```

## Output

The above program generates the following output –



```
Git CMD
E:\design-patterns>python template-pattern.py
Prepare Pizza
Cook Pizza
Eat Pizza
+++++
Prepare Tea
Cook Tea
Eat Tea
E:\design-patterns>
```

## Explanation

This code creates a template to prepare meal. Here, each parameter represents the attribute to create a part of meal like tea, pizza, etc.

The output represents the visualization of attributes.

# Python Design Patterns - Flyweight

The flyweight pattern comes under the structural design patterns category. It provides a way to decrease object count. It includes various features that help in improving application structure. The most important feature of the flyweight objects is immutable. This means that they cannot be modified once constructed. The pattern uses a HashMap to store reference objects.

## How to implement the flyweight pattern?

The following program helps in implementing the flyweight pattern –

```
class ComplexGenetics(object):
    def __init__(self):
        pass

    def genes(self, gene_code):
        return "ComplexPatter[%s]TooHugeinSize" % (gene_code)
class Families(object):
    family = {}

    def __new__(cls, name, family_id):
        try:
            id = cls.family[family_id]
        except KeyError:
            id = object.__new__(cls)
            cls.family[family_id] = id
        return id

    def set_genetic_info(self, genetic_info):
        cg = ComplexGenetics()
        self.genetic_info = cg.genes(genetic_info)

    def get_genetic_info(self):
        return (self.genetic_info)

def test():
    data = (('a', 1, 'ATAG'), ('a', 2, 'AAGT'), ('b', 1, 'ATAG'))
    family_objects = []
    for i in data:
        obj = Families(i[0], i[1])
        obj.set_genetic_info(i[2])
        family_objects.append(obj)

    for i in family_objects:
```

```

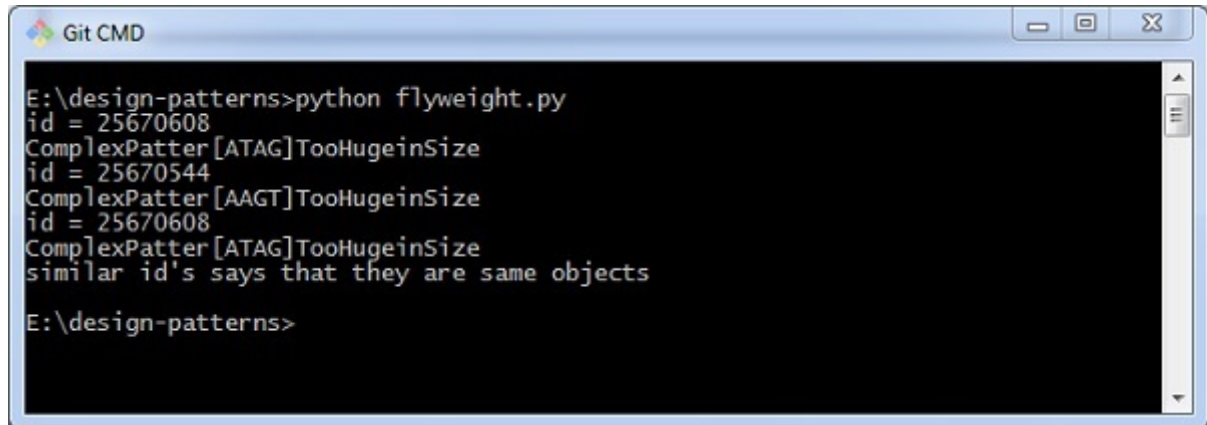
    print "id = " + str(id(i))
    print i.get_genetic_info()
    print "similar id's says that they are same objects "

if __name__ == '__main__':
    test()

```

## Output

The above program generates the following output –



```

E:\design-patterns>python flyweight.py
id = 25670608
ComplexPatter[ATAG]TooHugeinSize
id = 25670544
ComplexPatter[AAGT]TooHugeinSize
id = 25670608
ComplexPatter[ATAG]TooHugeinSize
similar id's says that they are same objects
E:\design-patterns>

```

## Python Design Patterns - Abstract Factory

The abstract factory pattern is also called factory of factories. This design pattern comes under the creational design pattern category. It provides one of the best ways to create an object.

It includes an interface, which is responsible for creating objects related to Factory.

## How to implement the abstract factory pattern?

The following program helps in implementing the abstract factory pattern.

```

class Window:
    __toolkit = ""
    __purpose = ""

    def __init__(self, toolkit, purpose):
        self.__toolkit = toolkit
        self.__purpose = purpose

    def getToolkit(self):
        return self.__toolkit

    def getType(self):
        return self.__purpose

```



```
class GtkToolboxWindow(Window):
    def __init__(self):
        Window.__init__(self, "Gtk", "ToolboxWindow")

class GtkLayersWindow(Window):
    def __init__(self):
        Window.__init__(self, "Gtk", "LayersWindow")

class GtkMainWindow(Window):
    def __init__(self):
        Window.__init__(self, "Gtk", "MainWindow")

class QtToolboxWindow(Window):
    def __init__(self):
        Window.__init__(self, "Qt", "ToolboxWindow")

class QtLayersWindow(Window):
    def __init__(self):
        Window.__init__(self, "Qt", "LayersWindow")

class QtMainWindow(Window):
    def __init__(self):
        Window.__init__(self, "Qt", "MainWindow")

# Abstract factory class
class UIFactory:
    def getToolboxWindow(self): pass
    def getLayersWindow(self): pass
    def getMainWindow(self): pass

class GtkUIFactory(UIFactory):
    def getToolboxWindow(self):
        return GtkToolboxWindow()
    def getLayersWindow(self):
        return GtkLayersWindow()
    def getMainWindow(self):
        return GtkMainWindow()

class QtUIFactory(UIFactory):
    def getToolboxWindow(self):
        return QtToolboxWindow()
    def getLayersWindow(self):
        return QtLayersWindow()
    def getMainWindow(self):
        return QtMainWindow()
```

```

if __name__ == "__main__":
    gnome = True
    kde = not gnome

    if gnome:
        ui = GtkUIFactory()
    elif kde:
        ui = QtUIFactory()

