## PYTHON EXCEPTIONS

- An Exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- An "exception" is usually defined as "something that does not conform to the norm".
- Python provides two very important features to handle any unexpected error in our Python programs and to add debugging capabilities in them: Exception Handling and Assertions.

## Handling Exceptions

- Up to now, 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. Exception, when raised, can and should be *handled* by the program.
- If we have some *suspicious* code that may raise an exception, we can defend our program by placing the suspicious code in a *try*: block. After the try: block, include an *except:* statement, followed by a block of code which handles the problem as elegantly as possible. We can also have the e*xcept* clause with multiple exceptions separated by comma and the *finally* clause which must be executed though exception is generated.

**Syntax:**

```
try:
        Code
except Exception 1:
        If there is Exception 1, then execute this block.
except Exception 2:
        If there is Exception 2, then execute this block.
……
else:
        If there is no exception then execute this block.
```

- Few points about the above – mentioned syntax:
    - A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
    - You can also provide a generic except clause, which handles any exception.
    - After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
    - The else-block is a good place for code that does not need the try: block's protection.

**Example:**

```
n = 10
div = 0
try:
    z = n / div
except:
    print('Division Error')
```

- List of Standard Exceptions with their 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. |
| ZeroDivisonError | 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. |

**Example:**

```
n = 15
>>> try:
          print (n/0) #Try with value other than 0
except ZeroDivisionError:
          print('Error in Division')
else:
          print('Perfect Statement')

Error in Division   #Output
```

## Exceptions as a Control Flow Mechanism

- Don't think of exceptions as purely for errors. They are a convenient flow-of-control 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 / similar to Python's None) indicating that something has gone amiss / incorrect.
- 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.
- The *raise* statement does two things: it creates an *exception* object, and immediately leaves the expected program execution sequence to search the enclosing *try* statements for a matching *except* clause. The effect of a *raise* statement is to either divert execution in a matching *except* suite, or to stop the program because no matching *except* suite was found to handle the exception.
- The *exception* object created by *raise* can contain a message string that provides a meaningful error message.
- 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.

***Using Exceptions for Control Flow***

```
>>> def getRatios(v1, v2):
 """Assumes: v1 and v2 are lists of equal length of nos.
     Returns: a list containing the meaningful values of
              v1[i]/v2[i]"""
          ratios = []
          for index in range(len(v1)):
               try:
                  ratios.append(v1[index] / float(v2[index]))
          except ZeroDivisionError:
               ratios.append(float('nan'))
          except:
               raise ValueError('Called with bad arguments')
     return ratios

>>> try:
         print(getRatios([1.0,2.0,7.0,6.0],[1.0,2.0,0.0,3.0]))
         print(getRatios([],[]))
         print(getRatios([1.0,2.0],[3.0]))
except ValueError as msg:
           print(msg)

#Output
          [1.0, 1.0, nan, 2.0]
          []
          Called with bad arguments
```

- There are two except blocks associated with the try block. If an exception is raised within the try block, Python first checks to see if it is a ZeroDivisionError. If so, it appends a special value, nan, of type float to ratios. (The value nan stands for “not a number.” There is no literal for it, but it can be denoted by converting the string 'nan' or the string 'NaN' to type float. When nan is used as an operand in an expression of type float, the value of that expression is also nan.) If the exception is

anything other than a ZeroDivisionError, the code executes the second except block, which raises a ValueError exception with an associated string.

