# Data Structures & Algorithms

Week 6

Inheritance and Abstract classes

Lecturer: Dr. Msagha J Mbogholi, PhD

# Flashback from Lesson 5

- There are four pillars of object orientation: abstraction, encapsulation, modularity, and hierarchy.
- Object oriented languages such as Java have the *interface* keyword which allows for the creation of interfaces; Python, however, does not.
- An abstract class and interface appear similar in Python. The only difference in two is that the abstract class may have some non-abstract methods, while all methods in interface must be abstract, and the implementing class must override all the abstract methods.
- A bag belongs to a group of collections known as unordered collections
- A constructor is a special method that is used exclusively to instantiate a class; that is to say, it is used to create the objects of that class.
- The steps to take when designing the interface: List each of the method headers with its documentation: complete each method with a single pass or return statement; pass statement is used in the mutator methods that return no value; each accessor method returns a simple default value, such as False, 0, or None.

# Content

- Inheritance
- Abstract classes

# Part 1

Inheritance

# 1.1 Introduction

- In lesson 5 hierarchy was listed as one of the pillars of object orientation.
- It was mentioned that classes are viewed in a hierarchy of parents and children. The parents are higher up in the hierarchy while the children occupy lower positions in the hierarchy.
- In this lesson we examine hierarchy in a more detailed fashion. The reason for this is that we have seen in lesson 5 how to implement the methods of an interface in another interface; we saw how the ArrayBag interface implemented the Bag interface. Well, this was just scrapping the tip of the iceberg; turns out that we can do more than just get one interface to implement another.
- Before we can do more with the Bag interface it is imperative that we understand what inheritance means from the Python perspective.

# 1.1 Introduction

- Different programming languages implement inheritance differently; it is not in the scope of this course to examine each language and how it implements inheritance.
- As we shall see in the course of this lesson there is also a close relationship between inheritance and abstract classes. Thus, whereas this course is not really about programming *per se* we can't avoid explaining some concepts that are crucial for the learner to understand in order to grasp the applications thereof in code.
- In this first part we wish to define some concepts, and then go on to show how they are used in Python. If you are already very familiar with Python inheritance, and how it is implemented in code you can skip to part 2 of this lesson.
- But wait, let us go through this first part together; even if it is a refresher to you.

# 1.2 Definitions

- Let us first define a few terms and the go on to explain them in detail in the next few slides:
- Inheritance: the concept of one class using the methods of another class. This class shares some characteristic(s) of the class whose methods it uses. Thus this class, say A, is a type of , say B.
- Overriding: the ability of a class to inherit the methods of its parent class, and customize them to suit its needs.
- Polymorphism: is to take many forms. In Python this means the same action being performed in different ways. In polymorphism, a method can process objects differently depending on the class type or data type. I guess you can see that overriding is a form of polymorphism already.
- We shall see application of these terms as we proceed with the lesson.

# 1.3 Inheritance

- Python provides support for four types of inheritance:
  - Single inheritance
  - Multiple inheritance
  - Multilevel inheritance
  - Hierarchical inheritance
  - Hybrid inheritance
- In describing these different forms of inheritance we use the term base class to mean the parent class; also the derived class means the child class.

# 1.3.1 Single inheritance

- In single inheritance a derived class inherits properties from a single parent class; this enables code reusability and the addition of new features to existing code. The syntax for the child class is: class <childname>(<parentname>). It is captured in Figure 1.
- Let us demonstrate this by creating a parent class then a derived class:
- #create the parent class
- class Person:
  def __init__(self, fname, lname):
  self.firstname = fname
  self.lastname = lname

  def printname(self):
  print(self.firstname, self.lastname)

# 1.3.1 Single inheritance

- # Create a child (derived) class
- “””To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class”””
- class Student(Person):
  pass
- #student inherits all the properties of person class without any additions.
- Let us consider one more example:

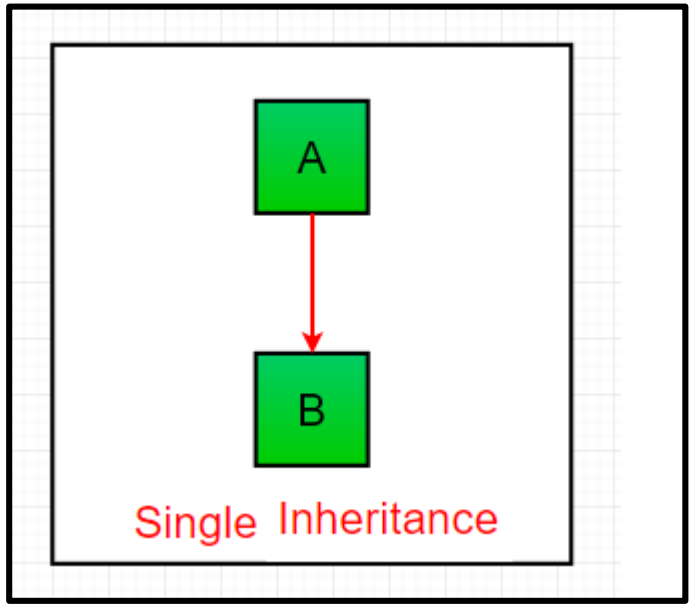
```
# Base class
class Parent:
   def myparentfunc(self):
      print("This function is in the parent class.")
```

# 1.3.1 Single inheritance

```
# Derived class
 class Child(Parent):
   def mychildfunc(self):
      print("This function is in child class.")
#running each method separately
Test = child()
Test.myparentfunc()
Test.mychildfunc()
```

**Figure 1**

*Single inheritance*




(From *Types of Inheritance Python*, 2020)

# 1.3.2 Multiple inheritance

- When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class. This is seen in Figure 2
- The syntax to use is:
- Class Parent1:
  - <some methods>
- Class Parent2:
  - <some methods>
- …..Class ParentN:
  - <some methods>
- Class Child (Parent1, Parent2,…ParentN)
  - <some methods>

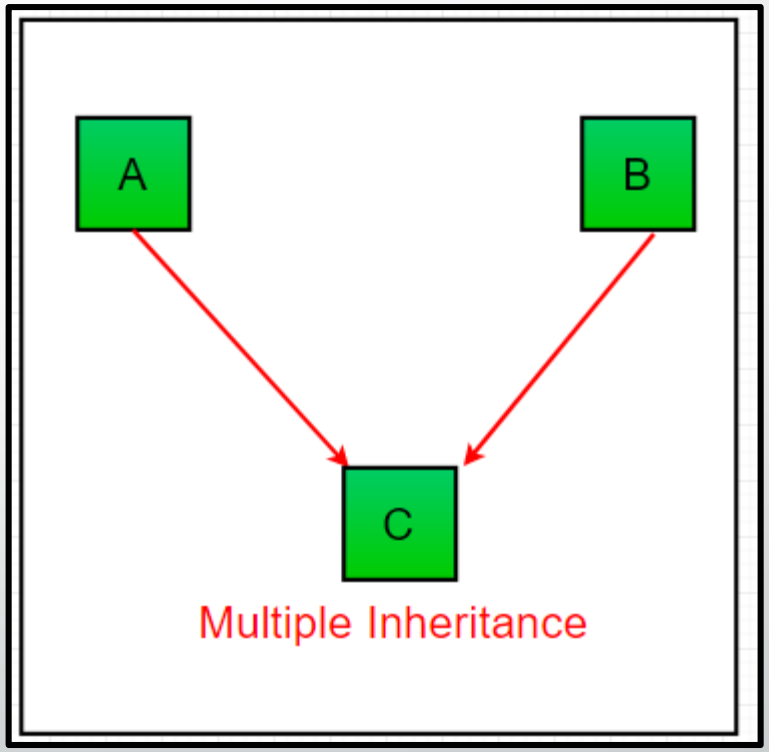
# 1.3.2 Multiple inheritance

Let us demonstrate this using an example:

```
class Father:
    def Summation(self,a,b):
        return a+b;
class Mother:
    def Multiplication(self,a,b):
        return a*b;
class Son(Father,Mother):
    def Divide(self,a,b):
        return a/b;
Jim = Son()
#all the three methods are available to multichild
```

**Figure 2**

*Multiple inheritance*




(From *Types of Inheritance Python*, 2020)

# 1.3.3 Multilevel inheritance

- In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather. (*Types of Inheritance Python*, 2020). There is no limit as to the number of levels, as long as they all inherit from each top to bottom; it is shown in Figure 3
- The syntax here is as follows:
- Class Great_great_ grandfather:
- <some methods>
- Class Great_grandfather:
- <some methods>
- …class Father:
- <some methods>