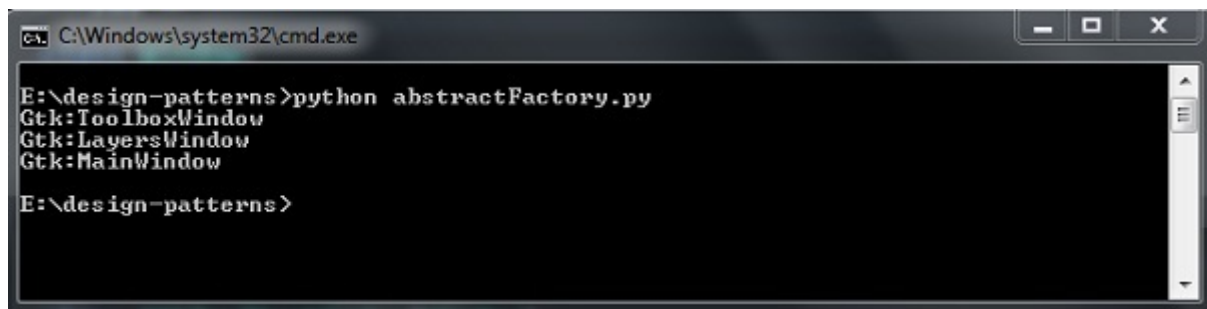
    toolbox = ui.getToolboxWindow()
    layers = ui.getLayersWindow()
    main = ui.getMainWindow()

    print "%s:%s" % (toolbox.getToolkit(), toolbox.getType())
    print "%s:%s" % (layers.getToolkit(), layers.getType())
    print "%s:%s" % (main.getToolkit(), main.getType())

```

## Output

The above program generates the following output –



```

C:\Windows\system32\cmd.exe
E:\design-patterns>python abstractFactory.py
Gtk:ToolboxWindow
Gtk:LayersWindow
Gtk:MainWindow
E:\design-patterns>

```

## Explanation

In the above program, the abstract factory creates objects for each window. It calls for each method, which executes the output as expected.

# Python Design Patterns - Object Oriented

The object oriented pattern is the most commonly used pattern. This pattern can be found in almost every programming language.

## How to implement the object oriented pattern?

Let us now see how to implement the object oriented pattern.

```

class Parrot:
    # class attribute
    species = "bird"

```

```
# instance attribute
def __init__(self, name, age):
    self.name = name
    self.age = age

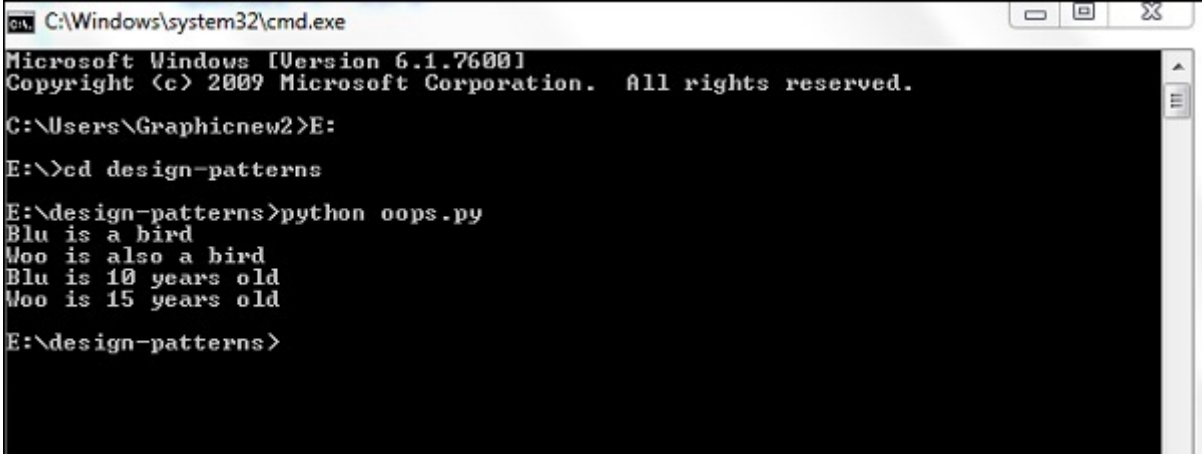
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

## Output

The above program generates the following output



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Graphicnew2>E:
E:\>cd design-patterns
E:\design-patterns>python oops.py
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
E:\design-patterns>
```

## Explanation

The code includes class attribute and instance attributes, which are printed as per the requirement of the output. There are various features that form part of the object oriented pattern. The features are explained in the next chapter.

# Object Oriented Concepts Implementation

In this chapter, we will focus on patterns using object oriented concepts and its implementation in Python. When we design our programs around blocks of statements, which manipulate the data around functions, it is called procedure-oriented programming. In object-oriented programming, there are two main instances called classes and objects.

# How to implement classes and object variables?

The implementation of classes and object variables are as follows –

```
class Robot:
    population = 0

    def __init__(self, name):
        self.name = name
        print("(Initializing {})".format(self.name))
        Robot.population += 1

    def die(self):
        print("{} is being destroyed!".format(self.name))
        Robot.population -= 1
        if Robot.population == 0:
            print("{} was the last one.".format(self.name))
        else:
            print("There are still {:d} robots working.".format(
                Robot.population))

    def say_hi(self):
        print("Greetings, my masters call me {}.".format(self.name))

    @classmethod
    def how_many(cls):
        print("We have {:d} robots.".format(cls.population))

droid1 = Robot("R2-D2")
droid1.say_hi()
Robot.how_many()

droid2 = Robot("C-3PO")
droid2.say_hi()
Robot.how_many()

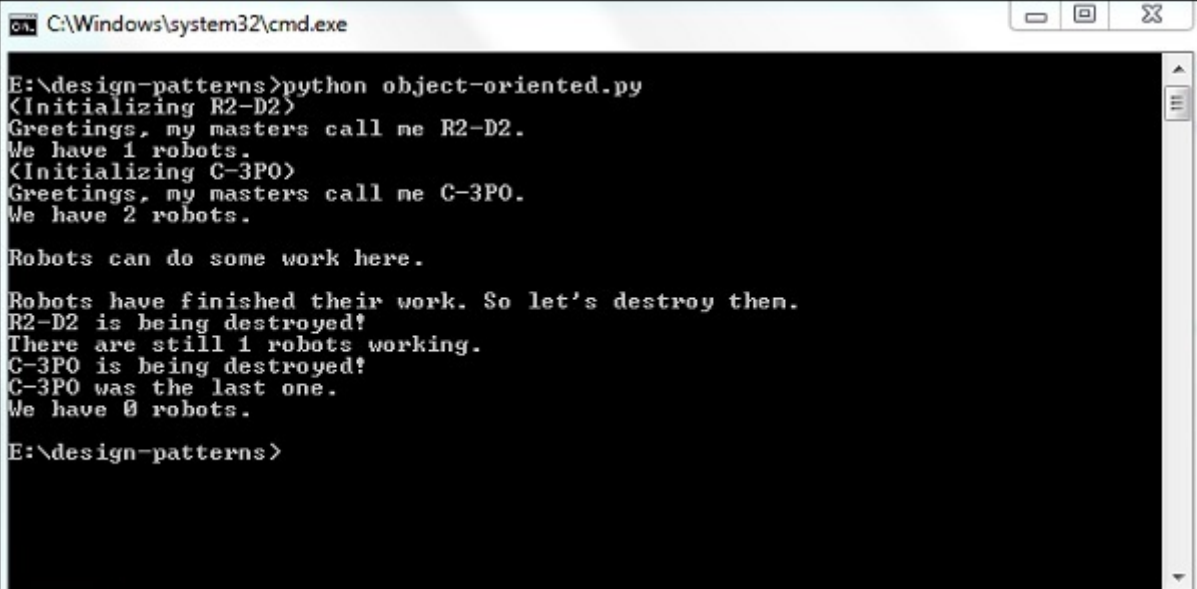
print("\nRobots can do some work here.\n")

