2 OOP using Python

❖ Handling exceptions

We have treated exceptions as fatal events. When an exception is raised, the program terminates, and we go back to our code and attempt to figure out what went wrong. When an exception is raised that causes the program to terminate, we say that an unhandled exception has been raised.

An exception does not need to lead to program termination. Exceptions, when raised, can and should be handled by the program. Sometimes an exception is raised because there is a bug in the program (like accessing a variable that doesn't exist), but many times, an exception is something the programmer can and should anticipate. A program might try to open a file that does not exist. If an interactive program asks a user for input, the user might enter something inappropriate.

Exception handling enables you handle errors gracefully and do something meaningful about it. Like display a message to user if intended file not found. Python handles exception using try .. except .. block.

```
try:
    # write some code
    # that might throw exception
except <ExceptionType>:
    # Exception handler, alert the user
```

- First statement between try and except block is executed.
- If no exception occurs then code under except clause will be skipped.
- If file don’t exists then exception will be raised and the rest of the code in the try block will be skipped
- When exceptions occurs, if the exception type matches exception name after except keyword, then the code in that except clause is executed.
- A try statement can have more than once except clause, It can also have optional else and/or finally statement.

```
try: <body>
except <ExceptionType1>:
<handler1>
except <ExceptionTypeN>:
<handlerN>
finally:
<process_finally>
```

Example:

```
try:
    num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
    result = num1 / num2
    print("Result is", result)

except ZeroDivisionError:
    print("Division by zero is error !!")

except SyntaxError:
    print("Comma is missing. Enter numbers separated by comma like this 1, 2")

except:
    print("Wrong input")

else:
    print("No exceptions")

finally:
    print("This will execute no matter what")
```

List all the standard Exceptions available in Python

Exception Name	Description
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.

EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.

NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

❖ Exceptions as a control flow mechanism

Don't think of exceptions as purely for errors. They are a convenient flow-ofcontrol mechanism that can be used to simplify programs. In many programming languages, the standard approach to dealing with errors is to have functions return a value (often something analogous to Python's none) indicating that something has gone amiss. Each function invocation has to check whether that value has been returned. In Python, it is more usual to have a function raise an exception when it cannot produce a result that is consistent with the function's specification.

The Python raise statement forces a specified exception to occur. The form of a raise statement is

raise exceptionName(arguments)

The exceptionName is usually one of the built-in exceptions, e.g., ValueError. However, programmers can define new exceptions by creating a subclass of the built-in class Exception. Different types of exceptions can have different types of arguments, but most of the time the argument is a single string, which is used to describe the reason the exception is being raised.

❖ Assertions

The Python assert statement provides programmers with a simple way to confirm that the state of the computation is as expected. An assert statement can take one of two forms:

assert Boolean expression
or
assert Boolean expression, argument

When an assert statement is encountered, the Boolean expression is evaluated. If it evaluates to True, execution proceeds on its merry way. If it evaluates to False, an AssertionError exception is raised. Assertions are a useful defensive programming tool. They can be used to confirm that the arguments to a function are of appropriate types. They are also a useful debugging tool. The can be used, for example, to confirm that intermediate values have the expected values or that a function returns an acceptable value.

```
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32
print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
```

❖ Abstract Data Types and Classes

The notion of an abstract data type is quite simple. An abstract data type is a set of objects and the operations on those objects. These are bound together so that one can pass an object from one part of a program to another, and in doing so provide access not only to the data attributes of the object but also to operations that make it easy to manipulate that data.

The specifications of those operations define an interface between the abstract data type and the rest of the program. The interface defines the behavior of the operations—what they do, but not how they do it. The interface thus provides an abstraction barrier that isolates the rest of the program from the data structures, algorithms, and code involved in providing a realization of the type abstraction.

Programming is about managing complexity in a way that facilitates change. There are two powerful mechanisms available for accomplishing this: decomposition and abstraction. Decomposition creates structure in a program, and abstraction suppresses detail. The key is to suppress the appropriate details. This is where data abstraction hits the mark.

One can create domain specific types that provide a convenient abstraction. Ideally, these types capture concepts that will be relevant over the lifetime of a program. If one starts the programming process by devising types that will be relevant months and even decades later, one has a great leg up in maintaining that software.

In Python, one implements data abstractions using classes. Contains a class definition that provides a straightforward implementation.