***Control Flow without a try - except***

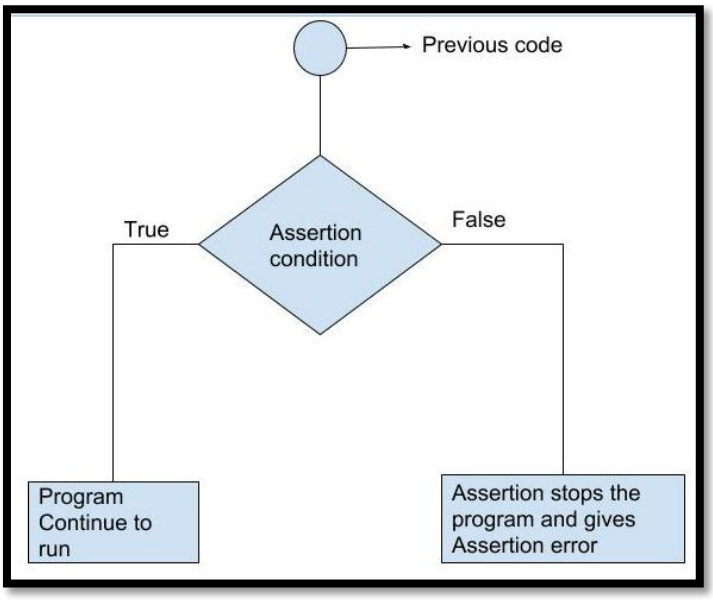
```
>>> def getRatios(v1, v2):
        ratios=[]
        if len(v1) != len(v2):
            raise ValueError('Called with bad arguments')
        for index in range(len(v1)):
            v1e = v1[index]
            v2e = v2[index]
            if(type(v1e) not in (int, float)) or (type(v2e)
                not in (int, float)):
                raise ValueError('Wrong Type Passed')
            if v2e == 0.0:
                ratios.append(float('NaN'))
            else:
                ratios.append(v1e/v2e)
    return ratios

>>> try:
        print(getRatios([1.0,2.0,7.0,6.0],[1.0,2.0,0.0,3.0]))
        print(getRatios([],[]))
        print(getRatios([1.0,2.0],[3.0]))
 except ValueError as msg:
          print(msg)

    #Output
        [1.0, 1.0, nan, 2.0]
        []
        Called with bad arguments
```

## Assertions

- The Python *assert* statement provides programmers with a simple way to confirm that the state of the computation is as expected.
- Assertions are statements that assert or state a fact confidently in our program. For example, while writing a division function, our confident the divisor shouldn't be zero, our assert divisor is not equal to zero.
- Assertions are simply Boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and moves to the next line of code. However, if it's false, the program stops and throws an error.
- Flowchart of Assertion:




- Python has built-in *assert* statement to use assertion condition in the program. *assert* statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an *AssertionError.*

```
Syntax:
    assert <condition>
    assert <condition>, <error message>
```

- In Python we can use assert statement in two ways as mentioned above.
    - *assert* statement has a condition and if the condition is not satisfied the program will stop and give *AssertionError*.

- *assert* statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives *AssertionError* along with the error message

**Example - 1: Using assert without Error Message**

```
>>> def avg(marks):
        assert (len(marks) != 0)
        return sum(marks) / len(marks)

>>> mark1=[]
>>> print(avg(mark1))

#Output
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    print(avg(mark1))
  File "<pyshell#65>", line 2, in avg
    assert (len(marks) != 0)
AssertionError
```

- We got an error as we passed an empty list mark1 to *assert* statement, the condition became false and assert stops the program and give *AssertionError*.

**Example - 2: Using assert with Error Message**

```
>>> def avg(marks):
        assert len(marks) != 0, "List is empty"
        return sum(marks) / len(marks)

>>> mark1=[55,78,87]
>>> print(avg(mark1))
73.3333333333
```

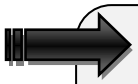
```
>>> mark1=[]
>>> print(avg(mark1))
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    print(avg(mark1))
  File "<pyshell#76>", line 2, in avg
    assert len(marks) != 0,"List is empty"
AssertionError: List is empty
```

**:: Points to Remember for Assertions ::**

- Assertions are the condition or Boolean expression which are always supposed to be true in the code.
- *assert* statement takes an expression and optional message.
- *assert* statement is used to check types, values of argument and the output of the function.
- *assert* statement is used as debugging tool as it halts the program at the point where an error occurs.