print("Robots have finished their work. So let's destroy them.")
droid1.die()
droid2.die()

Robot.how_many()
```

## Output

The above program generates the following output –



```

C:\Windows\system32\cmd.exe

E:\design-patterns>python object-oriented.py
<Initializing R2-D2>
Greetings, my masters call me R2-D2.
We have 1 robots.
<Initializing C-3PO>
Greetings, my masters call me C-3PO.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.

E:\design-patterns>

```

## Explanation

This illustration helps to demonstrate the nature of class and object variables.

- “population” belongs to the “Robot” class. Hence, it is referred to as a class variable or object.
- Here, we refer to the population class variable as Robot.population and not as self.population.

## Python Design Patterns - Iterator

The iterator design pattern falls under the behavioral design patterns category. Developers come across the iterator pattern in almost every programming language. This pattern is used in such a way that it helps to access the elements of a collection (class) in sequential manner without understanding the underlying layer design.

### How to implement the iterator pattern?

We will now see how to implement the iterator pattern.

```

import time

def fib():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b

g = fib()

try:

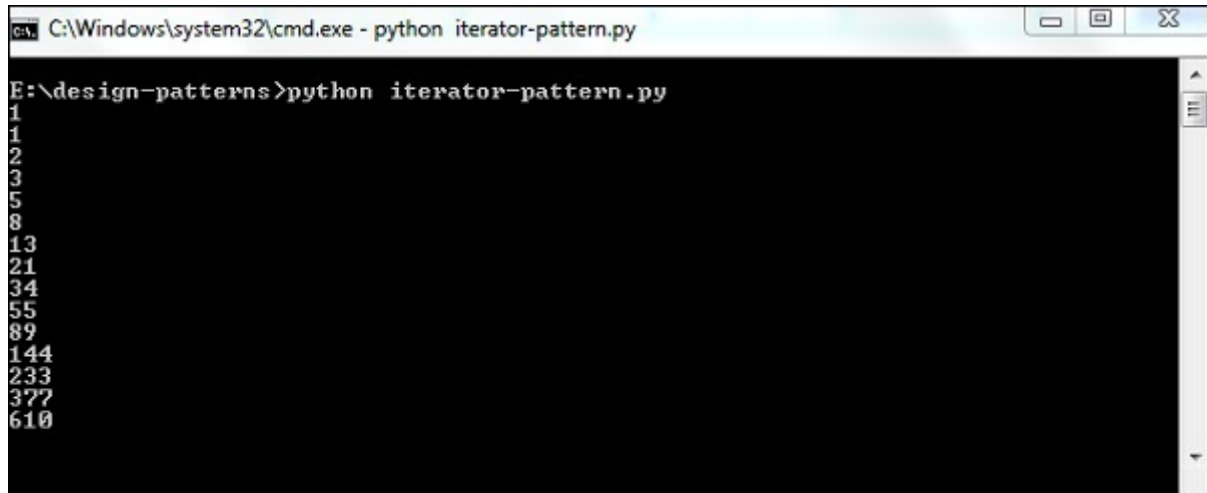
```

```
for e in g:
    print(e)
    time.sleep(1)

except KeyboardInterrupt:
    print("Calculation stopped")
```

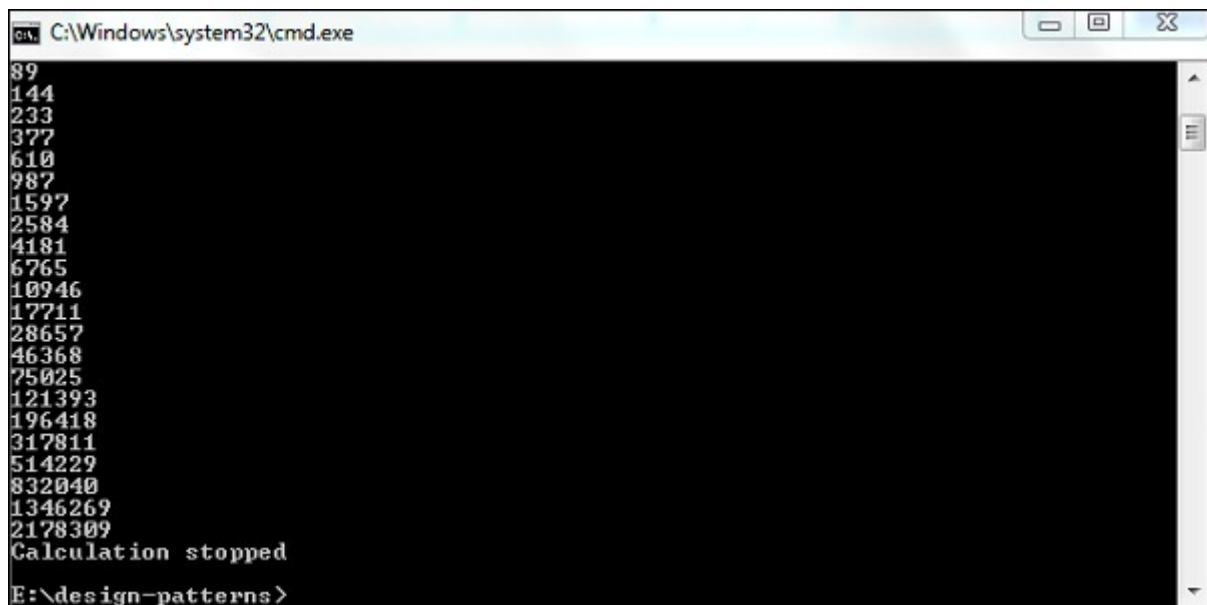
## Output

The above program generates the following output –