Example:

```
import datetime
class Person(object):
    def __init__(self, name):
        self.name = name
        try:
            lastBlank = name.rindex(' ')
            self.lastName = name[lastBlank+1:]
```

```
        except:
            self.lastName = name
            self.birthday = None
    def getName(self):
        return self.name
    def getLastName(self):
        return self.lastName

    def setBirthday(self, birthdate):
        self.birthday = birthdate

    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return (datetime.date.today() - self.birthday).days

    def __lt__(self, other):
        if self.lastName == other.lastName:
            return self.name < other.name
        return self.lastName < other.lastName

    def __str__(self):
        return self.name
p1 = Person('Ashvin Gami')
p2 = Person('Ketan Parbhulal Detroja')
print ('Display last name',p1.getLastName())
p2.setBirthday(datetime.date(1996, 10, 17))
print (p2.getName(), 'is', p2.getAge(), 'days old')
```

❖ Classes

Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects. Object is simply a collection of data (variables) and methods (functions) that act on those data. And, class is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object. As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

Defining a Class

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    Block of statement
```

Example:

```
class MyClass:
    "This is my second class"
    a = 10
    def func(self):
        print('Hello')

ob = MyClass()
print('Access By class Name=',MyClass.a)
print('Access By Object Name=',ob.a)
print(ob.func)
 print(MyClass.__doc__)
```

Constructors

Class functions that begin with double underscore (__) are called special functions as they have special meaning. Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Example:

```
class ComplexNumber:
   def __init__(self,r = 0,i = 0):
       self.real = r
       self.imag = i
   def getData(self):
       print("{0}+{1}j".format(self.real,self.imag))

c1 = ComplexNumber(2,3)
c1.getData()
c2 = ComplexNumber(5)
c2.attr = 10
print((c2.real, c2.imag, c2.attr))
```

❖ Inheritance

Many types have properties in common with other types. For example, types list and str each have len functions that mean the same thing. Inheritance provides a convenient mechanism for building groups of related abstractions. It allows programmers to create a type hierarchy in which each type inherits attributes from the types above it in the hierarchy.

The class object is at the top of the hierarchy. This makes sense, since in Python everything that exists at runtime is an object. Because Person inherits all of the properties of objects, programs can bind a variable to a **Person**, append a Person to a list, etc.

The class **MITPerson** in below inherits attributes from its parent class, Person, including all of the attributes that Person inherited from its parent class, object.

Example:

```
class MITPerson(Person):
    nextIdNum = 0

    def __init__(self, name):
        Person.__init__(self, name)
        self.idNum = MITPerson.nextIdNum
        MITPerson.nextIdNum += 1
    def getIdNum(self):
        return self.idNum
    def __lt__(self, other):
        return self.idNum < other.idNum

p1 = MITPerson('Ashvin Gami')
p2 = MITPerson('Ketan Parbhulal Detroja')
p3 = MITPerson('Billy Bob Beaver')
p4 = Person('Billy Bob Beaver')

print ('p1 < p2 =', p1 < p2)
print ('p3 < p2 =', p3 < p2)
print ('p4 < p1 =', p4 < p2)
```

In the jargon of object-oriented programming, **MITPerson** is a subclass of **Person**, and therefore inherits the attributes of its superclass. In addition to what it inherits, the subclass can:

• **Add new attributes**. For example, MITPerson has added the class variable nextIdNum, the instance variable idNum, and the method getIdNum.

• **Override attributes of the superclass.** For example, MITPerson has overridden __init__ and __lt__.

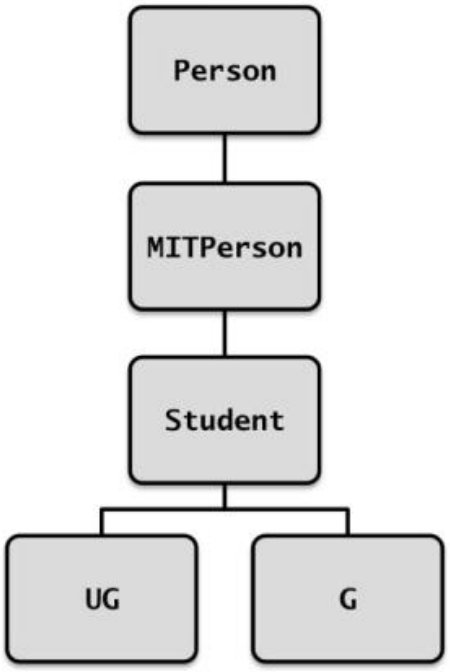
The method MITPerson.__init__ first invokes Person.__init__ to initialize the inherited instance variable self.name. It then initializes self.idNum, an instance variable that instances of MITPerson have but instances of Person do not.

Multiple Levels of Inheritance

Adding UG seems logical, because we want to associate a year of graduation (or perhaps anticipated graduation) with each undergraduate. But what is going on with the classes Student and Grad? By using the Python reserved word pass as the body, we indicate that the class has no attributes other than those inherited from its superclass.

By introducing the class Grad, we gain the ability to create two different kinds of student and use their types to distinguish one kind of object from another.

