Data Structures & Algorithms

Week 5

Implementation in Object Oriented Environments

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 4

- A one-dimensional array is composed of multiple sequential elements stored in contiguous bytes of memory and allows for random access to the individual elements.
- An array (rather than the list) is the primary implementing structure in the collections of Python and many other programming languages (including C).
- While Python does not provide the array structure as part of the language itself, it now includes the ctypes module as part of the Python Standard Library. This module provides access to the diverse set of data types available in the C language and the complete functionality provided by a wide range of C libraries.
- Python also has the array module which can be used to create and manipulate arrays. In order to use this simply import it using the *import* command in Python.
- A dynamic array is just like a normal array. The difference between the two is that the size of a dynamic array can be dynamically modified at runtime. We don't need to specify the size of the array beforehand. In a dynamic array, once the contagious memory is filled, a bigger chunk of memory is allocated.
- After arrays, linked structures are probably the most commonly used data structures in programs. Like an array, a linked structure is a concrete data type that implements many types of collections, including lists
- Three types of linked structures : singly linked, doubly linked, and circular linked structures.

Content

- Interfaces
- The Bag Interface
- Coding Bag Interface
- Array Based Implementation
- Link Based Implementation
- Performance and Testing
- UML Class Diagram

Part 1

Interfaces

1.1 Introduction

- We have come across interfaces in our day to day life, sometimes without even realizing it. An interface is simply a go-between two 'systems' if we can call it that.
- Your face is an interface between your brain and the person you are talking to, for example. The expressions produced by your face tell other people how you are feeling; when you are happy you laugh, when you are in grief you cry; when you are embarrassed it produces a different expression, and so on. All these expressions are based on how your brain receives input from the environment; for example, you receive news that a loved one has passed on (input), you are immediately grieved and you express it by crying or screaming (or both).
- Some people have taught themselves not to show any expressions, or to pretend (note they have taught themselves or been taught as this isn't natural) by showing expressions suiting a particular situation. For example political leaders read the crowd mood and adjust accordingly; we have professional mourners in who will come and cry and wail at a funeral, and so on.

1.1 Introduction

- However, have you given thought to all the processes that go on in the brain before your (inter)face expresses an emotion? The brain is the most complex machine in the whole (known) universe. Fortunately for the other party they are spared the working of the brain and only get to see the expression on your face; based on it they can relate / react / respond to what they see and hear from you.
- In earlier lessons it has been shared that computing is mostly modelled on humans; humans receive input (from the environment), process it (give it meaning in the brain and decide what action to take; no action in this case is also an action), then perform some action based on the processing (output). And back to basics, we know that a computer has both hardware and software.
- It is not surprising therefore that the interface principle has been borrowed from the human and applied in the computer world. Let us examine a simple hardware example, the case of a USB flash disk. You will normally store some information on the flash disk, like maybe videos, music, or even important files that require portability. The flash disk has its storage component and an interface that allows you to plug in to the computer so that you can view/edit/read your files. Ever thought about how this interface works? I don't think so; you just know that you plug it in and, voila, the contents of your flash will pop up on the screen.