## Abstract Data Types and Classes

- Classes can be used in many different ways. Python supports OOP concept.
- The key to OOP is thinking about objects as collections of both data and methods.
- OOP is about 50 years old. In mid 1970s people began to write articles explaining the benefits of OOP in programming.
- Objects are the core things that Python programs manipulate.
- Every object has a type that defines the kinds of things that programs can do with objects of that type.
- Python has been an Object-Oriented language since it existed.



**OOP Terminologies to keep in mind:**

Class, Class Variable, Data Member, Function Overloading, Inheritance, Instance, Method, Operator Overloading

- An *abstract data type* is a set of objects and the operations on those objects.
- In Python we are using ADTs in our program like integers, lists, strings, dictionaries etc.
- Data abstraction in Python can be implemented using *classes*.
- Everything in Python is a class.

```
>>> x = 42
>>> type(x)
<class 'int'>

>>> y = 4.34
>>> type(y)
<class 'float'>

>>> def f(x):
        return x + 1

>>> type(f)
<class 'function'>

>>> import math
>>> type(math)
<class 'module'>
```

- The *class* statement creates a new class definition. The name of the class is immediately follows the keyword *class* followed by a colon.

```
Syntax:   class class_name:
               ''docstring''
               class_suite

Example:  >>>class Robot:
                    pass
```

- The body of class consists of an indented block of statements.
- A class object is created, when the definition is left normally, i.e. via the end. Above is example of Python class (Only three words and two lines of code.)

```
Example:
    >>> class Robot:
            name = ''
            def __init__(self, name=None):
                    self.name = name
                    print(self.name)
            def printme(self):
                      print('Hello! I am Method')

Check followings for output
>>> x = Robot()
>>> x = Robot('rpbc')
>>> print(x.name)
>>> x.printme()
```

- In above example:
    - we have class Robot with attribute name.
    - The __init__ method is a method which is immediately and automatically called after an instance has been created.
    - The name of this method ( __init__ ) is fixed and it is not possible to choose another name.
    - Python has a number of special method names that start and end with two underscores. The __init__ is one of the so-called *magic methods*.
    - The __init__ method can be anywhere in a class definition, but usually it is the first method of a class.
    - *self* is not a keyword in Python, but it's just a strong convention. Python decided to do methods in a way that makes the instance to which the method belongs be *passed* automatically. The first parameter of Python Class Method is the instance of the method is called on. That makes methods entirely the same as functions.
- When any variable is declared / defined inside class body (i.e. outside of any method), then it is sometimes referred to as *data attribute* of the class. (In above example, name).

- When a function definition occurs within a class definition, the defined function is called a *method* and is associated with the class. These methods are sometimes referred to as *method attributes* of the class. (In above example, printme()).
- Classes support two kinds of operation:
    - Instantiation: Used to create instances of the class. For Example, x = Robot() creates a new object of type *Robot*. This object is called an instance of Robot().
    - Attribute References: Uses dot notation to access attributes associated with the class. For Example, *x.name* refers to the member associated with the instance *x* of type *Robot*.

**Example:**

```
>>> class test(object):
          def __init__(self):
                    self.vals=[]
          def insert(self,e):
               if not e in self.vals:
                         self.vals.append(e)
          def member(self,e):
               return e in self.vals
          def remove(self,e):
               self.vals.remove(e)
          def getMembers(self):
               return self.vals[:]
          def __str__(self):
               self.vals.sort()
               result=''
               for e in self.vals:
                  result = result + str(e) +','
               return '{' + result[:-1] + '}'
```

## Inheritance

- Inheritance enables us to define a class that takes all the functionality from parent class and allows us to add more.
- Inheritance is a powerful feature in Object Oriented Programming.
- 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 hierarchy.
- Inheritance refers to defining a new class with little or no modification to an existing class. The new class is called *derived (or child) class* and the one from which it inherits is called the *base (or parent) class.*
- Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

**Syntax:**

```
class BaseClass:
        Body of base class
class DerivedClass(BaseClass):
        Body of derived class
```

**Example:**

```
>>> class Person:
        def __init__(self, first, last):
            self.fname= first
            self.lname= last

        def Name(self):
            return self.fname+ " " + self.lname


>>> class Employee(Person):
        def __init__(self, first, last, enum):
            Person.__init__(self,first, last)
            self.empno = enum

        def GetEmployee(self):
             return self.Name() + ", " +  self.empno

>>> x = Person('Guido','Rossum')
>>> y = Employee('Mike','Sherill','101')
```

- The __init__ method of our Employee class explicitly invokes the __init__ method of the Person class. We could have used super instead.

```
def __init__(self, first, last, enum):
      super().__init__(first, last)
      self.empno = enum
```

- We used super() without arguments. This is only possible in Python 3. We could have written "super(Employee,self).__init__(first, last).

