Case Study Report
on
Python

By

Shubhankar Tripathi

U49332825

1. Inheritance

Inheritance allows us to create *is a* relationships between two or more classes, abstracting common logic into super-classes and managing specific details in the subclass[1]. Just like we learned in Java, that all the classes are the subclass of the special class named **object** which allows Python to treat all objects in the same way. Analogous to Java, Python also offers a subclass to be derived from its parent class or getting extended from its parents'.

In terms of syntax, the syntax is as simple and minimal as it is in Java. In Java, *extend* keyword is used to define a subclass inheriting from a superclass. In Python, it is simply required that the name of the parent class is included inside parentheses after the subclass name followed by colon terminating the class definition.

JAVA//

```
       //parent class definition syntax
       class Parent{..
          ..       }
       //subclass definition syntax
       class Child extends Parent{..
          ..       }
```

PYTHON//

```
       //parent class definition syntax
       class Parent:
          ..
       // subclass definition syntax
       class Child(Parent):
          ..
```

Method Overriding and super

Overriding means altering or replacing a method of the superclass with a new method (with the same name) in the subclass. The most obvious use of inheritance is to add functionality to an existing class. In the following example, *Employee* superclass is responsible for maintaining all the employees. The *Lawyer* subclass is adding an extra behavior of getting pink vacation form while complying and inheriting the Employee class.

JAVA//

```
      public class Lawyer extends Employee{
            public String getVacationForm() {
                   return “pink?”;
          }
          ...
      }
```

PYTHON// (override.py)

```
class Employee:
    def getVacationForm(self):
        return ("Sure!!")
class Lawyer(Employee):
    def getVacationForm(self):
        return ("pink?")
guy = Lawyer()
print (guy.getVacationForm())
```

Note: In Python, statement terminator is not used like it is done in Java. In Java, semi-colon used to be used as end of statement markers. The creators of Python though, used indentation as statement terminator.

Since there might be a case where a we need a way to execute the original superclass's method, we used *super* function for that. *super* function returns the object as an instance of the parent class, allowing to call the parent method directly. In the following example the *Lawyer* constructor, when called, is supposed to refer to the superclass's constructors. Further in *getSalary* method, it uses the superclass method by overriding it.

JAVA//

```
      public class Lawyer extends Employee {
           public Lawyer(String name) {
             super(name);
           }
           public double getSalary(){
             double baseSalary = super.getSalary();
             return baseSalary + 5000.00;
      }
```

Python// (super.py)

```
class Employee:
    def __init__(self,name):
        self.emp =name
    def getSalary(self):
        return 3000
class Lawyer(Employee):
    def __init__(self, name):
        super().__init__(name)
    def getSalary(self):
        baseSalary = super().getSalary()
        return baseSalary + 5000.00
guy = Lawyer("The New Guy")
print (guy.getSalary())
```

Method Overloading

[2]In Java, two or methods are overloaded if they have the same name but different number and types of parameters. In python, we can’t have two methods with the same name. Thus, Method overloading is accomplished by using optional and default parameters or by checking the type of the parameters and responding immediately. A single method with a single optional parameter is defined. When the method is called, the value of this parameter will be either zero i.e., no arguments given, or whatever type of parameter with its value.

JAVA//

```
    public class Display{
          public void disp(char c){
                System.out.println(c);
          }
          public void disp(char c, int sum){
                System.out.println(c + “ ”+num);
          }
    }
    class App{
          public static void main(String args[]){
                Display dsp = new Display();
                dsp.disp(‘s’);
                dsp.disp(‘s’,12)
         }
    }
```

Python// (overload.py)

```
class Display:
    def _init_(self):
        pass
    def disp(self,char, num = None):
        if num != None:
            print(char,num)
        else:
            print(char)
def main():
    dsp = Display()
    dsp.disp('s')
    dsp.disp('s',12)
if __name__=="__main__":
    main()
```

Polymorphism

It is the ability of different types of objects too feature their own version of a method. It is a concept describing different behaviors happen depending on which subclass is being used, without having to explicitly know what the subclass actually is. In the following example, a program opens various document files. A file processor will load an *DocFile* object and then *open* it. We will put a *open()* method on the object, which is responsible for opening and extracting the document and displaying it over the screen. Each type of file can be represented by a different subclass of *DocFile*, for example, *WordFile*, *PdfFile*. Each of these would have a *open()* method, but that method would be implemented differently for each file to ensure the correct procedure is followed. The file processor object would never need to know which subclass of *DocFile* it is referring to; it just calls *open()* and polymorphically lets the object take care of the actual details of playing.