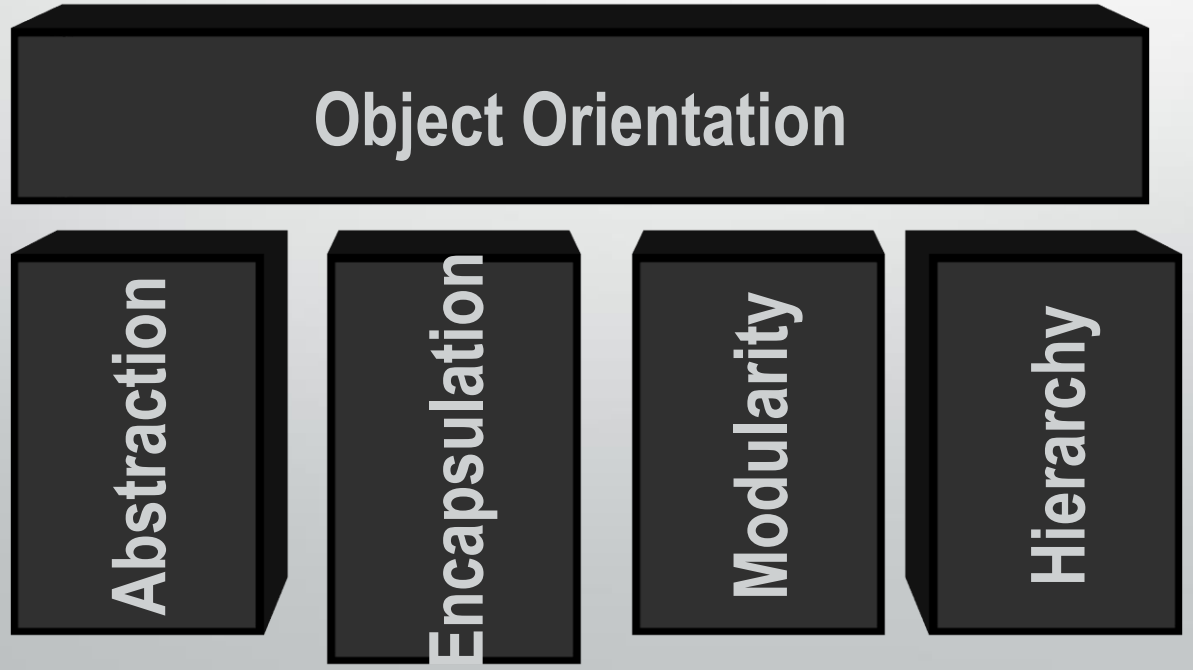
1.1 Introduction

- How about software? Let's consider an operating system like Windows for example; most shortcuts are on the desktop, you just double click and the required application opens up and you are good to go. Windows has a powerful user interface that responds to user actions; the user is not bothered with HOW an action occurs, just that it occurs. When it doesn't happen as expected the ICT support will get calls from users saying "my computer has refused".
- Users also communicate with applications via their interfaces; this is one of the reasons UX (user experience) has blown up to be such a big thing in today's technology world; how the user interacts with the application and their whole experience is the buzz of programming today. When using an application the user is saved from the internal working of the program (they don't see the code for example, or how different modules call each other in the background); s/he is just concerned with the ease of use of the interface (including layout and clarity of buttons, the help feature, and so on).
- The concept of abstraction ensconces the user; just like with the feature we call encapsulation, programs are designed to hide the implementation details from the user. This saves the user having to know the details (phew!), and also the user can't interfere with the source code of the program(s).

1.2 Object oriented concepts

- Before looking at the main topic of this section it is worthwhile to do a quick recap of the pillars of object orientation, or object oriented programming. Even though it is assumed the learner has a basic knowledge of programming it is assumed it is not always in an object oriented programming language; if so this will serve as a good refresher.
- There are four pillars of object orientation shown in Figure 1.

Figure 1
Pillars of object orientation



1.2 Object oriented concepts

- Abstraction: this principle is about only showing what is necessary, and hiding the implementation details. In another way it means only showing what is necessary to get something done, while hiding other details. This saves from distraction (complexity) and also increases efficiency (can you imagine having to access the program directly through its code? That would be nerve wrecking for most users!) We talk of increasing levels of abstraction in programming; that is to say, as we delve into more and more details we decrease the degree of abstraction. For example, at the highest level of abstraction you know this course is taught by a professor; decrease the abstraction to know his name (yours truly); decrease it further to know his gender; further to know his descent, then which university he is based in, his height, and so on.
- Encapsulation: encapsulation as you may guess from the English world is to put together into one. In object orientation we put together (bundle) the related data and programs into one unit (for example a class), and we can then work with them easily. The class acts as a blueprint for creating objects that will access its methods (operations).

1.2 Object oriented concepts

- Modularity: If you have been involved in procurement of enterprise software you will notice them talking of different program modules. A modular approach in programming involves breaking down a program into smaller manageable parts (called modules). This helps in reducing the complexity of the program.
- Hierarchy: this refers to the use of different levels in a program, a hierarchy so to speak. This is used in object orientation to allow classes to inherit the characteristics (features) of other classes by arranging them in a hierarchy. Thus those classes at the top are called parents, and the ones lower in the hierarchy are called children (again, like humans). Again just like humans, child classes inherit the attributes (characteristics) and methods (operations) of their parents. This means you don't have to keep rewriting code for the child classes; all you have to do is to declare that Y is a child of X and the language immediately does the association.

1.3 Interfaces in Python

- 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. (*Python - Interfaces*, n.d.). Let us not worry about the term overriding as we shall discuss it more in detail on our next lesson (lesson 6).
- Python further deviates from other languages in one other aspect. It doesn't require the class that's implementing the interface to define all of the interface's abstract methods. (Murphy, n.d.)
- Thus Python can create an interface in two ways: formal and informal. The difference between the two is discussed next.

1.3 Interfaces in Python

- "A python informal interface is also a class that defines methods that can be overridden but without force enforcement. An informal interface also called Protocols or Duck Typing… A formal Interface is an interface which enforced formally…" (Banu, 2020)
- To create a formal interface, we need to use ABCs (Abstract Base Classes).
- An ABC is simple as an interface or base classes define as an abstract class in nature and the abstract class contains some methods as abstract. Next, if any classes or objects implement or drive from these base classes, then these bases classes forced to implements all those methods. Note that the interface cannot be instantiated, which means that we cannot create the object of the interface. So we use a base class to create an object, and we can say that the object implements an interface. And we will use the type function to confirm that the object implements a particular interface or not. (Banu, 2020).