## Encapsulation and Information Hiding

- There are two important concepts at the heart of OOP: Encapsulation and Information Hiding.
- *Encapsulation* means the bundling together of data attributes and the methods for operating on them.
- *Information Hiding:* Many programming languages (Java, C++) provide mechanisms for enforcing information hiding. Programmers can make the data attributes of a class invisible to clients of the class, and thus require that the data be accessed only through the object's methods. Unfortunately, Python does not provide mechanisms for enforcing information hiding. There is no way for the implementer of a class to restrict access to the attributes of class instances.
- In Python, an object's attributes may or may not be visible outside the class definition. We need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

```
>>> class test:
          __count = 0
          def hide_info(self):
                self.__count += 1
                print(self.__count)

>>> counter = test()
>>> counter.hide_info()
>>> counter.hide_info()
>>> print(counter.__count)       #Generates Error
'''Python protects this type of members by internally
   changing name to include the class name. To access
   this member (object._classname__attname) '''
>>> print(counter._test__count)
```

## Search Algorithms

- A *search algorithm* is a method for finding an item or group of items with specific properties within a collection of items (also known as *search space*.
- Searching is very basic necessity when we store data in different data structures.
- Linear Search and Using Indirection to Access Elements:
    - Python uses the following algorithm to determine if an element is in a list:

```
def search (L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

    - If the element *e* is not in the list, the algorithm will perform O(len(L)) tests, i.e. the complexity is at best linear in the length of L. This will be linear only if each operation inside the loop can be done in constant time.
    - For example, consider that we have

      list1 = [35, 4, 5, 29, 17, 58, 0]

    - The access to above list is always constant time, as we have all elements of type int.
    - To see how this constant time access works, let's consider int occupies 4 units of memory, and this units can be 8 bits, 16 bits. So how to access the $i^{th}$ elements of the list i.e. list1[i]?
    - The answer is, if we know the starting address of list1 then it will be

      *list1 + (4 * i)*

    - This is the way how traditional list (like an array) is implemented. It is the concept that each element will have a definite (exact / specific) size.
    - Python List is different as it contains heterogeneous elements (int, float, strings etc.). So, how would list be implemented in Python?
    - One of the very immediate and oldest ways of doing it is a *Linked List*. In a Linked List, every element has a data part and a pointer part which pointed to the next element. The end of the list is identified by a NULL element.

- This concept is called *Indirection (array of pointers)*. So we are same as the initial array of int concepts because each element of list is a pointer, so I can now access each element in constant time.
- *Indirection* is very powerful programming concepts but it is bad if we have too many indirections as the value stored using indirection is very far apart the memory.

- Linear Search:
  - In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data structures.

```
>>> def linear_search(values, search_key):
        i = 0
        found = False
        while i < len(values):
            if values[i] == search_key:
                found = True
                break
            else:
                i = i + 1
        return found

>>> list1 = [64, 12, 22, 52, 11, 90]

   # Check output of followings:
>>> linear_search(list1, 12)
>>> linear_search(list1, 101)
```

- Binary Search:
  - Also known as Interpolation Search.
  - This search algorithm works on the searching position of the required value.
  - For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

- Initially search position is the position of the middle most item of the collection.
- If a match occurs, then the index of the item is returned.
- If the middle item is greater than the item, then the search position is again calculated in sub – array to the right of the middle item. Otherwise the item is searched in the sub – array to the left of the middle item.
- This process continues on the sub – array as well until the size of sub – array reduces to zero.

**Example: - Binary Search**