JAVA//

```
class DocFile {
     public void open() {
          System.out.println("opening the document.");
     }
}

class WordFile extends DocFile {
     public void open() {
          System.out.println("opening the word file. ");
     }
}

class PdfFile extends DocFile {
     public void open() {
          System.out.println("opening the pdf file");
     }
}

class RtfFile extends DocFile {
     public void open() {
          System.out.printf("opening rtf file");
     }
}

public class App {
     public static void main(String[] args) {
          DocFile wf = new WordFile();
          wf.open();
          DocFile pdf = new PdfFile();
          pdf.open();
          DocFile rtf = new RtfFile();
          rtf.open();
     }
}
```

Python// (polymorphism.py)

```
class DocFile:
    def open(self):
        print ("opening the document.")
class WordFile(DocFile):
    def open(self):
        print ("opening the word file.")
class PdfFile(DocFile):
    def open(self):
        print ("opening the pdf file.")
class RtfFile(DocFile):
    def open(self):
        print ("opening the rtf file.")
class XpsFile:
    def open(self):
        print ("opening the xps file.")
word = WordFile()
word.open()
pdf = PdfFile()
pdf.open()
rtf = RtfFile()
rtf.open()
xps = XpsFile()
xps.open()
```

Python makes polymorphism less cool because of duck typing. Duck typing in Python allows us to use any object that provides the required behavior without forcing it to be a subclass. The dynamic nature of Python makes this trivial. In the last example, the class XpsFile doesn't really need to extend DocFile in order to inherit open method and thus print using over-rided method polymorphically.

Polymorphism is one of the most important reasons to use inheritance in many object-oriented contexts like in Java. Because any objects that supply the correct interface can be used interchangeably in Python, it reduces the need for polymorphic common superclasses. Inheritance can still be useful for sharing code but, if all that is being shared is the public interface, duck typing is all that is required. This reduced need for inheritance also reduces the need for multiple inheritance; often, when multiple inheritance appears to be a valid solution, one can just use duck typing to mimic one of the multiple superclasses.

Abstract Base Classes

While duck typing is useful, it is not always easy to tell in advance if a class is going to fulfill the protocol you require. Therefore, Python has the concept of abstract base classes. Abstract base classes, define a set of methods and properties that a class must implement in order to be considered a duck-type instance of that class. The class can extend the abstract base class itself in order to be used as an instance of that class, but it must supply all the appropriate methods. Python doesn't have (and doesn't need) a formal Interface contract, the Java-style distinction between abstraction and interface doesn't exist. If someone goes through the effort to define a formal interface, it will also be an abstract class. The only differences would be in the stated intent in the docstring[3].

Most of the abstract base classes that exist in the Python Standard Library live in the collections module. One of the simplest ones is the Container class. The Container class has exactly one abstract method that needs to be implemented, __contains__.

Python// (oddcontainer.py)

```
class OddContainer:
    def __contains__(self, x):
        if not isinstance(x, int) or not x % 2:
            return False
        return True
from collections import Container
odd_container = OddContainer()
print (isinstance(odd_container, Container))
print (issubclass(OddContainer,Container))
```

Running the above script establishes why duck typing is way more useful in Python than classical polymorphism. In the example, the class OddContainer hasn't actually extended anything. But, the *is-a* relationship can be created without overhead of using inheritance(or multiple inheritance).

Also, the class implements *container* abstract base class can use the *in* keyword. So, any class that has a __contains__ method is a *Container* and can therefore be queried by *in* keyword.

Creating an abstract base class

In Java, An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.[4] An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon).