1.3 Interfaces in Python

- In this lesson we use the bag collection to create an interface that can be used by objects. The bag interface allows clients to use bags effectively and allows implementers to produce new classes that implement this interface.

Part 2

The bag interface

2.1 Introduction

- A bag belongs to a group of collections known as unordered collections. Recall that in lesson 2 linear and non-linear collections were discussed.
- Unordered collections include bags, dictionaries, and sets. If we consider an example of a bag of sweets (candy),it will demonstrate the term unordered more easily. You can put different types of candy in the bag, and take them out; however, they aren't arranged in any particular order in the bag. At best you can perform a visual check to take out the particular one you would like. Otherwise there is no order in how they are arranged in the bag.
- Table 1 provides the categories of operations that are performed on collections. In terms of implementation, users view collection types as abstractions; they are concerned with what they can do with an ADT (such as an array or a bag for that matter), but not HOW it does it. The issue of how it does it belongs to the developer. How does the developer ensure the 'how' part?

2.1 Introduction

- Programming languages such as Java provide several implementations of collections (for example Java has the java.util package provides implementation for arrays and lists); however, this is not the case with Python. With Python the programmer is expected to provide the implementation through classes, which contain methods to implement the operations of the said classes. This is done through creating interfaces which act as abstract classes providing methods for the objects of the class to implement.
- In this lesson we consider the implementation of the bag collection; the code for design and implementation is provided by Lambert (2019).

Table 1

Collection operations

Category of Operation	Description
Determine the size	Use Python's `len` function to obtain the number of items currently in the collection.
Test for item membership	Use Python's `in` operator to search for a given target item in the collection. Returns `True` if the item is found, or `False` otherwise.
Traverse the collection	Use Python's `for` loop to visit each item in the collection. The order in which the items are visited depends upon the type of collection.
Obtain a string representation	Use Python's `str` function to obtain the string representation of the collection.
Test for equality	Use Python's `==` operator to determine whether two collections are equal. Two collections are equal if they are of the same type and contain the same items. The order in which pairs of items are compared depends on the type of collection.
Concatenate two collections	Use Python's `+` operator to obtain a new collection of the same type as the operands, and containing the items in the two operands.
Convert to another type of collection	Create a new collection with the same items as a source collection. Cloning is a special case of type conversion, where the two collections are of the same type.
Insert an item	Add the item to the collection, possibly at a given position.
Remove an item	Remove the item from the collection, possibly at a given position.
Replace an item	Combine removal and insertion into one operation.
Access or retrieve an item	Obtain an item, possibly at a given position.

(From Lambert, 2019)

2.2 Designing the interface

- We have already mentioned that object orientation is based on the real world. It therefore helps to consider the bag from the real world perspective (as we did with the candy example earlier in this lesson).
- We are not restricted as to what a bag can contain; it can contain candy, handkerchiefs, toiletries, shoes, and so on. Arising therefrom, we also need to know what we can do with the bag; how items are removed, added, how to view items in the bag without emptying it, and so on.
- Other useful operations that Lambert (2019) shares include to "determine whether two bags contain the same objects and combine the contents of two bags into a third bag (**concatenation**). Last but not least, you need to know how to create a bag—one that is initially empty or one that comes already filled with the contents of another collection."

2.2 Designing the interface

- The next step is to draw up a list of function names, method names, and operator symbols that meet the descriptions of the identified (expected) operations of the bag.
- Lambert (2019) advises that "You should strive to conform to common, conventional usage when selecting method or function names. For example, in the case of collections, the functions len and str always designate operations that return a collection's length and its string representation, respectively. The operator symbols +, ==, and in always stand for the operations of concatenation, equality, and item membership. A for loop is used to visit all of a collection's items. The methods add and remove, in addition to having the obvious meanings, also belong to the interfaces of other collections, such as sets. Where convention is lacking, use common sense: the meanings of the methods isEmpty and clear are immediately recognizable. (The latter empties a bag.)"
- Consequently, the functions and method names for the bag interface will be: add, clear, count, for ..., in, isEmpty, len, remove, str, +, and ==; the names imply the meanings and should not be difficult to decipher. For example, add will simply add something to the bag, remove is to take out an item, and so on.