```
class Student(MITPerson):
    pass
class UG(Student):
    def __init__(self, name, classYear):
        MITPerson.__init__(self, name)
        self.year = classYear
    def getClass(self):
         return self.year
class Grad(Student):
   pass
p5 = Grad('Dixit Patel')
p6 = UG('Hardik Dalsaniya', 2000)
print (p5, 'is a graduate student is', type(p5) == Grad)
print (p5, 'is an UG student is', type(p5) == UG)
print (p6, 'is a graduate student is', type(p5) == Grad)
print (p6, 'is an UG student is', type(p5) == UG)
```



❖ Encapsulation and information hiding

Encapsulation is the packing of data and functions operating on that data into a single component and restricting the access to some of the object's components.

Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables etc.

Difference between Abstraction and Encapsulation
Abstraction is a mechanism which represents the essential features without including implementation details.

Encapsulation:—Information hiding.
Abstraction:—Implementation hiding.

Encapsulation:

Python follows the philosophy of we're all adults here with respect to hiding attributes and methods; i.e. you should trust the other programmers who will use your classes. Use plain attributes whenever possible.

You might be tempted to use getter and setter methods instead of attributes, but the only reason to use getters and setters is so you can change the implementation later if you need to. However, Python 2.2 and later allows you to do this with properties:

Protected members:

Protected member is (in C++ and Java) accessible only from within the class and its subclasses. How to accomplish this in Python? The answer is—by convention. By prefixing the name of your member with a single underscore, you're telling others "don't touch this, unless you're a subclass". See the example below:

Private members:

But there is a method in Python to define Private: Add "__" (double underscore) in front of the variable and function name can hide them when accessing them from out of class.

Python doesn't have real private methods, so one underline in the beginning of a method or attribute means you shouldn't access this method. But this is just convention. I can still access the variables with single underscore.

Also when using double underscore (__).we can still access the private variables

An example of accessing private member data. (Using name mangling)

```
class Person:
  def __init__(self):
    self.name = 'Ashvin'
    self.__lastname = 'Gami'

  def PrintName(self):
    return self.name +' ' + self.__lastname

#Outside class
P = Person()
print(P.name)
print(P.PrintName())
print(P.__lastname)
```

#AttributeError: 'Person' object has no attribute '__lastname'

Note: The __init__ method is a constructor and runs as soon as an object of a class is instantiated.

Its aim is to initialize the object

- Access public variable out of class, succeed
- Access private variable our of class, fail
- Access public function but this function access Private variable __B successfully since they are in the same class.

An example of accessing private member data. (Using name mangling technique)

```
class SeeMee:
  def youcanseeme(self):
    return 'you can see me'

  def __youcannotseeme(self):
    return 'you cannot see me'

#Outside class
Check = SeeMee()
print(Check.youcanseeme())
# you can see me
print(Check.__youcannotseeme())
```

#AttributeError: 'SeeMee' object has no attribute '__youcannotseeme' If you need to access the private member function

```
class SeeMee:
  def youcanseeme(self):
    return 'you can see me'
```

```
    def __youcannotseeme(self):
      return 'you cannot see me'