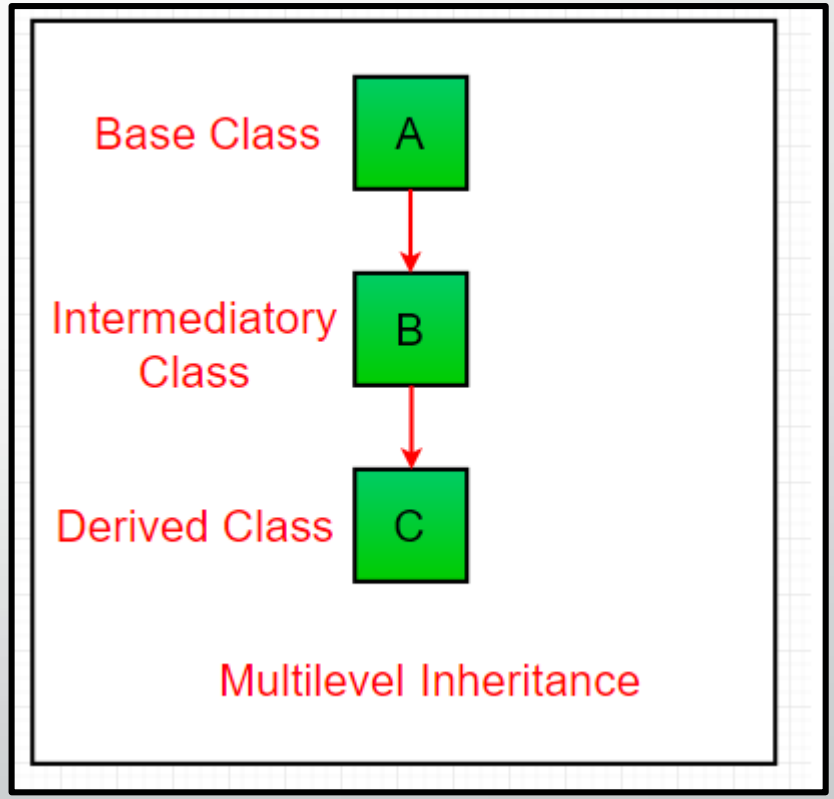
# 1.3.3 Multilevel inheritance

Let us examine an example(*Inheritance in Python - Javatpoint*, n.d.):

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog

class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
```

**Figure 3**
*Multilevel inheritance*




(From *Types of Inheritance Python*, 2020)

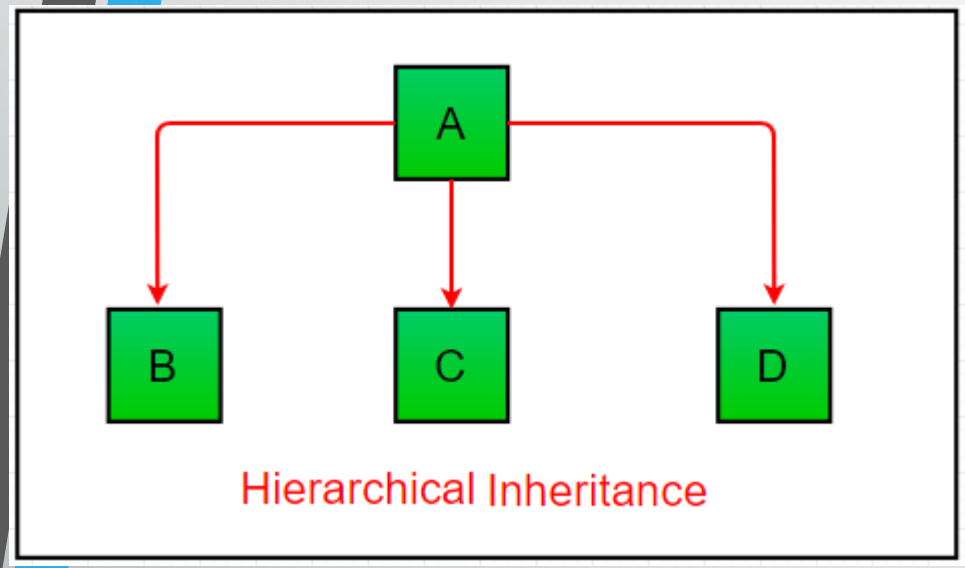
# 1.3.4 Hierarchical inheritance

- When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. (*Types of Inheritance Python*, 2020). The syntax is similar to single inheritance for each derived class. Figure 4 shows multiple inheritance.
- Consider the class Person from the example in single inheritance with the derived class being class Child. We could add another derived class Child2 as follows:

```
class Child2(Parent):
    def mychildfunc2(self):
        print("This function is in child2 class.")
```

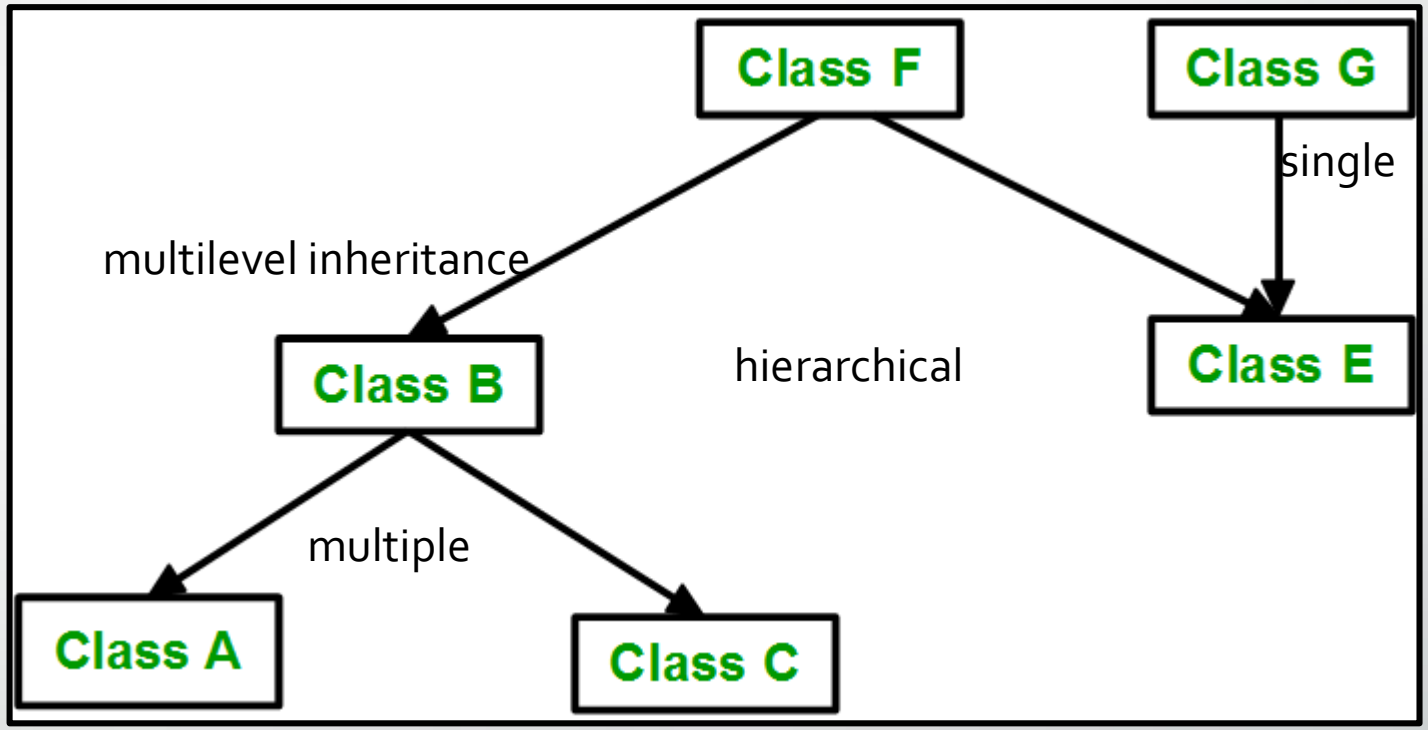
- We would then create objects of any of the derived classes and apply the methods appropriately.

**Figure 4**
*Hierarchical inheritance*




(From *Types of Inheritance Python*, 2020)

**Figure 5**
*Hybrid inheritance*




(From *Types of Inheritance Python*, 2020)

# 1.3.5 Hybrid inheritance

- Inheritance consisting of multiple types of inheritance is called hybrid inheritance. *(Types of Inheritance Python*, 2020).
- As shown in Figure 5, all the four types of inheritance are captured under one roof; there is single inheritance, multiple inheritance, multilevel inheritance, and hierarchical inheritance.

# 1.4 Method overriding

- We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class. (*Inheritance in Python - Javatpoint*, n.d.)
- Let us consider an example:

```
class Animal:
    def speak(self):
        print("speaking")
class Dog(Animal):
    def speak(self):
        print("Barking")
```

# 1.5 Polymorphism

- The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function. (*Polymorphism in Python*, 2018)
- If we consider the methods that are available to the collections object we notice polymorphism in the use of inbuilt methods such as *len*:

```
# len() being used for a string
print(len("polymorphism"))