2.2 Designing the interface

- Since an interface is abstract (the user just needs to know what an operation does and not how it does it) the next step is to try and imagine what a user will expect the interface to do, what arguments s/he will provide, and what returns (results) they will expect. Remember the 'how' of the interface is the work of the implementer (that means us who are developing the interface!).
- A simple way to do this is to perform some operations as you would do when running an ordinary Python program, and apply these to the interface to be designed. The user would expect to be able to add an item to the bag, print its contents (to see what's in it), find an item in the bag, remove an item, and so on. How would they do this with existing Python commands?
- Assume the bag is x, some of these operations are shown in the next slide (21). We notice that most of these user operations and methods have implementations in Python; thus we need to find a way to map these to the equivalent Python operation. This is shown in table 2.

2.2 Designing the interface

```
x.clear() # Make the bag empty
for item in range(10):  # Add 10 numbers to the bag
x.add(item)   #add an item to the bag
print(x)   # Print the contents of the bag
print(4 in x)   # check whether 4 is in the bag?
p = x + x   # Contents replicated in a new bag
print(len(p))   # 10 numbers
for item in p: # Print them all individually
        print(item)
for item in range(5): # Remove half of them
        p.remove(item)
print(p == x)
```

Table 2

User operations and arguments for Bag

User's Bag Operation	Method in a Bag Class
b = <class name>(<optional collection>)	__init__(self, sourceCollection = None)
b.add(item)	add(self, item)
b.clear()	clear(self)
b.count(item)	count(self, item)
b.isEmpty()	isEmpty(self)
b.remove(item)	remove(self, item)
len(b)	__len__(self)
str(b)	__str__(self)
for item in b:	__iter__(self)
item in b	__contains__(self, item)
	Not needed if __iter__ is included
b1 + b2	__add__(self, other)
b == anyObject	__eq__(self, other)

(From Lambert, 2019)

2.3 Special method(constructor)

- 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. Recall that we earlier mentioned that a class is a blueprint? Well, what can you do with a blueprint say of a building? You use it to make the physical building that people will dwell in; the blueprint by itself is not the building. Similarly the class is a blueprint; to use it effectively you must create instance of the class that will use the methods of the class; this is what is referred to as the object.
- In table 2, the first row contains this special method called the constructor. On the left is how the user will call it, while on the right is the equivalent Python implementation (what the user will not see, only the implementer will). How do we use it? We can create an empty bag or a bag containing some items (what is referred to as 'optional collection').

2.3 Special method(constructor)

- For example suppose we have an implementing class called ArrayBag and another called LinkedBag we create objects for each as follows:
- from arraybag import ArrayBag #import the ArrayBag class to be used here
- from linkedbag import LinkedBag #import the LinkedBag class to be used here
- bag1 = LinkedBag() #an empty LinkedBag object
- bag2 = ArrayBag([100, 92, 2, 50]) #an ArrayBag object utilizing the optional #collection component of the constructor
- Lastly determine whether there are any pre or post conditions.

2.4 Special conditions

- Pre-condition is a statement or set of statements that outline a condition that should be true when an action is called. The precondition statement indicates what must be true before the function is called. For example to calculate the square root of a number a precondition is that it must be greater than zero. (*Pre-Condition*, 2023)
- Post Condition is a statement or set of statements describing the outcome of an action if true when the operation has completed its task. The Post Conditions statement indicates what will be true when the action finishes its task. For example, To identify the square root of a number, the precondition is that the number should be greater than zero. The POST Condition is that the square root of the number is displayed on the console. (*Post Condition*, 2023)
- Lambert (2019) adds that "Documentation in an interface should also include a statement of any exceptions that could be raised, usually as the result of the failure to adhere to the preconditions of a method. For example, a bag's remove method might raise a KeyError if the target item is not in the bag."

Part 3

Coding Bag Interface

3.1 Method

- As described earlier programming languages like Java provide the syntax for coding interfaces, unlike Python. Java provides the methods in the classes that objects must use.
- We can do the same by creating our own classes in Python and copying how Java does it. Lambert (2019) provides a classical example of how to go about creating an interface for the bag collection. This interface can't be used in practical (in fact he refers to it as a pseudo interface); however, we use it to demonstrate the steps and actual code that you can use.
- In order to use this code in practice, you will need to modify it to suit the requirements of your bag.
- We begin by describing the steps to take when designing the interface, followed by the actual code for our demonstration interface.

3.1 Method

- The steps are as follows (Lambert, 2019):
- 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.
- So that the method headers can be checked with the compiler, place them within a class whose suffix is "Interface."
- We demonstrate this with the baginterface program (baginterface.py) written by Lambert.