2. Meta-Programming

Reflection in Python[5]

Reflection is an API in Java which is used to examine or modify the behavior of methods, classes, interfaces at runtime. In Python, Reflection refers to the ability for code to be able to examine attributes about objects that might be passed as parameters to a function. For example, if we write type(obj) then Python will return an object which represents the type of obj. Reflection in Java can be used to get information about classes, constructors and methods using *getClass(), getConstructors()* and *getMethods()* respectively. In Python, we will implement similar Reflection-enabling functions like *type(), isinstance(), callable(), dir()* and *getattr()* etc. The type() function will serve the purpose of getClass(), getMethods()

Type and isinstance:

Python have a built-in method called as type which generally come in handy while figuring out the type of variable used in the program in the runtime. If a single argument (object) is passed to type() built-in, it returns type of the given object. If three arguments (name, bases and dict) are passed, it returns a new type object.

//Python : type() with a single object parameter : (typesingle.py)

```
num = 12
group = ["Post Malone", "Weeknd", "Zayn"]
future = "Tesla"
print(type(num))
print(type(group))
print(type(future))
```

//Python : type() with a name, bases and dict parameter : (typeanother.py)

```
class car(object):
    model = 'Roadster'
    year = 2019
obj =type('car',(object,),dict(model='roadster',make=2019))
print(type(obj))
print(vars(obj))
```

The isinstance() function checks and returns true, if the object (first argument) is an instance or subclass of classinfo class (second argument). Otherwise it returns false

```
//Python : (isinstance.py)
class Car:
    make: "Peugeot"
class Random:
    rnum = 12
obj = Car()
print(isinstance(obj,Car))
print(isinstance(obj,Random))
```

Callable()

A callable means anything that can be called. For an object, determines whether it can be called. A class can be made callable by providing a __call__() method. The callable() method returns True if the object passed appears callable. If not, it returns False.

```
//Python (callable.py)
name = "Tesla"
def func():
     print("Test")
obj = func
if (callable(name)):
     print("name is callable")
else:
     print("name is not callable")
if (callable(obj)):
     print("obj is callable")
else:
     print("obj is not callable")
```

Dir

The dir() method tries to return a list of valid attributes of the object. The dir() tries to return a li If the object has __dir__() method, the method will be called and must return the list of attributes. If the object has __dir__() method, the method will be called and must return the list of attributes. If the object doesn’t have __dir()__ method, this method tries to find information from the __dict__ attribute (if defined), and from type object. In this case, the list returned from dir() may not be complete.

```
//Python (dir.py)
name = ["spaceX","boeing","blue Origin"]
print(dir(name))

characters = ["a", "b"]
print(dir(name))
```

Getattr

The getattr() method returns the value of the named attribute of an object. If not found, it returns the default value provided to the function.The getattr method takes three parameters object, name and default(optional).

//Python (getattr.py)

```
class Car:
    year = 2018
    make = "Tesla"
car = Car()
print('The car is manufactured by:', getattr(car, "make"))
print('The car is manufactured by:', car.make)
```

Now we'll try implementing all the illustrations from the lecture in Python, related to Reflection.

Retrieving Class Objects

JAVA//

```
class Dummy{
}

public class App {
     static int primnum = 2
     static Integer num = 12;
     public static void main(String[] args) {
          Dummy foo = new Dummy();
          Class c = foo.getClass();
          Class d = System.out.getClass();
          Class e = "word".getClass();
          Class f = num.getClass();
          Class g = primnum.class;
          System.out.println(c);
          System.out.println(d);
          System.out.println(e);
          System.out.println(f);
          System.out.println(g);
     }
}
```

Python// (getType.py)

```
class Dummy:
    pass
class App:
    def run(self):
        num = 12
        foo = Dummy()
        print (type(foo))
        print (type(num))
        print (type("word"))
obj = App()
obj.run()
```

In the above example, use of getClass() method in Java has been substituted by type() function in Python. In Java, in order to dereference the primitive type, if we use getClass() method it will throw compile-time error. So .class is used to overcome that. There is no such problem like that in Python, since everything is an object, be it a data type or a Class or a function(method).

Retrieving with Class.forName(..) and it's analogous implementation in Python[6]

There's no direct function which takes a fully qualified class name and returns the class, however we can have all the pieces needed to build that, and we can connect them together.