#Outside class
Check = SeeMee()
print(Check.youcanseeme())
print(Check._SeeMee__youcannotseeme())
#Changing the name causes it to access the function
```

You can still call the method using its mangled name, so this feature doesn't provide much protection.
You should know the following for accessing private members and private functions:

1. When you write to an attribute of an object that does not exist, the python system will normally not complain, but just create a new attribute.
2. Private attributes are not protected by the Python system. That is by design decision.
3. Private attributes will be masked. The reason is, that there should be no clashes in the inheritance chain. The masking is done by some implicit renaming. Private attributes will have the real name

"__<className>_<attributeName>"

With that name, it can be accessed from outside. When accessed from inside the class, the name will be automatically changed correctly.

❖ Sorting Search Algorithms

We have learned that in order to write a computer program which performs some task we must construct a suitable algorithm. However, whatever algorithm we construct is unlikely to be unique – there are likely to be many possible algorithms which can perform the same task. Are some of these algorithms in some sense better than others? Algorithm analysis is the study of this question.

We will analysis four algorithms; two for each of the following common tasks:

- **sorting*:*** ordering a list of values
- **searching*:*** finding the position of a value within a list

Algorithm analysis should begin with a clear statement of the task to be performed. This allows us both to check that the algorithm is correct and to ensure that the algorithms we are comparing perform the same task.

Although there are many ways that algorithms can be compared, we will focus on two that are of primary importance to many data processing algorithms:

- **time complexity*:*** how the number of steps required depends on the size of the input
- **space complexity*:*** how the amount of extra memory or storage required depends on the size of the input

Note: Common sorting and searching algorithms are widely implemented and already available for most programming languages.

❖ Sorting algorithms

The sorting of a list of values is a common computational task which has been studied extensively. The classic description of the task is as follows:
Given a *list of values* and a function that *compares two values*, order the values in the list from smallest to largest.

The values might be integers, or strings or even other kinds of objects. We will examine two algorithms:

- ***Selection sort***, which relies on repeated *selection* of the next smallest item
- ***Merge sort***, which relies on repeated *merging* of sections of the list that are already sorted

Other well-known algorithms for sorting lists are

- *Insertion sort*,
- *bubble sort*,
- *heap sort*,
- *quick sort*
- *Shell sort*.

There are also various algorithms which perform the sorting task for restricted kinds of values, for example:

- ***Counting sort***, which relies on the values belonging to a small set of items
- ***Bucket sort,*** which relies on the ability to map each value to one of a small set of items
- ***Radix sort,*** which relies on the values being sequences of digits

If we restrict the task, we can enlarge the set of algorithms that can perform it. Among these new algorithms may be ones that have desirable properties. For example, *Radix sort* uses fewer steps than any generic sorting algorithm.

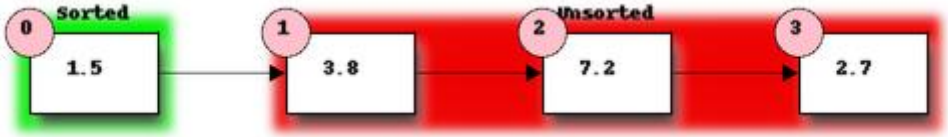
➢ Selection sort

To order a given list using selection sort, we repeatedly select the smallest remaining element and move it to the end of a growing sorted list. To illustrate selection sort, let us examine how it operates on a small list of four elements:

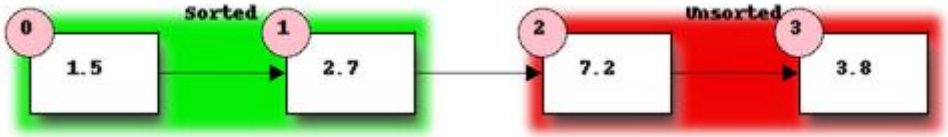
Initially the entire list is unsorted. We will use the front of the list to hold the sorted items – to avoid using extra storage space – but at the start this sorted list is empty.

First we must find the smallest element in the unsorted portion of the list. We take the first element of the unsorted list as a candidate and compare it to each of the following elements in turn, replacing our candidate with any element found to be smaller. This requires 3 comparisons and we find that element 1.5 at position 2 is smallest.

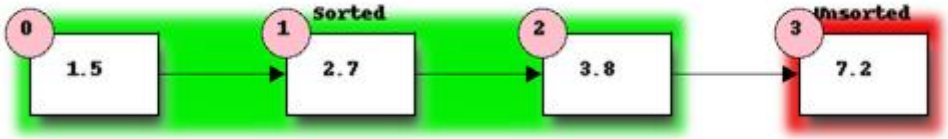
Now we will swap the first element of our unordered list with the smallest element. This becomes the start of our ordered list:



We now repeat our previous steps, determining that 2.7 is the smallest remaining element and swapping it with 3.8 – the first element of the current unordered section – to get:



Finally, we determine that 3.8 is the smallest of the remaining unordered elements and swap it with 7.2:



The table below shows the number of operations of each type used in sorting our example list:

Note that the number of *comparisons* and the number of *swaps* are independent of the contents of the list (this is true for selection sort but not necessarily for other sorting algorithms) while the number of times we have to assign a new value to the smallest candidate depends on the contents of the list.

The algorithm for selection sort is as follows:

1. Divide the list to be sorted into a sorted portion at the front (initially empty) and an unsorted portion at the end (initially the whole list).

2. Find the smallest element in the unsorted list:
   I. Select the first element of the unsorted list as the initial candidate.
   II. Compare the candidate to each element of the unsorted list in turn, replacing the candidate with the current element if the current element is smaller.
   III. Once the end of the unsorted list is reached, the candidate is the smallest element.
   IV. Swap the smallest element found in the previous step with the first element in the unsorted list, thus extending the sorted list by one element.

3. Repeat the steps 2 and 3 above until only one element remains in the unsorted list.

Example:

```
def selection_sort(items):
    i = 0
    while i<len(items):
       #smallest element in the sublist
        smallest = min(items[i:])
        #index of smallest element
        index_of_smallest = items.index(smallest)
       #swapping
        items[i], items[index_of_smallest] = items[index_of_smallest], items[i]
        i=i+1
    print (items)


a=[]
print("Enter size of  Array = ")
n = int(input())
for i in range(0,n):
    print("Enter Number = ")
    c=int(input())
    a.append(c)
print("\n.....Unsort Array values.....")
print(a)
print("\n.....Sort Array values.....")
selection_sort(a)
```

Note

The *Selection sort* algorithm as described here has two properties which are often desirable in sorting algorithms.

The first is that the algorithm is *in-place*. This means that it uses essentially no extra storage beyond that required for the input (the unsorted list in this case). A little extra storage may be used (for example, a temporary variable to hold the candidate for the smallest element). The important property is that the extra storage required should not increase as the size of the input increases.

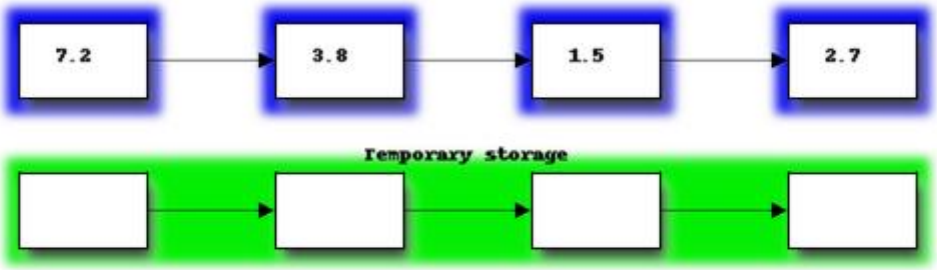
The second is that the sorting algorithm is *stable*. This means that two elements which are equal retain their initial relative ordering. This becomes important if there is additional information attached to the values being sorted (for example, if we are sorting a list of people using a comparison function that compares their dates of birth). Stable sorting algorithms ensure that sorting an already sorted list leaves the order of the list unchanged, even in the presence of elements that are treated as equal by the comparison.

- **Merge sort**

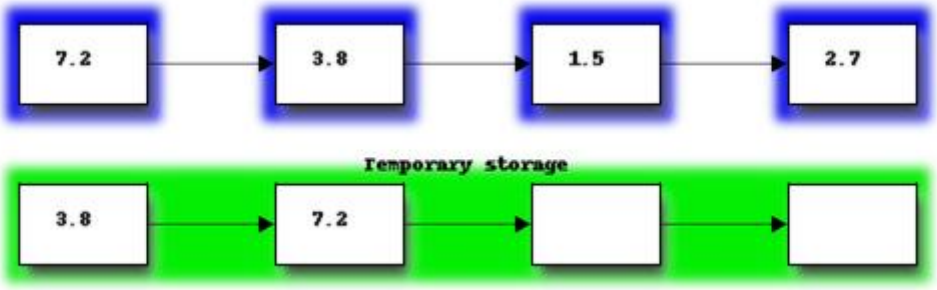
When we use merge sort to order a list, we repeatedly merge sorted sub-sections of the list – starting from sub-sections consisting of a single item each.

We will see shortly that merge sort requires significantly fewer operations than selection sort.

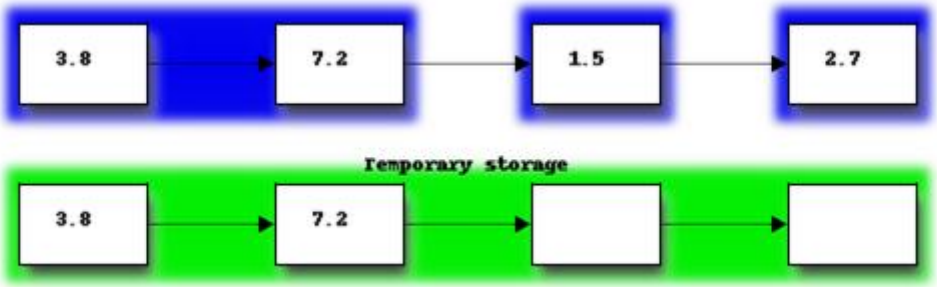
Let us start once more with our small list of four elements:



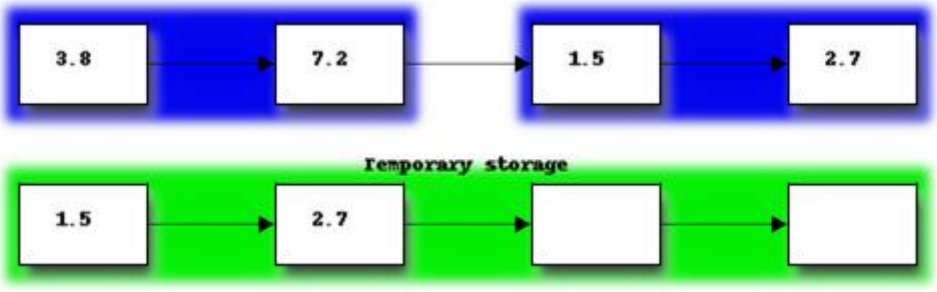
First we will merge the two sections on the left into the temporary storage. Imagine the two sections as two sorted piles of cards – we will merge the two piles by repeatedly taking the smaller of the top two cards and placing it at the end of the merged list in the temporary storage.



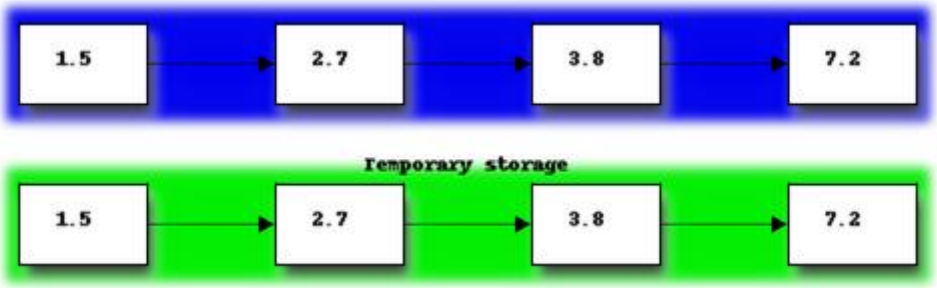