3.2 Baginterface.py

```
"""
File: baginterface.py
Author: Ken Lambert
"""
class BagInterface(object):
"""Interface for all bag types."""
    # Constructor
    def __init__(self, sourceCollection =
    None):
"""Sets the initial state of self, which includes
the contents of sourceCollection, if it's
present."""
        pass
    # Accessor methods
    def isEmpty(self):
"""Returns True if len(self) == 0,
or False otherwise."""
        return True

    def __len__(self):
"""Returns the number of items in self."""
        return 0
    def __str__(self):
"""Returns the string representation of
self."""
        return ""
```

3.2 Baginterface.py

```
def __iter__(self):
"""Supports iteration over a view of self."""
    return None
def __add__(self, other):
"""Returns a new bag containing the
contents of self and other."""
    return None
def __eq__(self, other):
"""Returns True if self equals other,
or False otherwise."""
    return False
def count(self, item):
"""Returns the number of instances of item
in self."""
    return 0
# Mutator methods
def clear(self):
"""Makes self become empty."""
    pass
def add(self, item):
"""Adds item to self."""
    pass
```

3.2 Baginterface.py

```
def remove(self, item):
"""Precondition: item is in self.
Raises: KeyError if item in not in self.
Postcondition: item is removed from self."""
    Pass
```

- We now have our blueprint class for the bag interface. We can now implement it in different bag implementations. We consider an array based implementation and a link based implementation for demonstration purposes.

Part 4

Array Based Implementation

4.1 Method

- In this section we wish to code implementation for a class called ArrayBag which is an array based implementation of the bag class.
- We wish to use the blueprint(baginterface.py) to develop our ArrayBag class. This means that we shall use the methods that we defined in the baginterface.py class. The difference here is that we shall now provide code on how the ArrayBag class will use these methods(both accessor and mutator methods). We go about this in the following manner:
- Choose an appropriate data structure to contain the collection's items and determine any other data that might be needed to represent the state of the collection. These data are assigned to instance variables in the __init__ method.
- Complete the code for the methods specified in the interface, always starting with the constructor, then the easy methods and lastly, the rest. (Lambert, 2019)

4.2 Implementation

- In this example we use the arrays type from arrays module.
- Let us follow the steps as demonstrated by Lambert (2019):
- The __init__ method is responsible for setting up the initial state of a collection (the constructor). Therefore, this method creates an array with an initial, default capacity and assigns this array to an instance variable named self.items. Because the default capacity is the same for all instances of ArrayBag, it's defined as a class variable (available to all members of the class). The default capacity is a fairly small value, such as 10, for reasons of economy; however, a default value of 0 is set.
- After initializing the two instance variables, the __init__ method must deal with the possibility that its caller has provided a source collection parameter. If that is the case, all the data in the source collection must be added to the new ArrayBag object. Fortunately the baginterface code already has a method for adding items and this is what we shall use.
- The code for this part of the design is easy to create. You just make a copy of the bag interface file, baginterface.py, and rename it to arraybag.py. You then add an import statement for the array, rename the class to ArrayBag, add a class variable for the default capacity, and complete the __init__ method.

4.2.1 Constructor code

```
"""
File: arraybag.py
Author: Ken Lambert
"""
from arrays import Array
class ArrayBag(object):
"""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)
    self.size = 0
    if sourceCollection:
        for item in sourceCollection:
            self.add(item)
```

4.2.2 Easy methods

- A cursory analysis shows the simplest methods in this interface are isEmpty, __len__, and clear; these methods simply check the array to see if its empty, return the length of the array, and clear its contents, respectively. If you ignore the problem of the array becoming full for now, the add method is also fairly simple. Here is the new code for these four methods, starting with the accessor methods, followed by the mutator methods:

```
# Accessor methods

def isEmpty(self):

"""Returns True if len(self) == 0, or False otherwise."""

    return len(self) == 0

def __len__(self):

"""Returns the number of items in self."""

    return self.size
```

4.2.2 Easy methods

```
# Mutator methods
def clear(self):
"""Makes self become empty."""
    self.size = 0
    self.items = Array(ArrayBag.DEFAULT_CAPACITY)
def add(self, item):
"""Adds item to self."""
# Check array memory here and increase it if necessary
    self.items[len(self)] = item
    self.size += 1
```

- The next method to build is the __iter__ method which allows for navigation of the array. Once this is done we can then complete the remaing methods. These are shared in the next few slides.