```
C:\Windows\system32\cmd.exe - python iterator-pattern.py
E:\design-patterns>python iterator-pattern.py
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
```

If you focus on the pattern, Fibonacci series is printed with the iterator pattern. On forceful termination of user, the following output is printed –



```
C:\Windows\system32\cmd.exe
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
Calculation stopped
E:\design-patterns>
```

## Explanation

This python code follows the iterator pattern. Here, the increment operators are used to start the count. The count ends on forceful termination by the user.

# Python Design Patterns - Dictionaries

Dictionaries are the data structures, which include a key value combination. These are widely used in place of JSON – JavaScript Object Notation. Dictionaries are used for API (Application Programming Interface) programming. A dictionary maps a set of objects to another set of objects. Dictionaries are mutable; this means they can be changed as and when needed based on the requirements.

## How to implement dictionaries in Python?

The following program shows the basic implementation of dictionaries in Python starting from its creation to its implementation.

```
# Create a new dictionary
d = dict() # or d = {}

# Add a key - value pairs to dictionary
d['xyz'] = 123
d['abc'] = 345

# print the whole dictionary
print(d)

# print only the keys
print(d.keys())

# print only values
print(d.values())

# iterate over dictionary
for i in d :
    print("%s %d" %(i, d[i]))

# another method of iteration
for index, value in enumerate(d):
    print (index, value , d[value])

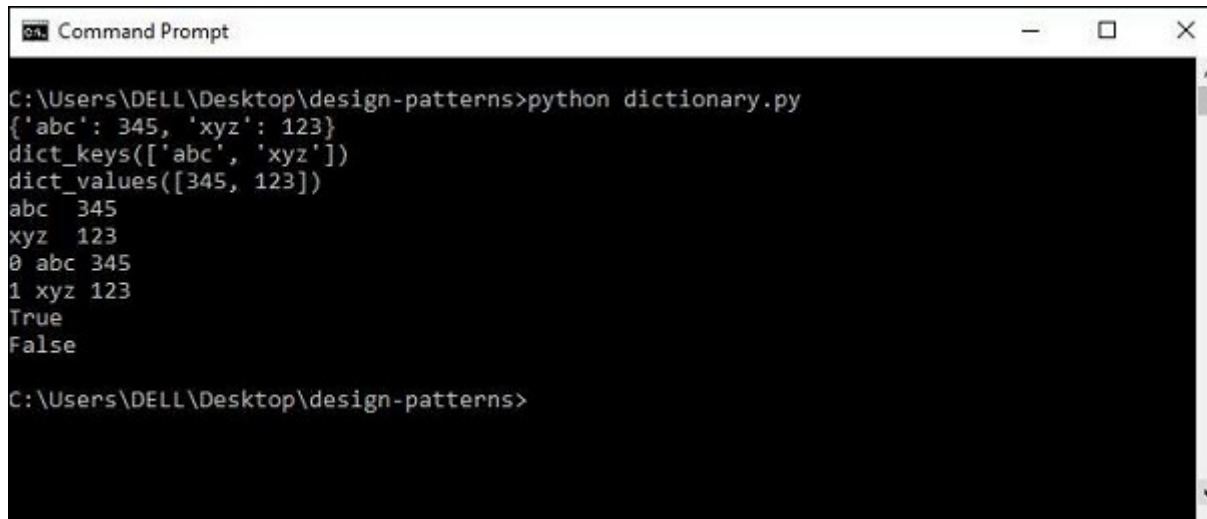
# check if key exist 23. Python Data Structure -print('xyz' in d)

# delete the key-value pair
del d['xyz']

# check again
print("xyz" in d)
```

## Output

The above program generates the following output –



```
Command Prompt
C:\Users\DELL\Desktop\design-patterns>python dictionary.py
{'abc': 345, 'xyz': 123}
dict_keys(['abc', 'xyz'])
dict_values([345, 123])
abc 345
xyz 123
0 abc 345
1 xyz 123
True
False
C:\Users\DELL\Desktop\design-patterns>
```

**Note** –There are drawbacks related to the implementation of dictionaries in Python.

## Drawback

Dictionaries do not support the sequence operation of the sequence data types like strings, tuples and lists. These belong to the built-in mapping type.

# Lists Data Structure

The Lists data structure is a versatile datatype in Python, which can be written as a list of comma separated values between square brackets.

## Syntax

Here is the basic syntax for the structure –

```
List_name = [ elements ];
```

If you observe, the syntax is declared like arrays with the only difference that lists can include elements with different data types. The arrays include elements of the same data type. A list can contain a combination of strings, integers and objects. Lists can be used for the implementation of stacks and queues.

Lists are mutable. These can be changed as and when needed.

## How to implement lists?

The following program shows the implementations of lists –

```
my_list = ['p','r','o','b','e']
# Output: p
print(my_list[0])
```



```
# Output: o
print(my_list[2])