Next we copy the merged list from the temporary storage back into the portion of the list originally occupied by the merged subsections:



We repeat the procedure to merge the second pair of sorted sub-sections:



Having reached the end of the original list, we now return to the start of the list and begin to merge sorted sub-sections again. We repeat this until the entire list is a single sorted sub-section. In our example, this requires just one more merge:



Notice how the size of the sorted sections of the list doubles after every iteration of merges. After M steps the size of the sorted sections is 2^M. Once 2^M is greater than N, the entire list is sorted. Thus, for a list of size N, we need M equals' $\log_2 N$ interactions to sort the list.

Each iteration of merges requires a complete pass through the list and each element is copied twice – once into the temporary storage and once back into the original list. As long as there are items left in both sub-sections in each pair, each copy into the temporary list also requires a comparison to pick which item to copy.

The total number of operations required for our merge sort algorithm is the product of the number of operations in each pass and the number of passes – i.e. $2N\log_2 N$ copies and roughly $N\log_2 N$ comparisons.

The algorithm for merge sort may be written as this list of steps:

1. Create a temporary storage list which is the same size as the list to be sorted.
2. Start by treating each element of the list as a sorted one-element sub-section of the original list.
3. Move through all the sorted sub-sections, merging adjacent pairs as follows:
    1. Use two variables to point to the indices of the smallest uncopied items in the two sorted sub-sections, and a third variable to point to the index of the start of the temporary storage.
    2. Copy the smaller of the two indexed items into the indicated position in the temporary storage. Increment the index of the sub-section from which the item was copied, and the index into temporary storage.
    3. If all the items in one sub-section have been copied, copy the items remaining in the other sub-section to the back of the list in temporary storage. Otherwise return to step 3 ii.
    4. Copy the sorted list in temporary storage back over the section of the original list which was occupied by the two sub-sections that have just been merged.
4. If only a single sorted sub-section remains, the entire list is sorted and we are done. Otherwise return to the start of step 3.