4.2.3 Remaining methods

```
def __iter__(self):
"""Supports iteration over a view of self."""
    cursor = 0
    while cursor < len(self):
        yield self.items[cursor]
    cursor += 1
def __str__(self):
"""Returns the string representation of self."""
    return "{" + ", ".join(map(str, self)) + "}"
def __add__(self, other):
"""Returns a new bag containing the contents
of self and other."""
    result = ArrayBag(self)
    for item in other:
        result.add(item)
    return result
def __eq__(self, other):
"""Returns True if self equals other, or False
otherwise."""
    if self is other:
        return True
    if type(self) != type(other) or \
    len(self) != len(other):
        return False
    for item in self:
        if self.count(item) != other.count(item):
            return False
    return True
```

4.2.3 Remaining methods

```
def remove(self, item):
"""Precondition: item is in self.
Raises: KeyError if item in not in self.
postcondition: item is removed from self."""
# 1. check precondition and raise an exception if necessary
    if not item in self:
        raise KeyError(str(item) + " not in bag")
# 2. Search for index of target item
    targetIndex = 0
    for targetItem in self:
        if targetItem == item:
            break
    targetIndex += 1
# 3. Shift items to the right of target left by one position
    for i in range(targetIndex, len(self) - 1):
        self.items[i] = self.items[i + 1]
# 4. Decrement logical size
    self.size -= 1
# 5. Check array memory here and decrease it if necessary
```

Part 5

Link Based Implementation

5.1 Method

- Links were described in lesson 2. therefore when developing this implementation we should remember the notion of the node (in this implementation we use a single linked structure so no tail).
- Secondly we still require to implement code for all the methods of the baginterface to the linkedbag as we did for the arraybag.
- The only methods that will have different implementations in the LinkedBag class are those that cannot avoid this direct access to data: __init__, __iter__, clear, add, and remove. (Lambert, 2019).
- Bearing this in mind we now develop the code in a similar manner as we did for the ArrayBag, and the resulting code (linkedbag.py) is as follows:

```
"""
File: linkedbag.py
Author: Ken Lambert
"""
from node import Node
class LinkedBag(object):
"""A link-based 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.items = None
        self.size = 0
        if sourceCollection:
        for item in sourceCollection:
            self.add(item)
def __iter__(self):
"""Supports iteration over a view of self."""
    cursor = self.items
    while cursor != None:
        yield cursor.data
        cursor = cursor.next
def add(self, item):
"""Adds item to self."""
    self.items = Node(item, self.items)
    self.size += 1
```

```
def remove(self, item):
    """Precondition: item is in self.
    Raises: KeyError if item is not in self.
    Postcondition: item is removed from self."""
    # Check precondition and raise an exception if necessary
        if not item in self:
            raise KeyError(str(item) + " not in bag")
    # Search for the node containing the target item
    # probe will point to the target node, and trailer
    # will point to the node before it, if it exists
        probe = self.items
        trailer = None
    for targetItem in self:
        if targetItem == item:
            break
        trailer = probe
        probe = probe.next
    # Unhook the node to be deleted, either the first one or
    # one thereafter
    if probe == self.items:
        self.items = self.items.next
    else:
        Trailer.next = probe.next
    # Decrement logical size
    self.size -= 1
```

Part 6

Performance and testing

6.1 Performance

- Lambert (2019) reports that the running time for both implementations is very similar:"
- The in and remove operations take linear time in both implementations, because they incorporate a sequential search. The remove operation in ArrayBag must do the additional work of shifting data items in the array, but the cumulative effect is not worse than linear.
- The +, str, and iter operations are linear, as would be expected for any collection.
- The remaining operations are constant time, although ArrayBag's add incurs an occasional linear time hit to resize the array.
- The two implementations have the expected memory trade-offs. When the array within an ArrayBag is better than half full, it uses less memory than a LinkedBag of the same logical size. In the worst case, a LinkedBag uses twice as much memory as an ArrayBag whose array is full.
- Because of these memory trade-offs, removals from an ArrayBag are generally slower than they are on a LinkedBag."

6.2 Testing

- Once all the coding is complete you need to test it to ensure that it meets the requirements envisioned. This can be done using different types of tools and approaches (most of which are beyond the scope of this course).
- However, a simpler way of doing this is to just run a tester program that will check all the code we have written for the two implementations.
- Lambert (2019) provides the code for the bag class; you can test this on a standalone environment to see if your code is running correctly. The code is provided in slide 46. The name of the file is testbag.py

```
"""
File: testbag.py
Author: Ken Lambert
A tester program for bag implementations.
"""
from arraybag import ArrayBag
from linkedbag import LinkedBag
def test(bagType):
"""Expects a bag type as an argument and runs some tests
on objects of that type."""
print("Testing", bagType)
lyst = [2013, 61, 1973]
print("The list of items added is:", lyst)
)
b1 = bagType(lyst)
print("Length, expect 3:", len(b1))
print("Expect the bag's string:", b1)
print("2013 in bag, expect True:", 2013 in b1)
print("2012 in bag, expect False:", 2012 in b1)
print("Expect the items on separate lines:")
for item in b1:
        print(item)
b1.clear()
print("Clearing the bag, expect {}:", b1)
b1.add(25)
b1.remove(25)
print("Adding and then removing 25, expect {}:",
b1)
b1 = bagType(lyst)
b2 = bagType(b1)
print("Cloning the bag, expect True for ==:", b1 ==
b2)
```