# len() being used for a list
print(len([10, 20, 30]))
```

- We notice the same method len() being applied to a string and to a list.

# Part 2

Abstract classes

# 2.1 Introduction

- In lesson 5 we created three interfaces: the bag interface, which was then implemented by the arraybag and linkedbag interfaces. In this part we use the knowledge we have garnered in part 1 of this lesson to create an arraysorted bag. The sorted bag is similar to the bag but the requirements introduce three significant differences (Lambert, 2019):
- The sorted bag allows the client to visit its elements in sorted order, via the for loop.
- The sorted bag's in operator runs in logarithmic time.
- The items added to the bag must be comparable with each other. This means that they recognize the comparison operators <, <=, >, and >=, and that they are of the same type.
- Based on these requirements that the methods for this class are all the same as for the bag interface with the exception of three methods: __init__, add, and __contains__. We begin by using inheritance to create a sorted bag with those three methods.

# 2.2 Inheritance

- The ArrayBag implements the Bag interface; while the ArraySortedBag inherits from the ArrayBag interface. Do you notice the multilevel inheritance here? We can demonstrate it as:
- ArraySortedBag → ArrayBag ---- > Bag
- This means that ArraySortedBag also implements the Bag interface through multilevel inheritance. Next, remember that without inheritance we would have to copy all the code from the ArrayBag to the ArraySortedBag, then modify it to suit our requirements (if we didn't know better, we would not call this overriding right?)
- However, since we now know the rules of inheritance for a subclass we shall perform the following operations:

# 2.2 Inheritance

- Start by deleting all the methods that will not change; these will automatically be available to the subclass by inheritance. However, we still require the __init__ method in the sub class (this is the constructor)
- By inheritance rules remember to put the parent name in the parantheses of the child (recall Class <childname>(parentname))
- Modify the code for the methods that will change (method overriding), including the constructor (__init__)
- Add any new methods for the new subclass that are exclusive to it (meaning they are not used/declared in the parent class).
- Let us start by modifying the code for the most important method (the constructor).

# 2.2.1 Constructor

- The constructor is the __init__ method. This is a special method and isn't directly inherited *per se*. We need to call it from the parent class in those instances it does not need any modification.
- This means that the the __init__ method in ArraySortedBag must call the __init__ method in its parent class (ArrayBag) in order to initialize the data it has (what it will use). Based on inheritance the special syntax used here is:
- ArrayBag.__init__(self, sourceCollection)
- The explanation for this is that the parent class name, ArrayBag, enables Python to select which version of the __init__ method to run. When we run ArraySortedBag() to create a new instance of ArraySortedBag, Python runs the ArraySortedBag's __init__ method. This version of __init__ in turn must run the __init__ method in the parent class, ArrayBag. It does this by calling ArrayBag.__init__.

# 2.2.1 Constructor

- Further, self must refer to an instance of ArraySortedBag, not ArrayBag. That's why self is passed as an additional argument to ArrayBag.__init__. Put another way, the instance of ArraySorted bag says to ArrayBag's __init__ method, "I'm passing you a reference to myself, so you will use my own add method, not yours, to add the items from the optional source collection to myself." (Lambert, 2019)
- The optional source collection is also passed as an argument to the parent class's __init__ method. If the source collection is present, an interesting transaction occurs here. The source collection is passed *up* in the class hierarchy, from ArraySortedBag to its parent, ArrayBag (think of getting your parent to do some work for you). However, when this collection arrives in the context of ArrayBag's __init__ method, this method passes each of the collection's items back *down* to ArraySortedBag's add method to add them in the appropriate manner.
- Lastly since we know that a sorted bag is created the same way as a bag the sorted bag's constructor will have the same header as the constructor of its parent class. Let us see the constructor for the ArraySortedBag based on what we have discussed so far (Lambert, 2019):

# 2.2.1 Constructor

```
"""
File: arraysortedbag.py
Author: Ken Lambert
"""
from arraybag import ArrayBag
class ArraySortedBag(ArrayBag):
"""An array-based sorted bag implementation."""
    # Constructor
    def __init__(self, sourceCollection = None):
    """Sets the initial state of self, which includes the contents of sourceCollection, if it's present."""
        ArrayBag.__init__(self, sourceCollection)
```

# 2.3 New methods

- The only new method to add that is missing in the parent class is the contains method. Lambert (2019) explains how this method will work:
- "Python automatically generates a sequential search operation, using the ArrayBag iterator, when the **in** operator is used on a bag. To override this behavior for sorted bags, include a __contains__ method in ArraySortedBag. Now when Python sees the in operator used on a sorted bag, it also sees that bag's __contains__ method and calls it.
- This method implements a binary search… on the sorted bag's array of items. This array, named self.items and located in the ArrayBag class, is accessible in any of its subclasses. Thus, you can reference this variable directly during the search, as shown in the next code segment."
- We note that this is an accessor method as it will only work on the collection internally.

# 2.3 New methods