# Output: e
print(my_list[4])

# Error! Only integer can be used for indexing
# my_list[4.0]

# Nested List
n_list = ["Happy", [2,0,1,5]]

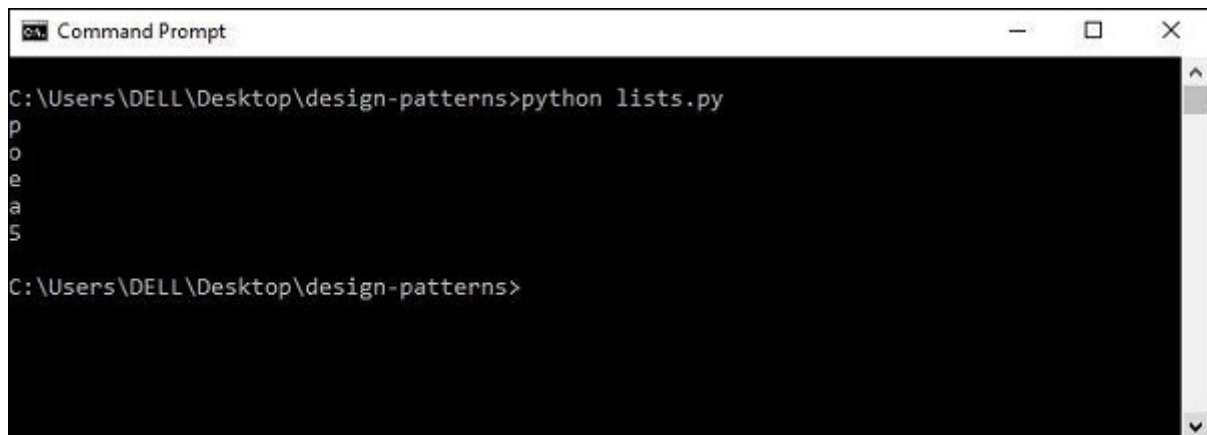
# Nested indexing

# Output: a
print(n_list[0][1])

# Output: 5
print(n_list[1][3])
```

## Output

The above program generates the following output –



```
Command Prompt
C:\Users\DELL\Desktop\design-patterns>python lists.py
p
o
e
a
5
C:\Users\DELL\Desktop\design-patterns>
```

The built-in functions of Python lists are as follows –

- **Append()**– It adds element to the end of list.
- **Extend()**– It adds elements of the list to another list.
- **Insert()**– It inserts an item to the defined index.
- **Remove()**– It deletes the element from the specified list.
- **Reverse()**– It reverses the elements in list.
- **sort()** – It helps to sort elements in chronological order.

# Python Design Patterns - Sets

A set can be defined as unordered collection that is iterable, mutable and there is no inclusion of duplicate elements in it. In Python, set class is a notation of mathematical set. The main advantage of using a set is that it includes highly optimized method for checking specific element.

Python includes a separate category called frozen sets. These sets are immutable objects that only support methods and operators that produce a required result.

## How to implement sets?

The following program helps in the implementation of sets –

```
# Set in Python

# Creating two sets
set1 = set()
set2 = set()

# Adding elements to set1
for i in range(1, 6):
    set1.add(i)

# Adding elements to set2
for i in range(3, 8):
    set2.add(i)

print("Set1 = ", set1)
print("Set2 = ", set2)
print("\n")

# Union of set1 and set2
set3 = set1 | set2# set1.union(set2)
print("Union of Set1 & Set2: Set3 = ", set3)

# Intersection of set1 and set2
set4 = set1 & set2# set1.intersection(set2)
print("Intersection of Set1 & Set2: Set4 = ", set4)
print("\n")

# Checking relation between set3 and set4
if set3 > set4: # set3.issuperset(set4)
    print("Set3 is superset of Set4")
elif set3 < set4: # set3.issubset(set4)
```

```
print("Set3 is subset of Set4")
else : # set3 == set4
    print("Set3 is same as Set4")

# displaying relation between set4 and set3
if set4 < set3: # set4.issubset(set3)
    print("Set4 is subset of Set3")
    print("\n")

# difference between set3 and set4
set5 = set3 - set4
print("Elements in Set3 and not in Set4: Set5 = ", set5)
print("\n")

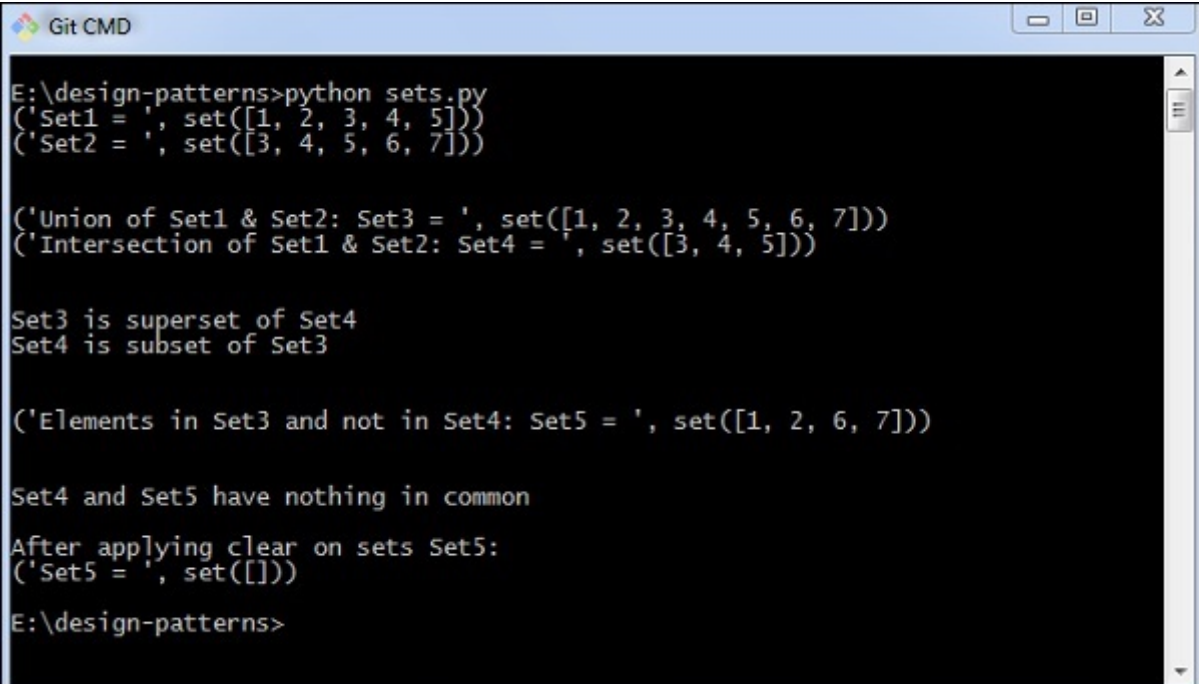
# checkv if set4 and set5 are disjoint sets
if set4.isdisjoint(set5):
    print("Set4 and Set5 have nothing in common\n")

# Removing all the values of set5
set5.clear()

print("After applying clear on sets Set5: ")
print("Set5 = ", set5)
```

## Output

The above program generates the following output –



```
Git CMD
E:\design-patterns>python sets.py
('Set1 = ', set([1, 2, 3, 4, 5]))
('Set2 = ', set([3, 4, 5, 6, 7]))

('Union of Set1 & Set2: Set3 = ', set([1, 2, 3, 4, 5, 6, 7]))
('Intersection of Set1 & Set2: Set4 = ', set([3, 4, 5]))

Set3 is superset of Set4
Set4 is subset of Set3

('Elements in Set3 and not in Set4: Set5 = ', set([1, 2, 6, 7]))