6.2 Testing

```
print("+ the two bags, expect two of each item:", b1 + b2)
for item in lyst:
b1.remove(item)
print("Remove all items, expect {}:", b1)
print("Removing nonexistent item, expect crash with KeyError:")
b2.remove(99)
test(ArrayBag)
# test(LinkedBag)
```

- This program can also be run for the LinkedBag (which is commented out for this first part) or any other implementation of the bag interface. As mentioned earlier in the lesson the whole idea is to have the interface as is, while having different implementations of it.

Part 7

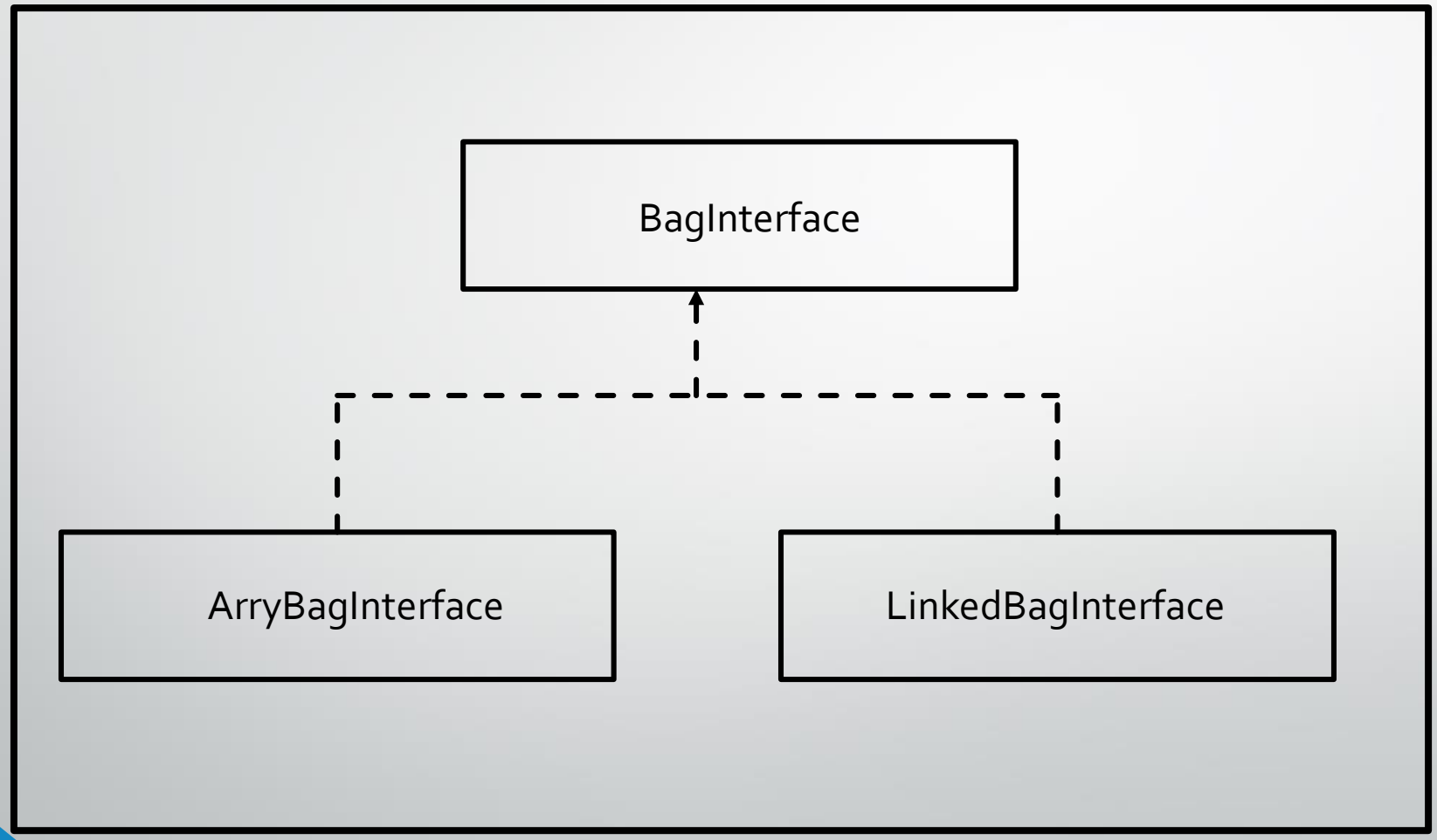
UML Class diagram

7.1 Introduction

- The class diagram is used in the UML (Unified Modelling Language) to show the relationship between different classes at different levels of detail. In the diagram you show how one class relates to another; you can show how many instances of each class exist in a given relationship, and so on.
- Going into the details of the class diagram is beyond the scope of this course (we need a whole separate lesson for this, ha!) but we will describe just three relationships in order to understand the class diagrams (Figure 1 and Figure 2) that follow:
- Dependency: denoted by a dotted arrow ---- > shows that one class depends on another; if you change one object's interface, you need to change the dependent object. The arrow points from dependent to needed objects.