```
def mergeSort(alist):
    #print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    #print("Merging ",alist)

a=[]
print("Enter size of  Array = ")
n = int(input())
for i in range(0,n):
     print("Enter Number = ")
     c=int(input())
     a.append(c)
print("\n.....Unsort Array values.....")
print(a)
mergeSort(a)
print("\n.....Sort Array values.....")
print(a)
```

➢ Searching algorithms

Searching is also a common and well-studied task. This task can be described formally as follows: Given a *list of values*, a function that *compares two values* and a *desired value*, find the position of the desired value in the list.

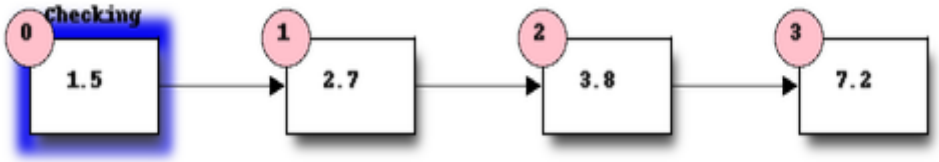
We will look at two algorithms that perform this task:

- **linear search**, which simply checks the values in sequence until the desired value is found
- **binary search**, which requires a sorted input list, and checks for the value in the middle of the list, repeatedly discarding the half of the list which contains values which are definitely either all larger or all smaller than the desired value

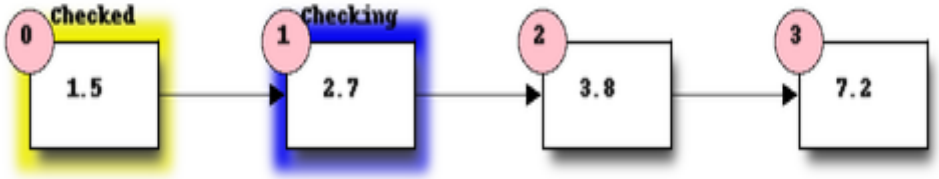
There are numerous other searching techniques. Often they rely on the construction of more complex data structures to facilitate repeated searching. Examples of such structures are *hash tables* (such as Python's dictionaries) and *prefix trees*. Inexact searches that find elements similar to the one being searched for are also an important topic.

❖ Linear search

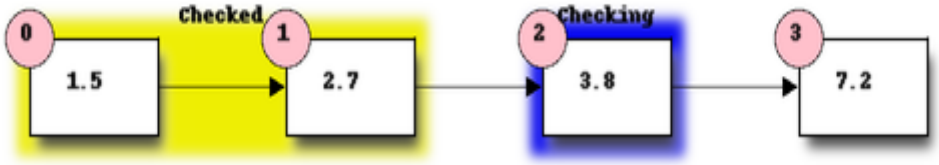
Linear search is the most basic kind of search method. It involves checking each element of the list in turn, until the desired element is found. For example, suppose that we want to find the number 3.8 in the following list:



We start with the first element, and perform a comparison to see if its value is the value that we want. In this case, 1.5 is not equal to 3.8, so we move onto the next element:



We perform another comparison, and see that 2.7 is also not equal to 3.8, so we move onto the next element:



We perform another comparison and determine that we have found the correct element. Now we can end the search and return the position of the element (index 2).

We had to use a total of 3 comparisons when searching through this list of 4 elements. How many comparisons we need to perform depends on the total length of the list, but also whether the element we are looking for is near the beginning or near the end of the list. In the

worst-case scenario, if our element is the last element of the list, we will have to search through the entire list to find it.

If we search the same list many times, assuming that all elements are equally likely to be searched for, we will on average have to search through half of the list each time. The cost (in comparisons) of performing linear search thus scales linearly with the length of the list.

Example:

```
def linear_search(items):
    x = int(input("Enter value to search: "))
    found = False
    for i in range(len(items)):
        if(items[i] == x):
            found = True
            print("%d Value Found at %dth position"%(x,i))
            break
    if(found == False):
        print("%d is not in list"%x)