Set4 and Set5 have nothing in common
After applying clear on sets Set5:
('Set5 = ', set([]))
E:\design-patterns>
```

The frozen set can be demonstrated using the following program –

```
normal_set = set(["a", "b", "c"])

# Adding an element to normal set is fine
normal_set.add("d")

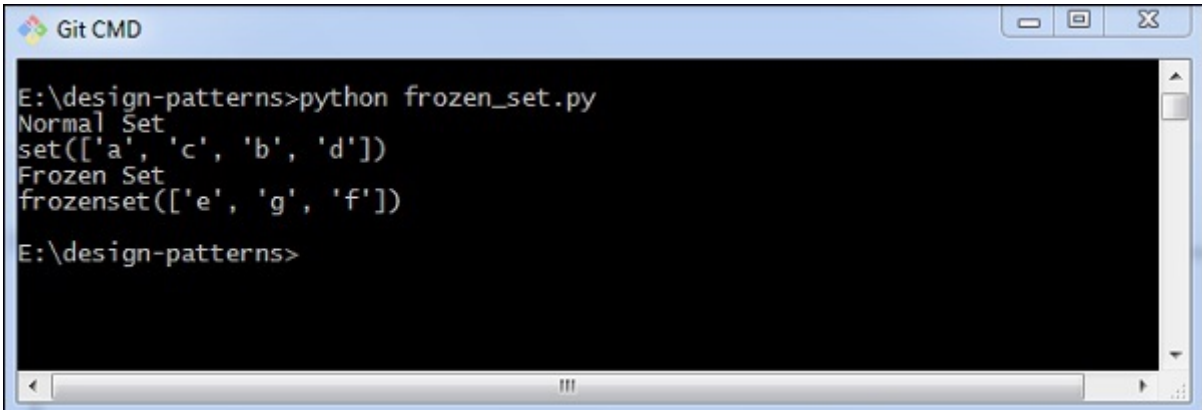
print("Normal Set")
print(normal_set)

# A frozen set
frozen_set = frozenset(["e", "f", "g"])

print("Frozen Set")
print(frozen_set)
```

## Output

The above program generates the following output –



```
Git CMD
E:\design-patterns>python frozen_set.py
Normal Set
set(['a', 'c', 'b', 'd'])
Frozen Set
frozenset(['e', 'g', 'f'])
E:\design-patterns>
```

## Python Design Patterns - Queues

Queue is a collection of objects, which define a simple data structure following the FIFO (Fast In Fast Out) and the LIFO (Last In First Out) procedures. The insert and delete operations are referred as **enqueue** and **dequeue** operations.

Queues do not allow random access to the objects they contain.

### How to implement the FIFO procedure?

The following program helps in the implementation of FIFO –

```
import Queue

q = Queue.Queue()

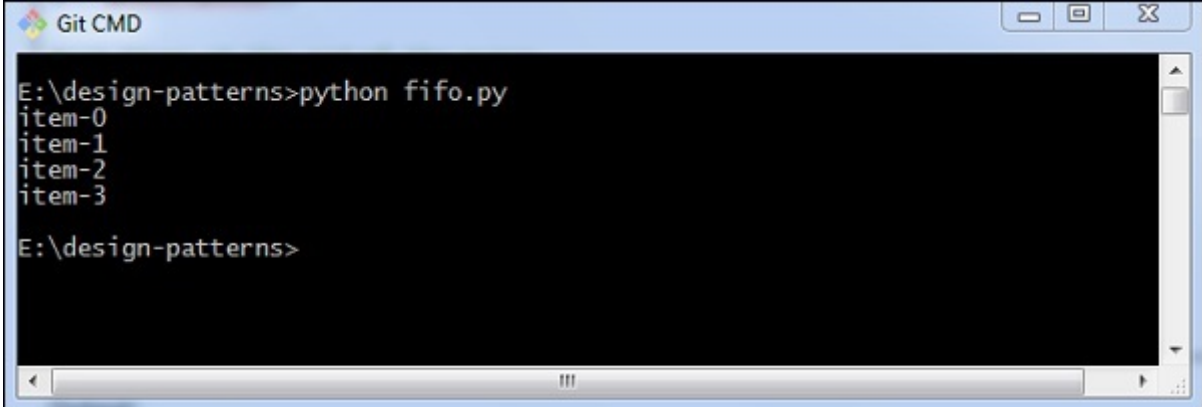
#put items at the end of the queue
```

```
for x in range(4):
    q.put("item-" + str(x))

#remove items from the head of the queue
while not q.empty():
    print q.get()
```

## Output

The above program generates the following output –



```
Git CMD
E:\design-patterns>python fifo.py
item-0
item-1
item-2
item-3
E:\design-patterns>
```

## How to implement the LIFO procedure?

The following program helps in the implementation of the LIFO procedure –

```
import Queue

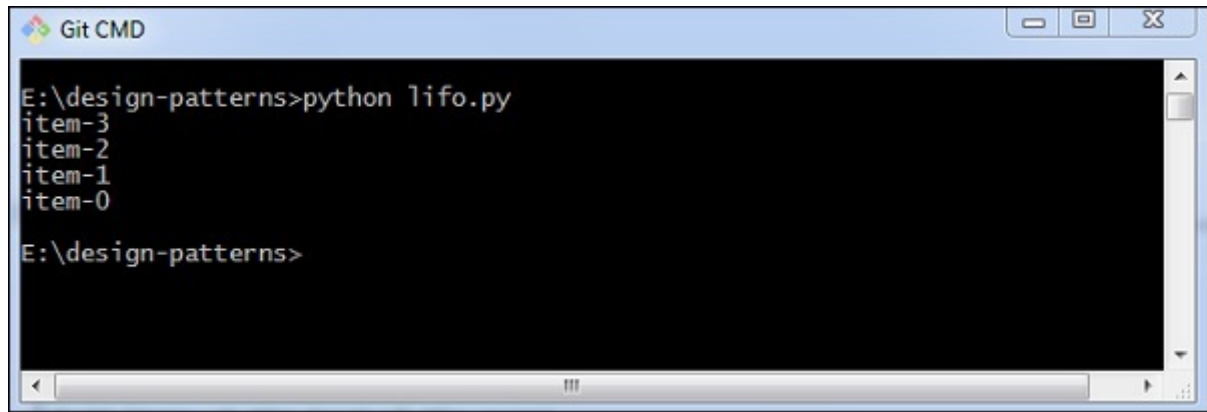
q = Queue.LifoQueue()