```
# Accessor methods
def __contains__(self, item):
"""Returns True if item is in self, or False otherwise."""
        left = 0
        right = len(self) - 1
        while left <= right:
                midPoint = (left + right) // 2
                if self.items[midPoint] == item:
                        return True
                elif self.items[midPoint] > item:
                        right = midPoint - 1
                else:
                        left = midPoint + 1
                return False
```

# 2.4 Overriding methods

- We need to override the add method in the ArraySortedBag. Since this bag sorts items it means it has to move them to certain positions in the collection.
- This means that we have to search for this position and two scenarios arise here: when the bag is empty or when the new item is greater than or equal to the last item—in which this search is unnecessary. In those cases, you can add the new item by passing it along to the add method in the ArrayBag class.
- If you can't get away with passing the task to ArrayBag, you must look in the array for the first item that's greater than or equal to the new item. Then open a hole for the new item, insert it, and increment the size of the bag. (Lambert, 2019).
- This allows us to override the add method in the ArraySortedBag as follows (Lambert, 2019):

```
# Mutator methods
def add(self, item):
    """Adds item to self."""
    # Empty or last item, call ArrayBag.add
    if self.isEmpty() or item >= self.items[len(self) - 1]:
        ArrayBag.add(self, item)
    else:
        # Resize the array if it is full here
        # Search for first item >= new item
        targetIndex = 0
        while item > self.items[targetIndex]:
            targetIndex += 1
        # Open a hole for new item
        for i in range(len(self), targetIndex, -1):
            self.items[i] = self.items[i - 1]
        # Insert item and update size
        self.items[targetIndex] = item
        self.size += 1
```

# 2.4 Overriding methods

- The only other method left to modify is the __add__ method:
- The __add__ method, which Python runs when it sees the + operator used with two bags, has practically the same code in the ArrayBag class and the LinkedBag class. The only difference is the class name used to create a new instance for the result bag. You simply repeat this pattern for __add__ in ArraySortedBag, as follows (Lambert, 2019):

```
def __add__(self, other):
    """Returns a new bag with the contents of self and other."""
    result = ArraySortedBag(self)
    for item in other:
        result.add(item)
    return result
```

# 2.5 Using ABC

- "In this section, you learn how to eliminate redundant methods and data in a set of existing classes by factoring the code for them into a common superclass. Such a class is called an **abstract class** to indicate that it captures the common features and behavior of a set of related classes.
- An abstract class is not normally instantiated in client applications. Its subclasses are called **concrete classes** to indicate that they are the classes actually used to create objects in client applications. Both categories of classes are in turn distinguished from *interfaces* (*see Lesson 5*), which simply specify the methods of a given class or set of classes without any implementing code." (Lambert, 2019)
- We wish to identify the redundancy in the code we have generated so far in order to create an Abstract class (ABC) for ease of use.

# 2.5 Using ABC

- Lambert (2019) describes the approach as follows:
- You can remove the redundant methods from the bag classes and place them in a new class called AbstractBag. The bag classes then access these methods via inheritance by becoming subclasses of AbstractBag. The modified framework of your bag classes is depicted in the class diagram (Figure 6).
- The AbstractBag class does not implement the bag interface. That's because only a subset of the bag methods are included in AbstractBag. The other three bag classes continue to conform to the bag interface.
- You now have a more obvious class hierarchy, with two levels of inheritance. The ArraySortedBag class now inherits some methods and data directly from its parent class ArrayBag, and other methods and data indirectly from its ancestor class AbstractBag. In general, the methods and variables of a class are available to any of its descendant classes.
- To create the AbstractBag class, you start by copying the contents of one of its subclasses to a new file and save that file as abstractbag.py. Then perform the following steps:

# 2.5 Using ABC

- 1. Delete any irrelevant imports and rename the class AbstractBag.
- 2. Delete all the methods that directly access the instance variable self.items, except for the __init__ method.
- As we had earlier discussed, the __init__ method is a special method and has to be redone for the AbstractBag as well.
- The __init__ method in AbstractBag performs two roles as described by Lambert (2019):
- 1. Introducing the variable self.size and initializing it to 0.
- 2. Adding the items from the source collection to self, if the source collection is present.
- Therefore, you delete the line of code that initializes the variable self.items. This code is still the responsibility of the __init__ method in the subclasses.
- The code for the __init__ method will now appear as follows:

```
"""
File: abstractbag.py
Author: Ken Lambert
"""
class AbstractBag(object):
"""An abstract bag implementation."""
    # Constructor
    def __init__(self, sourceCollection = None):
    """Sets the initial state of self, which includes the contents of sourceCollection, if it's present."""
        self.size = 0
        if sourceCollection:
            for item in sourceCollection:
                self.add(item)
```

# 2.5 Using ABC

- Each subclass of AbstractBag must now import this class, using the inheritance naming convention (name in parentheses) in the class header, omit the redundant methods mentioned earlier, and include a modified __init__ method.
- For example, the changes to the __init__ method in the ArrayBag collection are shared in the next slide. This method is still responsible for setting self.items to a new array, but that's the only line of code kept from before. After you run this code, run the __init__ method in AbstractBag, which initializes the bag's size and adds the items from the source collection if necessary: (Lambert, 2019)
- The code for the __init__ method of the LinkedBag will also be written in a similar manner.

```
"""
File: arraybag.py
Author: Ken Lambert
"""
from arrays import Array
from abstractbag import AbstractBag
class ArrayBag(AbstractBag):
"""An array-based bag implementation."""
# Class variable
DEFAULT_CAPACITY = 10
    # Constructor
    def __init__(self, sourceCollection = None):
    """Sets the initial state of self, which includes the contents of sourceCollection, if it's present."""
        self.items = Array(ArrayBag.DEFAULT_CAPACITY)
        AbstractBag.__init__(self, sourceCollection)
```

# 2.5 Using ABC

- Next we also modify the __add__ method of AbstractBag to allow it to work with all types of bags:

```
def __add__(self, other):
"""Returns a new bag containing the contents of self and other."""
        result = type(self)(self)
        for item in other:
                result.add(item)
        return result
```

- Python's type function is used to access the type of self and then use the resulting type to create a clone of self in the usual manner. If you don't do it in this manner and attempt to use the original __add__ method an excepton will be raised; the exception states that AbstractBag does not know about ArrayBag (or LinkedBag or ArraySortedBag, if that's the class from which you copied this method). Of course, AbstractBag cannot know anything about its subclasses. The cause of this error is that the __add__ method has attempted to create an instance of ArrayBag to hold its results. (Lambert, 2019)

# 2.6 One ABC

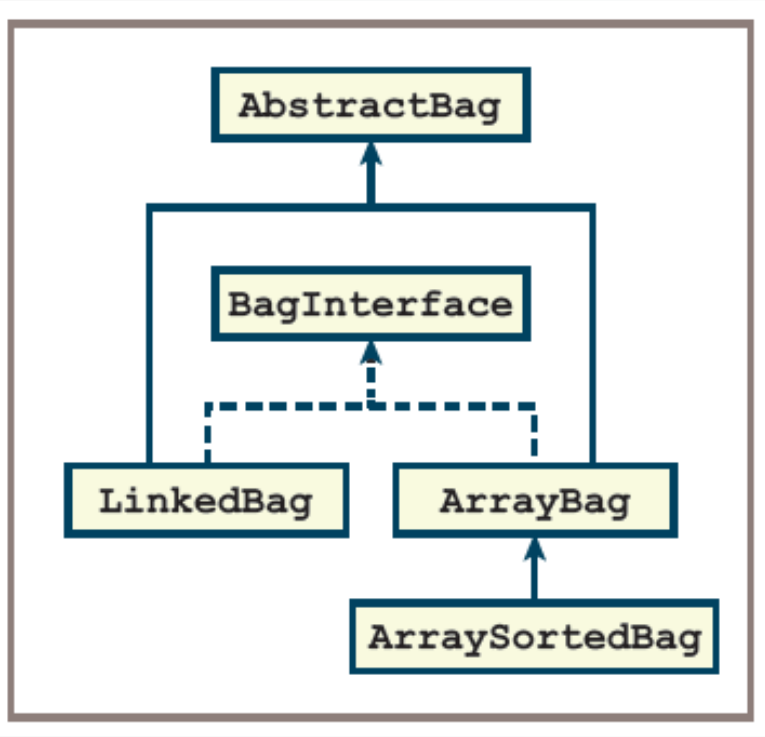
- Based on everything done so far we can now create on general AbstractCollection class for all collections as follows (Lambert, 2019):
- The AbstractCollection class is responsible for introducing and initializing the variable self.size. This variable is used by all the collection classes below it in the hierarchy.
- The __init__ method of AbstractCollection can also add the items from the source collection to self, if necessary.
- This class also includes the most general methods available to all collections: isEmpty, __len__, count, and __add__. “Most general” in this case means that their implementation need never be changed by a subclass.
- Finally, AbstractCollection also includes default implementations of the __str__ and __eq__ methods. Their current form in AbstractBag is appropriate for unordered collections, but most collection classes are likely to be linear rather than unordered. Therefore, these two methods are left as is in AbstractBag, but new implementations are provided in AbstractCollection.

# 2.6 One ABC

- The new __str__ method uses the square brackets to delimit the string, and the new __eq__ method compares pairs of items at given positions. New subclasses of AbstractCollection are still free to customize __str__ and __eq__ to suit their needs. Figure 7 shows the class diagram for the AbstractCollection.
- To create the AbstractCollection class we follow the procedure as we did for the ArraySortedBag collection; we copy the code from its child (AbstractBag, as seen in Figure 7) and then performing the following three steps (Lambert, 2019):
- Rename the class to AbstractCollection.
- Modify the __str__ and __eq__ methods to provide reasonable default behavior.
- Remove the isEmpty, __len__, count and __add__ methods from AbstractBag.

**Figure 6**

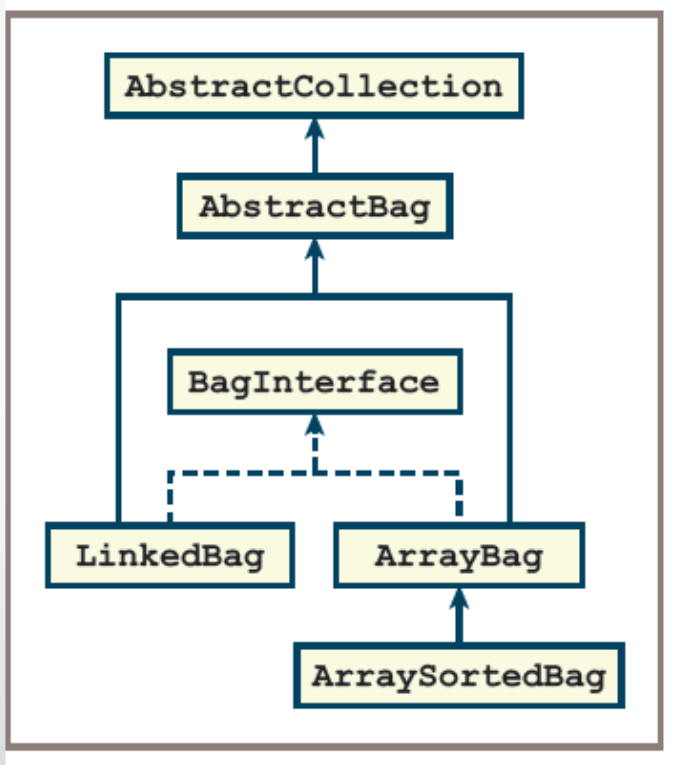
*Modified bag framework*




(From Lambert, 2019)

**Figure 7**

*AbstractCollection*




(From Lambert, 2019)

# Summary

- Inheritance: the concept of one class using the methods of another class. This class shares some characteristic(s) of the class whose methods it uses.
- There are four types of inheritance: single, multiple, multilevel, and hierarchical.
- Overriding: the ability of a class to inherit the methods of its parent class, and customize them to suit its needs.
- Polymorphism: is to take many forms. In Python this means the same action being performed in different ways. In polymorphism, a method can process objects differently depending on the class type or data type. Overriding is a form of polymorphism already.
- An abstract class is not normally instantiated in client applications. Its subclasses are called **concrete classes** to indicate that they are the classes actually used to create objects in client applications. Both categories of classes are in turn distinguished from *interfaces*, which simply specify the methods of a given class or set of classes without any implementing code.

# References


- *Inheritance in Python - javatpoint*. (n.d.). Www.javatpoint.com. Retrieved October 1, 2023, from https://www.javatpoint.com/inheritance-in-python
- Lambert, K. (2019). *Fundamentals of Python: Data Structures*. Cengage Learning.
- *Polymorphism in Python*. (2018, December 4). GeeksforGeeks. https://www.geeksforgeeks.org/polymorphism-in-python/
- *Types of inheritance Python*. (2020, January 14). GeeksforGeeks. https://www.geeksforgeeks.org/types-of-inheritance-python/