Java//

```
class App{
     public static void main(String[] args){
          Class c = Class.forName("whatever.App");
     System.out.println(c);
     }
}
```

Python// (getClass.py)

```
class Dummy:
    pass
def import_class(name):
    components = name.split('.')
    mod = __import__(components[0])
    for comp in components[1:]:
        mod = getattr(mod, comp)
    return mod
print (my_import('__main__.Dummy'))
```

Retrieving with TYPE field in Java and its analogous Python implementation

In Java, each of the primitive types and void has a wrapper class and each of the wrapper class contains a field TYPE which is equal to the Class for the primitive type being wrapped. Python takes care of it, by implementing type() function just like we already saw in sections before.

Methods that return Class Objects in Java and its analogous Python implementation

In Java, there are several Reflection APIs which returnClass objects. These may only be used if Class object has already been obtained either directly or indirectly. getSuperClass() is one such method that returns super class for given class. Since there are no direct functions like there was in Java, we will try a workaround here. Class objects have a __name__ attribute. It might be cleaner to introspect the base class(es) or super classes through the __bases__ attr of the derived class(subclass)

Java//

```
class SuperApp{
}
public class App extends SuperApp {

     public static void main(String[] args) throws
ClassNotFoundException {
          App obj = new App();
          Class c = obj.getClass().getSuperclass();
          System.out.println(c);
     }
}
```

Python// (getSuperClass.py)

```
class SuperApp:
    pass
class App(SuperApp):
    def get_super(self):
        for base in self.__class__.__bases__:
            print (base.__name__)
obj = App()
obj.get_super()
```

Methods that return Member Classes' Objects in Java and their analogous Python implementation

In Java, getClasses() returns all the **public** classes, interfaces, and enums that are members of the class, inner or nested classes, including inherited members.

Java//

```
public class App {

     public static void main(String[] args) throws
ClassNotFoundException {
          App obj = new App();
          Class[] c = obj.getClass().getClasses();
          System.out.println(c.length);
     }

     public class InnerApp1{

     }
     private class InnerApp2{

     }
}
```

Python// (memClass.py)

```
class App:
    class InnerApp1:
        pass
    class __InnerApp2__:
        pass
obj = App()
print (dir(obj))
```

Running above script would result in listing of every single attributes defined and present in the App class. Because of that its implementation is more analogous to getDeclaredClass() in Java. But, we can filter the special methods and attributes based on their name.

Python// (getDeclaredClass.py)

```
class App:
    class InnerApp1:
        pass
    class __InnerApp2__:
        pass
obj = App()
attributes = dir(obj)
print ([a for a in attributes if not(a.startswith('__') and
a.endswith('__'))] )
```

Method that returns Declaring Classes' Objects in Java and its analogous implementation in Python

The getDeclaringClass() method in Java is used to get the class in which members were declared.

```
Java//
public class App {

     public static void main(String[] args) throws
ClassNotFoundException {
          App obj = new App();
          App.InnerApp1 obj1 = obj.new InnerApp1();
          Class c = obj1.getClass().getDeclaringClass();
          System.out.println(c);
     }

     class InnerApp1{

     }
}
//Python (getDeclaringClass.py)
class App:
    class InnerApp1:
        pass
obj = App()
obj1 = obj.InnerApp1()
print (obj1.__class__)
```

Decorators

A decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it. It changes the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated. We can implement the decorator pattern anywhere, but Python facilitates the implementation by providing much more expressive features and syntax for that.

Essentially, decorators work as wrappers, modifying the behavior of the code before and after a target function execution, without the need to modify the function itself, augmenting the original functionality, thus decorating it. Decorator uses the @ symbol to get acknowledged. In the following example (undecorated.py), *outer* is called with *func* as an argument and it returns a new functions *inner.* Further in *decorated.py*, *@outer* notation is simply a special syntax to call an existing function, passing the new function as an argument, and use the return value to replace the new function. It makes it more concise.

```
Python// (undecorated.py)
def outer(func):
    def inner(name):
        return "Hey " + func(name)
    return inner
def f(name):
    return name + "!!!"
f = outer(f)
```

```
print (f("Bernie"))
```

Python// (decorated.py)

```
def outer(func):
    def inner(name):
        return "Hey " + func(name)
    return inner
@outer
def f(name):
    return name + "!!!"
print (f("Bernie"))
```

3. Generics

Generic programming is a concept in OOP where codes are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters[9]. A generic program is one that the programmer writes once, but which works over many different data types. Broadly speaking, generic programming aims at relieving the programmer from repeatedly writing functions of similar functionality for different user-defined data types. A generic function such as a pretty printer or a parser is written once and for all times; its specialization to different instances of data types happens without further effort from the user. This way generic programming greatly simplifies the construction and maintenance of software systems as it automatically adapts functions to changes in the representation of data[10].