#add items at the head of the queue
for x in range(4):
    q.put("item-" + str(x))

#remove items from the head of the queue
while not q.empty():
    print q.get()
```

## Output

The above program generates the following output –



```
Git CMD
E:\design-patterns>python lifo.py
item-3
item-2
item-1
item-0
E:\design-patterns>
```

## What is a Priority Queue?

Priority queue is a container data structure that manages a set of records with the ordered keys to provide quick access to the record with smallest or largest key in specified data structure.

## How to implement a priority queue?

The implementation of priority queue is as follows –

```
import Queue

class Task(object):
    def __init__(self, priority, name):
        self.priority = priority
        self.name = name

    def __cmp__(self, other):
        return cmp(self.priority, other.priority)

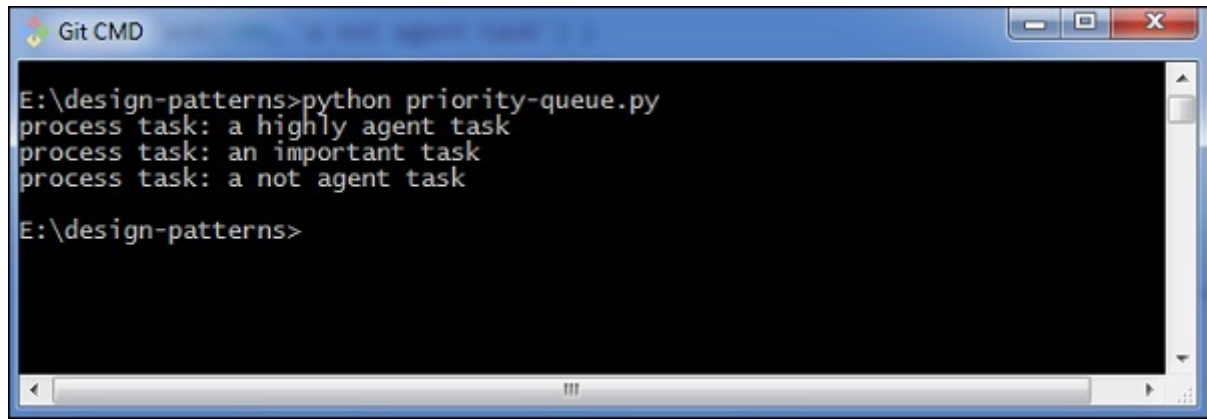
q = Queue.PriorityQueue()

q.put( Task(100, 'a not agent task') )
q.put( Task(5, 'a highly agent task') )
q.put( Task(10, 'an important task') )

while not q.empty():
    cur_task = q.get()
    print 'process task:', cur_task.name
```

## Output

The above program generates the following output –

A screenshot of a Git CMD terminal window. The window title is "Git CMD". The command prompt shows the directory "E:\design-patterns". The user has run the command "python priority-queue.py". The output of the script is displayed on three lines: "process task: a highly agent task", "process task: an important task", and "process task: a not agent task". The prompt "E:\design-patterns>" is shown again on the next line.

```
Git CMD
E:\design-patterns>python priority-queue.py
process task: a highly agent task
process task: an important task
process task: a not agent task
E:\design-patterns>
```

## Strings and Serialization

String serialization is the process of writing a state of object into a byte stream. In python, the “pickle” library is used for enabling serialization. This module includes a powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process of converting Python object hierarchy into byte stream and “unpickling” is the reverse procedure.

The demonstration of the pickle module is as follows –

```
import pickle

#Here's an example dict
grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }

#Use dumps to convert the object to a serialized string
serial_grades = pickle.dumps( grades )
print(serial_grades)

#Use loads to de-serialize an object
received_grades = pickle.loads( serial_grades )
print(received_grades)
```

## Output

The above program generates the following output –

```

C:\Windows\system32\cmd.exe

E:\design-patterns>python serialization.py
<dp0
s'Bob'
p1
172
s'Charles'
p2
187
s' Alice'
p3
189
s
<'Bob': 72, 'Charles': 87, 'Alice': 89>
E:\design-patterns>

```

## Concurrency in Python

Concurrency is often misunderstood as parallelism. Concurrency implies scheduling independent code to be executed in a systematic manner. This chapter focuses on the execution of concurrency for an operating system using Python.

The following program helps in the execution of concurrency for an operating system –

```

import os
import time
import threading
import multiprocessing

NUM_WORKERS = 4

def only_sleep():
    print("PID: %s, Process Name: %s, Thread Name: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name)
    )
    time.sleep(1)

def crunch_numbers():
    print("PID: %s, Process Name: %s, Thread Name: %s" % (
        os.getpid(),
        multiprocessing.current_process().name,
        threading.current_thread().name)
    )
    x = 0
    while x < 10000000:
        x += 1
    for _ in range(NUM_WORKERS):
        only_sleep()

```



```

end_time = time.time()
print("Serial time=", end_time - start_time)

# Run tasks using threads
start_time = time.time()
threads = [threading.Thread(target=only_sleep) for _ in range(NUM_WORKERS)]
[thread.start() for thread in threads]
[thread.join() for thread in threads]
end_time = time.time()

print("Threads time=", end_time - start_time)

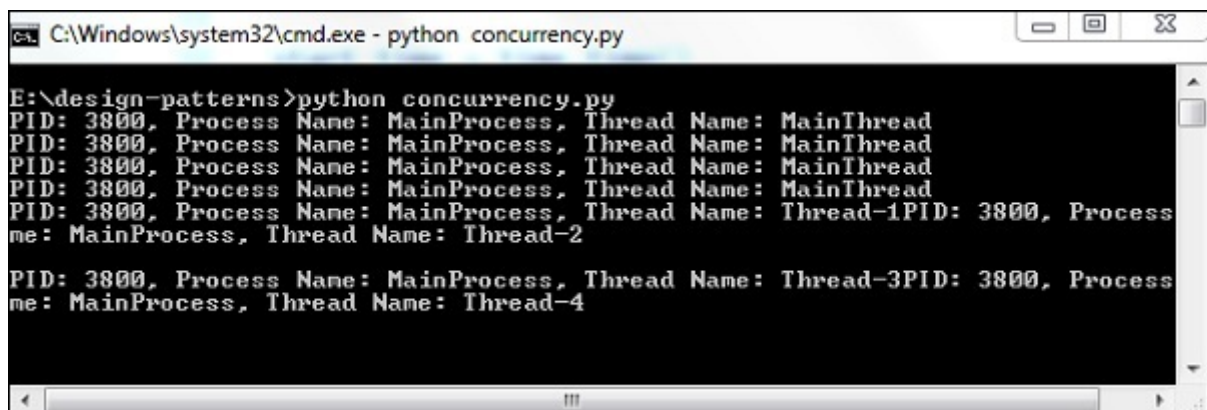
# Run tasks using processes
start_time = time.time()
processes = [multiprocessing.Process(target=only_sleep()) for _ in range(NUM_WORKERS)]
[process.start() for process in processes]
[process.join() for process in processes]
end_time = time.time()

print("Parallel time=", end_time - start_time)

```

## Output

The above program generates the following output –