```
>>> def intpolsearch(values,x):
      first = 0
      last = (len(values) - 1)
      found = False
      while first <= last and not found:
            mid = (last + first)//2

      if values[mid] == x:
          found = True
          return "Found "+str(x)+" at index  "+str(mid)

      if values[mid] < x:
          first = mid + 1
      else:
          last = mid - 1
      return "Searched element not in the list"
```

**Example: - Binary Search (Recursive)**

```
>>> def search(L, e):
        def bSearch(L, e, low, high):
            if high == low:
                return L[low] == e
            mid = (low + high)//2
            if L[mid] == e:
                return True
            elif L[mid] > e:
                if low == mid:
                    return False
                else:
                    return bSearch(L, e, low, mid - 1)
            else:
                return bSearch(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bSearch(L, e, 0, len(L) - 1)
```

## Sorting Algorithms

- Sorting refers to arranging data in a particular format.
- Sorting algorithm specifies the way to arrange data in particular order.
- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is a stored in a sorted manner.
- Sorting is also used to represent data in more readable formats.
- Python has following five implementations of sorting:
    - *Bubble Sort*: It is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

**Example of Bubble Sort**

```
>>> def bubblesort(list):
        for i in range(len(list)-1,0,-1):
                for j in range(i):
                        if list[j] > list[j+1]:
                                temp = list[j]
                                list[j] = list[j+1]
                                list[j+1] = temp
>>> list = [19,2,31,45,6,11,121,27]
>>> bubblesort(list)
>>> print(list)
```

- *Merge Sort*: This sort first divides the array into equal halves and then combines them into a sorted manner.

**Example of Merge Sort**

```
>>> def merge_sort(l1):
        if len(l1) <= 1:
                return l1
        middle = len(l1) // 2
        left_list = l1[:middle]
        right_list = l1[middle:]

        left_list = merge_sort(left_list)
        right_list = merge_sort(right_list)
        return list(merge(left_list, right_list))
```

```
>>> def merge(left_half,right_half):
        res = []
        while len(left_half) != 0 and len(right_half) != 0:
        if left_half[0] < right_half[0]:
                res.append(left_half[0])
                left_half.remove(left_half[0])
        else:
                res.append(right_half[0])
                right_half.remove(right_half[0])
        if len(left_half) == 0:
                res = res + right_half
        else:
                res = res + left_half
        return res

>>> l1 = [64,35,25,12,22,11,90]
>>> print(merge_sort(l1))
```

- *Insertion Sort*: Insertion sort involves finding the right place for a given element in a sorted list. So in beginning we compare the first two elements and sort them by comparing them. Then we pick the third element and find its proper position among the previous two sorted elements. This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

**Example of Insertion Sort**

```
>>> def insertion_sort(l1):
        for i in range(1, len(l1)):
            j = i - 1
            nxt_ele = l1[i]
            while (l1[j] > nxt_ele) and (j >= 0):
                l1[j+1] = l1[j]
                j = j - 1
                l1[j+1] = nxt_ele
```

```
>>> list = [196,2,63,21,31,45]
>>> insertion_sort(list)
>>> print(list)
[2, 21, 31, 45, 63, 196]
```

- *Selection Sort*: In this we start by finding the minimum value in a given list and move it to a sorted list. Then we repeat the process for each of the remaining elements in the unsorted list. The next element entering the sorted list is compared with the existing elements and placed at its correct position. So at the end all the elements from the unsorted list are sorted.

**Example of Selection Sort**

```
>>> def selection_sort(l1):
        for i in range(len(l1)):
            min_idx = i
            for j in range(i + 1, len(l1)):
                if l1[min_idx] > l1[j]:
                    min_idx = j
            l1[i],l1[min_idx] = l1[min_idx] ,l1[i]

>>> l = [19,2,31,45,30,11,121,27]
>>> selection_sort(l)
>>> print(l)
```

- *Shell Sort*: Shell sort involves sorting elements which are away from each other. We sort a large sublist of a given list and go on reducing the size of the list until all elements are sorted.

**Example of Shell Sort**