Unlike Java, Python(before v3) is not a statically typed language and thus actually addresses generic programming the way Java does. Python doesn't have any compile time type safety, so it wouldn't make sense to add generics to beef up the compile time type checks Python doesn't have. A list, for example, is an untyped collection. There is no equivalent of distinguishing between List<Integer> and List<String> because a Python list can store any type of object.

Python uses duck typing, so it doesn't need special syntax to handle multiple types. Type is a property of objects, not variables, and hence *necessarily* when you come to call a method on an object, or otherwise use properties that it has by virtue of its type, the presence or absence of that method is discovered at runtime[*]. So if it "looks like a duck and quacks like a duck" (i.e. if it turns out to have a quack() function) then it "is" a duck (anyway, you can treat it like one).

Duck typing is a concept that says that the "type" of the object is a matter of concern only at runtime and you don't need to to explicitly mention the type of the object before you perform any kind of operation on that object[11]. Consider the following example:

Python// (dktypn.py)

```
def func(p1,p2):
    print(p1+p2)
greet = func('Hey!!', 'Sam')
add = func(7,5)
```

In the above example, Python doesn't really care what's the type of parameters p1 and p2. Their type usually are taken care of at runtime up until the two objects can be calculated using '+' operators.

In Python v3.5, a new library called typing got added providing a feature which gives the hint of the type and doesn't bind it, so that there isn't anything being passed which shouldn't be passed.

Python// (hint.py)

```
def func(name:str, rank: int) -> None:
    print("%s holds %s rank in constructors' ranking." % (name,
rank))
func('Mercedes',1)
func('Williams',5)
```

In the above example, the function defined func is going to be expecting two arguments: one of the type str and another of type int. Also this function hints at returning nothing by using *None.* According to PEP 484, “Type hints may be built-in classes (including those defined in standard library or third-party extension modules), abstract base classes, types available in the types module, and user-defined classes”. So that means we can create our own class and add a hint.

4.Collections[7]

Python offers a module that contains several container data types called Collections. We will implement them and testing their usefulness. This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers*, dict, list, set* and *tuple*.

First off all, let’s see how do general-purpose built in container types work. Following are the types we will be describing:

- Lists
- Tuples
- Dictionaries
- Sets

Lists

A list represents an ordered, mutable collection of objects. We can have object of any type added and removed in a list. To create an empty list, empty square brackets or the list() function with no arguments are used. We can initialize a list with content of any sort using the same square bracket notation. The list() function also takes an iterable as a single argument and returns a shallow copy of that iterable as a new list. Following are the functions used to manipulate a list.

- len: len function is used to check the length of a list
- append(): This function is used add more objects to the list.
- remove(): This function is used to delete elements of the list. It doesn’t have a return value.
- pop(): This function also used to remove the item from the list but it returns the item as well.
- sort(): is used to sort the list using it and it doesn’t return a value.
- reverse(): is used to reverse the elements of the list.

Python// (list.py)

```
new_list = ["Acura","Lincoln","Cadillac","BMW","Peugeot", 12]
print ("Length of the new_list is", len(new_list))
# following command prints "None" because append doesn't return
any value
```

```
print (new_list.append("Verstappan"))
print (new_list)
print (new_list.remove("Cadillac"))
print (new_list)
new_list.append("Cadillac")
print ("Popped element:", new_list.pop(4))
new_list.sort()
print ("Sorted list:", new_list)
print ("Reversed list:", new_list)
```

Tuples

A tuple is similar to a list. In Tuple, elements cannot be changed once assigned while in list that can be done. Tuple is usually used for heterogenous datatypes and list for homogeneous datatypes. Iterating with tuple is faster than it is with list, since it is immutable. A tuple is created by placing all the elements inside a parentheses (), separated by comma. The parentheses are optional. Following are the various ways to manipulate it.