```

C:\Windows\system32\cmd.exe - python concurrency.py
E:\design-patterns>python concurrency.py
PID: 3800, Process Name: MainProcess, Thread Name: MainThread
PID: 3800, Process Name: MainProcess, Thread Name: MainThread
PID: 3800, Process Name: MainProcess, Thread Name: MainThread
PID: 3800, Process Name: MainProcess, Thread Name: MainThread
PID: 3800, Process Name: MainProcess, Thread Name: Thread-1PID: 3800, Process
me: MainProcess, Thread Name: Thread-2
PID: 3800, Process Name: MainProcess, Thread Name: Thread-3PID: 3800, Process
me: MainProcess, Thread Name: Thread-4

```

## Explanation

“multiprocessing” is a package similar to the threading module. This package supports local and remote concurrency. Due to this module, programmers get the advantage to use multiple processes on the given system.

# Python Design Patterns - Anti

Anti-patterns follow a strategy in opposition to predefined design patterns. The strategy includes common approaches to common problems, which can be formalized and can be gener

considered as a good development practice. Usually, anti-patterns are opposite and undesirable. Anti-patterns are certain patterns used in software development, which are considered as bad programming practices.

## Important features of anti-patterns

Let us now see a few important features of anti-patterns.

### Correctness

These patterns literally break your code and make you do wrong things. Following is a simple illustration of this –

```
class Rectangle(object):
    def __init__(self, width, height):
        self._width = width
        self._height = height
    r = Rectangle(5, 6)
    # direct access of protected member
    print("Width: {}".format(r._width))
```

### Maintainability

A program is said to be maintainable if it is easy to understand and modify as per the requirement. Importing module can be considered as an example of maintainability.

```
import math
x = math.ceil(y)
# or
import multiprocessing as mp
pool = mp.pool(8)
```

### Example of anti-pattern

Following example helps in the demonstration of anti-patterns –

```
#Bad
def filter_for_foo(l):
    r = [e for e in l if e.find("foo") != -1]
    if not check_some_critical_condition(r):
        return None
    return r

res = filter_for_foo(["bar", "foo", "faz"])

if res is not None:
```

```

    #continue processing
    pass

#Good
def filter_for_foo(l):
    r = [e for e in l if e.find("foo") != -1]
    if not check_some_critical_condition(r):
        raise SomeException("critical condition unmet!")
    return r

try:
    res = filter_for_foo(["bar", "foo", "faz"])
    #continue processing

except SomeException:
    i = 0
while i < 10:
    do_something()
    #we forget to increment i

```

## Explanation

The example includes the demonstration of good and bad standards for creating a function in Python.

# Python Design Patterns - Exception Handling

Handling exceptions is also a primary criterion of design patterns. An exception is an error that happens during the execution of a program. When a particular error occurs, it is important to generate an exception. This helps in curbing program crashes.

## Why use exceptions?

Exceptions are convenient ways of handling errors and special conditions in a program. When a user thinks that the specified code can produce an error then it is important to use exception handling.

## Example – Division by zero

```

import sys

randomList = ['a', 0, 2]

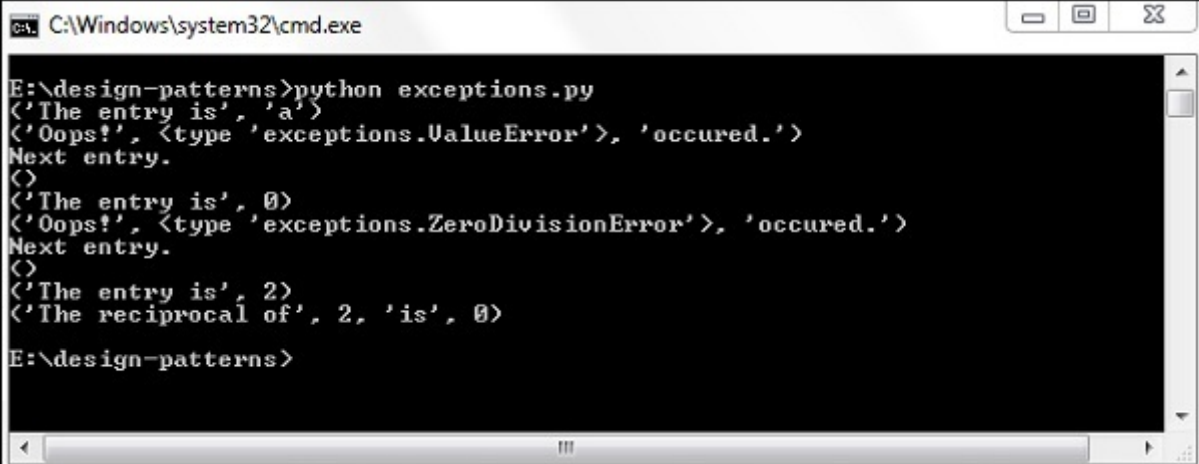
for entry in randomList:
    try:

```

```
    print("The entry is", entry)
    r = 1/int(entry)
    break
except:
    print("Oops!", sys.exc_info()[0], "occured.")
    print("Next entry.")
    print()
print("The reciprocal of", entry, "is", r)
```

## Output

The above program generates the following output –



```
C:\Windows\system32\cmd.exe
E:\design-patterns>python exceptions.py
<'The entry is', 'a'>
<'Oops!', <type 'exceptions.ValueError'>, 'occured.'>
Next entry.
<>
<'The entry is', 0>
<'Oops!', <type 'exceptions.ZeroDivisionError'>, 'occured.'>
Next entry.
<>
<'The entry is', 2>
<'The reciprocal of', 2, 'is', 0>
E:\design-patterns>
```

## Raising Exceptions

In Python programming specifically, exceptions are raised when corresponding error of code occurs at run time. This can be forcefully raised using the “**raise**” keyword.

## Syntax

```
raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt
```

---