- Aggregation: it is a whole-part relationship, shown with an unfilled diamond shape. The whole is where the diamond shape is situated, and the part can exist without the whole.
- Composition: is a more aggressive form of aggregation; in this instance removal of the whole results in removal of the part.

Figure 1

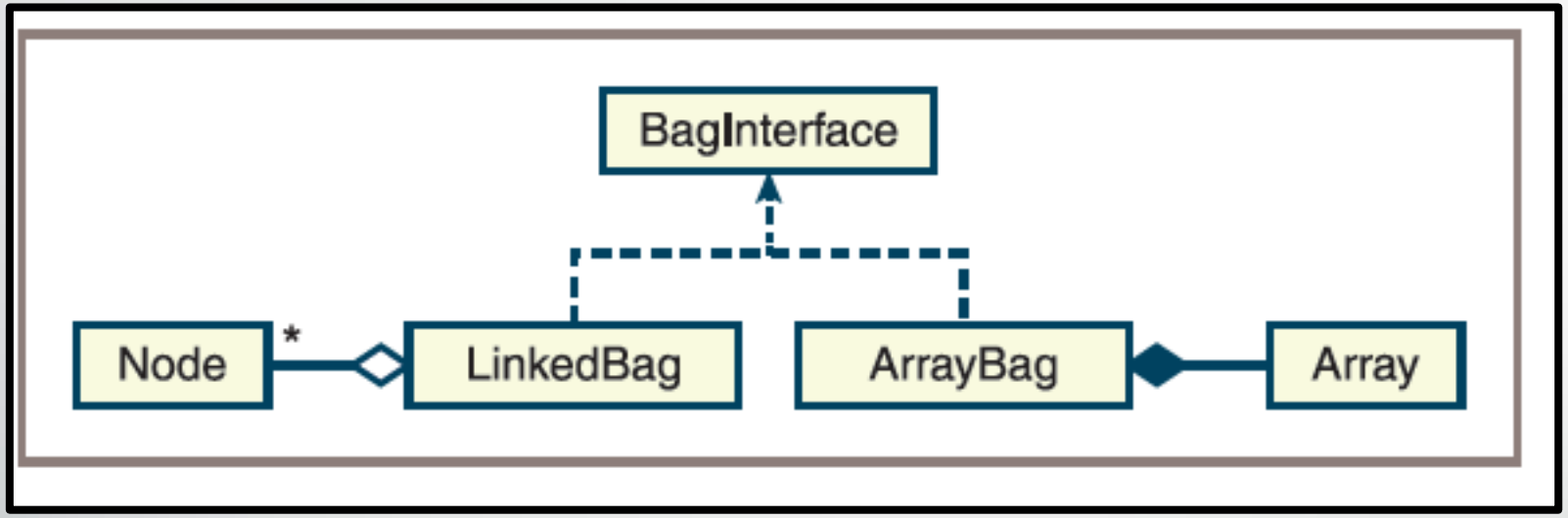
Class diagram showing interface relationship



(Adapted from Lambert, 2019)

Figure 2

Class diagram for interfaces



(From Lambert, 2019)

Summary

- 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.

References

- Banu, A. (2020, April 4). *Interface in Python | How to Create Interface in Python with Examples*. EDUCBA. https://www.educba.com/interface-in-python/
- Lambert, K. (2019). *Fundamentals of Python: Data Structures*. Cengage Learning.
- Murphy, W. (n.d.). *Implementing an Interface in Python – Real Python*. Realpython.com. Retrieved September 30, 2023, from https://realpython.com/python-interface/#python-interface-overview
- *Post Condition*. (2023). Www.tutorialspoint.com. https://www.tutorialspoint.com/software_testing_dictionary/post_condition.htm
- *Pre-Condition*. (2023). Www.tutorialspoint.com. https://www.tutorialspoint.com/software_testing_dictionary/pre_condition.htm
- *Python - Interfaces*. (n.d.). Www.tutorialspoint.com. Retrieved September 30, 2023, from https://www.tutorialspoint.com/python/python_interfaces.htm