- Indexing: can be used to access an item in a tuple. Index starts from 0.
- Negative Indexing: can also be used as Python allows it. The index of -1 refers to the last element, -2 refers to the second last element.
- Slicing: can be used to access a range of items in a tuple by using slicing operator, i.e., colon.
- Concatenation: The + operator is used to combine two tuples. The elements can be repeated in a tuple for a given number of times using * operator.

`Python// (tuple.py)`

```
new_tuple = ('Acura','Lincoln','Cadillac','BMW','Peugeot', 12)
print ("Length of the new_tuple is", len(new_tuple))
print (new_tuple[3])
print (new_tuple[-2])
print (new_tuple[2:5])
print (new_tuple[:3])
```

Dictionaries

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair. Dictionaries are optimized to retrieve values when the key is known. To create a dictionary, elements are placed inside curly braces separated by comma. Value can be of any data type and be duplicate, but key must be unique and must be of immutable type. Following are the functions that can be used to manipulate dictionaries.

- get(): is used to get the values for the keys. Alternate method is to put the key inside the square bracket.
- Adding an element: Since Dictionaries are mutable, it's possible to add or update elements to the dictionary using assignment operator.
- popitem(): is used to remove and return an arbitrary element (key, value) from the dictionary.
- clear(): can be used to remove all the elements at once.
- del: is the keyword that could be used to remove individual items or the entire dictionary.
- sorted(): can be used to sort the dictionary according to the key.

Python// (dict.py))

```
new_dict = {1: 'Acura', 2: 'Lincoln', 3: 'Cadillac', 4: 'BMW',
5: 'Peugeot', 6: 12}
print ("Length of the new_dict is", len(new_dict))
new_dict[7] = 'Subaru'
print (new_dict)
print ("3rd element:",new_dict.pop(3))
print ("arbitrary element: ", new_dict.popitem())
print (new_dict)
new_dict.clear()
print (new_dict)
```

Sets

Sets are unordered collections that can't have duplicate elements. Sets can't be sorted. The sort()method isn't available for sets. In comparison to lists, sets can check for the existence of an element faster than lists. Set doesn't support indexing To create a set, elements are comma-separated and put in the curly braces. The set() function can also be to create one by supplying an existing structure. Following are the functions used to manipulate a set.

- add(): is used to add elements to the set. Duplicate elements can't be added to the set.
- remove(): is used to remove the element from a set.

- difference(): is used to return a new set with elements of the set the function is applied to, but not thee elements of the set which is passed as argument.
- intersection(): is used to find out what is common between two sets.
- union(): is used to combine to sets to create a new set.

Python// (set.py)

```
new_set = {"Acura","Lincoln","Cadillac","BMW","Peugeot", 12}
print ("Length of the new_set is", len(new_set))
new_set.add("Subaru")
print (new_set)
new_set.add("Acura") #Adding a duplicate element
print (new_set.remove("Cadillac"))
print (new_set)
another_set = {"Mercedes", "Porche", "Jaguar", "Acura",
"Peugeot"}
print (new_set.difference(another_set))
print (another_set.difference(new_set))
print (new_set.intersection(another_set))
print (new_set.union(another_set))
```

Following are high performance container data types[8]:

- defaultdict
- OrderedDict
- counter
- deque
- namedtuple
- enum.Enum

defaultdict

Using *defaultdict* is a bit easier than using dict. Unlike *dict*, with *defaultdict* we don't need to check whether a key is present or not. So we can do:

Python// (defdict.py)

```
from collections import defaultdict

cars = (
    ('Tesla', 'Model S'),
    ('BMW', '7 Series'),
    ('Daimler', 'S Class'),
    ('VW', 'Passat'),
    ('Toyota', '86'),
    ('Honda', 'NSX'),
)

preferred_car = defaultdict(list)

for name, model in cars:
    preferred_car[name].append(model)

print(preferred_car)
```

Another important use case is when it is needed to append to nested lists inside a dictionary. If a *key* is not already present in the dictionary then it will throw a *KeyError*. *defaultdict* allows to circumvent this issue in a clever way.