```
>>> def shell(seq):
        inc = len(seq) // 2
        while inc:
        for i, el in enumerate(seq):
            while i >= inc and seq[i - inc] > el:
                    seq[i] = seq[i - inc]
                    i -= inc
                    seq[i] = el
        inc = 1 if inc == 2 else int(inc * 5.0 / 11)

>>> data = [22,7,2,-5,8,4]
>>> shell(data)
>>> print(data)
[-5, 2, 4, 7, 8, 22]
```

## Hash tables

- Hash tables are 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.
- Hash Table is a data structure where data are stored in an associative manner (key, value format). The key / index is unique.
- Hash Table stores data into an array format. It uses a hashing function that generates a slot or an index to store / insert any element or value.
- With the use of Hash table, 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 followings:
    - The Keys of the dictionary are hash table i.e. they are generated by hashing function which generates unique result.
    - The order of data elements in a dictionary is not fixed.
- Simple Implementation of the hash table and hashing function:

```
''' Hash Function / Hashing

This deals with generating slot / index to any key value.

Perfect hashing / perfect hash function is the one which assigns a unique slot for every key value.

Sometimes, there can be cases where the hash function generates the same index for multiple key values. '''


#Creating hash table of size 10 with empty data.

>>> hash_table = [None] * 10
>>> print(hash_table)
#[None, None, None, None, None, None, None, None, None, None]
```

```
''' Simple hash function that returns the modulus of the length of the hash table. (In our Example, length is 5) Modulo Operator (%) is used in the hashing function. The % operator yields the remainder from the division of the first argument by the second. '''

>>> def hash_func(key):
          return key % len(hash_table)

>>> print(hash_func(10))      # Output: 0
>>> print(hash_func(25))      # Output: 5

''' Insert Data into Hash Table '''
Use of hash function to generate an index and store the given value into that slot.
>>> def insert(hash_table, key, value):
          hash_key = hash_func(key)
          hash_table[hash_key] = value
>>> insert(hash_table, 10, 'Good')
>>> print(hash_table)
 #[Good, None, None, None, None, None, None, None, None, None]

>>> insert(hash_table, 26, 'Logic')
>>> print(hash_table)
#[Good, None, None, None, None, None, Logic, None, None, None]
```

- Collision: A collision occurs when two items / values get the same slot / index, i.e. the hashing function generates same slot number for multiple items. If proper collision resolution steps are not taken then the previous item in slot will be replaced by the new item whenever the collision occurs.

```
Check following (Consider above example of Hash Table):
>>> insert(hash_table, 20, 'Best')
>>> print(hash_table)
#It will replace "Good" with "best".
```

- *Two ways to resolve a collision:*
- Linear Probing: One way to resolve collision is to find another open slot whenever there is a collision and store the item in that open slot. The search for open slot starts from the slot where the collision happened. It moves sequentially through the slots until an empty slot is encountered. The movement is in a circular fashion. It can move to the first slot while searching for an empty slot. This kind of sequential search is called *Linear Probing.*
- Chaining: Another way to resolve collision is *chaining*. This allows multiple items exist in the same slot / index. This can create a chain/collection of items in a single slot. When the collision happens, the item is stored in the same slot using chaining mechanism. To implement chaining in Python, we need to create the hash table as a nested list.

```
>>> hash_table = [[] for _ in range(10)]
>>> print(hash_table)
[[], [], [], [], [], [], [], [], [], []]

#same hash function as above…
>>> def hash_func(key):
     return key % len(hash_table)

#using append() in insert function
>>> def insert(hash_table, key, value):
    hash_key = hash_func(key)
    hash_table[hash_key].append(value)

>>> insert(hash_table,10,'Hello')
>>> print(hash_table)
[['Hello'], [], [], [], [], [], [], [], [], []]

>>> insert(hash_table, 10, 'Hi')
>>> print(hash_table)
[['Hello','Hi'], [], [], [], [], [], [], [], [], []]
```