a=[]
n = int(input("Enter size of List = "))
for i in range(0,n):
    c=int(input("Enter List Value = "))
    a.append(c)
print("\n.....Display list values.....")
print(a)
linear_search(a)
```

❖ Binary search

Binary search is a more efficient search algorithm which relies on the elements in the list being sorted. We apply the same search process to progressively smaller sub-lists of the original list, starting with the whole list and approximately halving the search area every time.

We first check the *middle* element in the list.

- If it is the value we want, we can stop.
- If it is *higher* than the value we want, we repeat the search process with the portion of the list *before* the middle element.
- If it is *lower* than the value we want, we repeat the search process with the portion of the list *after* the middle element.

For example, suppose that we want to find the value 3.8 in the following list of 7 elements:

First we compare the element in the middle of the list to our value. 7.2 is *bigger* than 3.8, so we need to check the first half of the list next.

Now the first half of the list is our new list to search. We compare the element in the middle of this list to our value. 2.7 is *smaller* than 3.8, so we need to search the *second half* of this sub list next.

The second half of the last sub-list is just a single element, which is also the middle element. We compare this element to our value, and it is the element that we want.

We have performed 3 comparisons in total when searching this list of 7 items. The number of comparisons we need to perform scales with the size of the list, but much more slowly than for linear search – if we are searching a list of length N, the maximum number of comparisons that we will have to perform is $\log_2 N$.

Example:

```
def binary_sort(datalist,n,x):
    start = 0
    end = n - 1
    while(start <= end):
        mid = int((start + end)/2)
        if (x == datalist[mid]):
            return mid
        elif(x < datalist[mid]):
            end = mid - 1
        else:
            start = mid + 1
    return -1

datalist = []
n = int(input("Enter the size of the list: "))
```

```
for i in range(0,n):
    datalist.append(int(input("Enter %dth element: "%i)))

x = int(input("Enter the number to search: "))
position = binary_sort(datalist, n, x)

if(position != -1):
    print("Entered number %d is present at position: %d"%(x,position))
else:
    print("Entered number %d is not present in the list"%x)
```

❖ Algorithm complexity and Big O notation

We commonly express the cost of an algorithm as a function of the number N of elements that the algorithm acts on. The function gives us an estimate of the number of operations we have to perform in order to use the algorithm on N elements – it thus allows us to predict how the number of required operations will increase as N increases. We use a function which is an *approximation* of the exact function – we simplify it as much as possible, so that only the most important information is preserved.

For example, we know that when we use linear search on a list of N elements, on average we will have to search through half of the list before we find our item – so the number of operations we will have to perform is N/2. However, the most important thing is that the algorithm scales *linearly* – as N increases, the cost of the algorithm increases in proportion to N, not N^2 or N^3. The constant factor of 1/2 is insignificant compared to the very large differences in cost between – for example – N and N^2, so we leave it out when we describe the cost of the algorithm.

We thus write the cost of the linear search algorithm as O(N) – we say that the cost is *on the order of N*, or just *order N*. We call this notation *big O notation*, because it uses the capital O symbol (for *order*). We have dropped the constant factor 1/2. We would also drop any lower-order terms from an expression with multiple terms – for example, $O(N^3 + N^2)$ would be simplified to $O(N^3)$.

In the example above we calculated the *average* cost of the algorithm, which is also known as the *expected* cost, but it can also be useful to calculate the *best case* and *worst case* costs. Here are the best cases, expected and worst case costs for the sorting and searching algorithms we have discussed so far:

Algorithm	**Best case**	**Expected**	**Worst case**
Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Merge sort	O(N log N)	O(N log N)	O(N log N)
Linear search	O(1)	O(N)	O(N)
Binary search	O(1)	O(log N)	O(log N)

What does O(1) mean? It means that the cost of an algorithm is *constant*, no matter what the value of N is. For both these search algorithms, the best case scenario happens when the first element to be tested is the correct element – then we only have to perform a single operation to find it.

In the previous table, big O notation has been used to describe the *time complexity* of algorithms. It can also be used to describe their *space complexity* – in which case the cost function represents the number of units of space required for storage rather than the required number of operations. Here are the space complexities of the algorithms above (for the worst case, and excluding the space required to store the input):

Algorithm	**Space complexity**
Selection sort	O(1)
Merge sort	O(N)
Linear search	O(1)
Binary search	O(1)

None of these algorithms require a significant amount of storage space in addition to that used by the input list, except for the merge sort – which, as we saw in a previous section, requires temporary storage which is the same size as the input (and thus scales linearly with the input size).

❖ Hash tables

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words Hash table stores key-value pairs but the key is generated through a hashing function.

So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.

- The keys of the dictionary are hash table i.e. they are generated by hashing function which generates unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.

So we see the implementation of hash table by using the dictionary data types as below.

Example:

```
# Declare a dictionary
dict = {'Name': 'Ashvin', 'Age': 30, 'Class': 'First'}

# Accessing the dictionary with its key
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])

dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])

del dict['Name']; # remove entry with key 'Name'
#dict.clear();     # remove all entries in dict
#del dict ;        # delete entire dictionary

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```