OrderedDict

As the name suggests, OrderedDict keeps its entries sorted as just the way they are initially inserted. Overwriting a value of an existing key doesn’t change the position of that key. However, deleting and reinserting an entry moves the key to the end of the dictionary.

Python// w/o implementing OrderedDict (unorddict.py)

```
cars =  {"Tesla":"Model X", "BMW":"7-Series", "Merc":"S-Class"}
for key, value in cars.items():
    print(key, value)
```

//implementing OrderedDict (orddict.py)

```
from collections import OrderedDict
cars =  {"Tesla":"Model X", "BMW":"7-Series", "Merc":"S-Class"}
for key, value in cars.items():
    print(key, value)
```

Now, running the two different above displayed scripts would show how the first one retrieves thus prints in random order but the one using OrderedDict fixes that problem by retrieving the list in the order it was stored.

counter

Counter allows to count the occurrences of a particular item. For instance, it can be used to count the number of individual preferred_cars:

```
Python// (counter.py)
from collections import Counter
cars = (
    ('Tesla', 'Model S'),
    ('BMW', '7 Series'),
    ('Daimler', 'S Class'),
    ('Tesla', 'Model X'),
    ('BMW', '5 Series'),
    ('Honda', 'NSX'),
)

preferred_cars_count = Counter(name for name, model in cars)
print(preferred_cars_count)
```

deque

deque, just like in Java a double ended queue which means that one can append and delete elements from either side of the queue. Primarily, we will have to import the deque module from the collections library. And then we may instantiate deque object. It works like lists providing some of similar functions as well.

```
Python// (deq.py)

from collections import deque
dq = deque()
dq.append('spaceX')
dq.append('boeing')
dq.append('blue origin')
dq.append('ULA')
print (dq)
print (len(dq))
print (dq[2])
print (dq[-2])
print (dq.popleft())
print (dq.pop())
print (dq)
```

One can also limit the amount of items a deque can hold by setting maxlen attribute. By doing this when the maximum limit of our deque is achieved, it will simply pop out the items from the opposite end.

namedtuple

A tuple is an immutable list that allows to store a sequence of values separated by commas. They are just like lists but have a few key differences. The major one is that unlike lists, reassignment of an item in a tuple cannot be done. In order to access the value in a tuple integer indexes are used.

namedtuple turn tuples into convenient containers for simple tasks. With namedtuples we don’t have to use integer indexes for accessing members of a tuple. namedtuples act like dictionaries but unlike dictionaries they are immutable.

Python// (namedtuple.py)

```
from collections import namedtuple
cars = namedtuple('cars', 'make model')
car1 = cars(make = "Tesla", model = "Model 3")

print(car1)
print(car1.make)
```

A namedtuple has two required arguments. They are the tuple name and the tuple field_names. In the above example our tuple name was ‘cars’ and the tuple field_names were 'make' and 'model' of the cars. namedtuple makes tuples self-document. And as there is no requirement to use integer indexes to access members of a tuple, it makes it more subtle to maintain code. As namedtuple instances don't have per-instance dictionaries, they are lightweight and require no more memory than regular tuples which makes them faster than dictionaries. However, just like tuples attributes in namedtuples are immutable namedtuples are used to make the code self-documenting. Since they are backwards compatible with normal tuples, integer indexes can be used with namedtuples as well.

enum.Enum

There is another newly introduced useful collection is the enum object. It is available in the enum module. Enums (enumerated type) are basically a way to organize various collections of objects. Considering the cars namedtuple from the last example, it had a make field. The problem is, the make was a string. This poses some problems.Such as, What if the user types in tesla? Or TESLA? Enumerations can help us avoid this problem, by not using strings. Consider this example

Python// (enum.py)

```
from collections import namedtuple
from enum import Enum

class Vehicles(Enum):
    Tesla= 1
    Bmw= 2
    Daimler = 3
cars = namedtuple('cars', 'make model')
car1 = cars(make=Vehicles.Tesla, model="ModelX")
car2 = cars(make=Vehicles.Bmw, model="M5")
car3 = cars(make=Vehicles.Daimler, model="E63AMG")
car4 = cars(make=Vehicles.Bmw, model="6Series")
print (car2.make==car4.make)
print (car2.make)
```

This is much less error-prone. It requires to be specific, and enumerations should only be used for name types.There are ways to access enumeration members, e.g., all three methods will get you the value for Bmw:

```
Vehicles(2)
Vehicles['Bmw']
Vehicles.Bmw
```

5. Functional Programming

Functional programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input. The functional paradigm has its basis on the declarative aspect, which is ruled by two main constraints: the use of expressions and the lack of data mutation. The first one means that the code should tell what it aims to do instead of how to do it, and the second means that once a variable assumes a value it will never change on the course of the program execution.

Some of the characteristics of Functional Programming are:

- Avoid state representation
- Data are immutable
- First class functions
- High-order functions
- Recursion

State representation and Transformation

In functional programming, functions transform input into output, without an intermediate representation of the current state.

Python // not_func_cube.py

```
def cube(i):
    return i*i*i

input = [1, 2, 3, 4]
output = []
for k in input:
    output.append(cube(k))
print (output)
```

In this example, instead of iterating over the list we can use map() function to apply the cube function on all of the input elements. The output is also a little bit different, as the map()function will return an iterator rather than a list.

Python // func_cube.py

```
def cube(i):
    return i*i*i

input = [1, 2, 3, 4]
output = map(cube,input)
print (list(output))
```

Data are immutable

The concept of immutable data can be summarized as: functions will return new data as the outcome of a computation, leaving the original input intact. There are both mutable and immutable data structures like already been discussed in Collections, e.g. lists are mutable arrays, while tuples are immutable.

First class functions and High-order functions

A first class function is the one that appear anywhere in a program, including return values and arguments of other functions.

Python// firstClass.py

```
def firstClass(line):
    print ("Hey!", line)
firstClass("I'm first class.")
```

High-order functions are functions which can take functions as arguments (or return them as results). For example, this is what the map() function does, as one of the arguments is the name of the function to apply to a sequence. The use of map() function in func_cube.py example is one of the way first class functions are implemented.

reduce() and filter()

The functions reduce() and filter() naturally belong together with the map() function. While map will apply a function over a sequence, producing a sequence as output, reduce will apply a function of two arguments cumulatively to the items in a sequence, in order to reduce the sequence to a single value. Python has a built-in function in the functools standard library. The use of filter() consists in applying a function to each item in a sequence, building a new sequence for those items that return true. filter() returns an object of type generator, hence use of list function is required to visualize the actual list. Similar is situation with reduce(), since it also requires use of list().

Python// (reduce.py)

```
from functools import reduce
print(reduce((lambda x,y: y*x), [1,2,3,4]))
```

Python// (filter.py)

```
def startsWithA(name):
    return name[0]=='A'
def startsWithS(name):
    return name[0]=='S'
input = ["Acura", "AMG", "Subaru", "Suzuki"]
print (list(filter(startsWithA,input)))
print (list(filter(startsWithS, input)))
```

lambda functions

Python supports lambda functions just as Java does, where a small anonymous function is not bound to a name at runtime.

Python// lambda.py

```
input = ["Acura", "AMG", "Subaru", "Alfa Romeo"]
ranking = map(lambda i: len(i), input)
print (list(ranking))
```

Reference

[1] Chapter 3, When object are alike, Python OOP.

[2] http://home.wlu.edu/~lambertk/pythontojava/Overloading.htm'

[4] https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

[5] http://www.geeksforgeeks.org/reflection-in-python/

[6] https://stackoverflow.com/questions/452969/does-python-have-an-equivalent-to-java-class-forname/452981

[7]Chapter 6,Python Data Structures, Python OOP.

[8] https://docs.python.org/2/library/collections.html

[9] https://en.wikipedia.org/wiki/Generic_programming

[10] Generic Programs and Proofs, by Ralf Hinze.

[11] https://www.quora.com/What-is-Duck-typing-in-Python

[12] https://www.blog.pythonlibrary.org/2016/01/19/python-3-an-intro-to-type-hinting/