```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

In the above program, we created a class with the name `Parrot`. Then, we define attributes. The attributes are a characteristic of an object.
These attributes are defined inside the `__init__` method of the class. It is the initializer method that is first run as soon as the object is created.
Then, we create instances of the `Parrot` class. Here, `blu` and `woo` are references (value) to our new objects.
We can access the class attribute using `__class__.species`. Class attributes are the same for all instances of a class. Similarly, we

access the instance attributes using `blu.name` and `blu.age`. However, instance attributes are different for every instance of a class.

Example 2 : Creating Methods in Python

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Example 3: Use of Inheritance in Python

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
```

```
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Problem Description

The program sorts a list by **bubble sort**.

Problem Solution

1. Create a function bubble_sort that takes a list as argument.
2. Inside the function create a loop with a loop variable i that counts from the length of the list – 1 to 1.
3. Create an inner loop with a loop variable that counts from 0 up to i – 1.
4. Inside the inner loop, if the elements at indexes j and j + 1 are out of order, then swap them.
5. If in one iteration of the inner loop there were no swaps, then the list is sorted and one can return prematurely.

Program/Source Code

```
def bubble_sort(alist):
    for i in range(len(alist) - 1, 0, -1):
```

```
        no_swap = True
        for j in range(0, i):
            if alist[j + 1] < alist[j]:
                alist[j], alist[j + 1] = alist[j + 1], alist[j]
                no_swap = False
        if no_swap:
            return
alist = input('Enter the list of numbers: ').split()
alist = [int(x) for x in alist]
bubble_sort(alist)
print('Sorted list: ', end='')
print(alist)
```

Problem Description

The program sorts a list by **selection sort**.

Problem Solution

1. Create a function selection_sort that takes a list as argument.
2. Inside the function create a loop with a loop variable i that counts from 0 to the length of the list – 1.
3. Create a variable smallest with initial value i.
4. Create an inner loop with a loop variable j that counts from i + 1 up to the length of the list – 1.
5. Inside the inner loop, if the elements at index j is smaller than the element at index smallest, then set smallest equal to j.
6. After the inner loop finishes, swap the elements at indexes i and smallest.

Program/Source Code

```
def selection sort(alist):
    for i in range(0, len(alist) - 1):
        smallest = i
        for j in range(i + 1, len(alist)):
            if alist[j] < alist[smallest]:
                smallest = j
        alist[i], alist[smallest] = alist[smallest], alist[i]


alist = input('Enter the list of numbers: ').split()
alist = [int(x) for x in alist]
selection sort(alist)
print('Sorted list: ', end='')
print(alist)
```

Merge Sort

```
def merge_sort(unsorted_list):
   if len(unsorted_list) <= 1:
      return unsorted_list
# Find the middle point and devide it
   middle = len(unsorted_list) // 2
   left_list = unsorted_list[:middle]
   right_list = unsorted_list[middle:]

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

# Merge the sorted halves
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
unsorted_list = [64, 34, 25, 12, 22, 11, 90]
print(merge_sort(unsorted_list))
```

Example 1: Using assert without Error Message

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

mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be:

```
AssertionError
```

Example 2: Using assert with error message

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

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be:

```
Average of mark2: 78.0
AssertionError: List is empty.
```

Example :

```
x = 0
assert x > 0, 'Only positive numbers are allowed'
print('x is a positive number.')
```

Name mangling with double underscores

If you prefix an attribute name with double underscores (__) like this:

__attribute

Python will automatically change the name of the __attribute to:

_class__attribute

This is called the name mangling in Python.

By doing this, you cannot access the __attribute directly from the outside of a class like:

instance.__attribute
Code language: CSS (css)

However, you still can access it using the _class__attribute name:

instance._class__attribute

Example :

```
class Counter:
    def __init__(self):
        self.__current = 0

    def increment(self):
        self.__current += 1

    def value(self):
        return self.__current

    def reset(self):
        self.__current = 0
```

Now, if you attempt to access __current attribute, you'll get an error:

```
counter = Counter()
print(counter.__current)
```

Output:

```
AttributeError: 'Counter' object has no attribute '__current'
```

However, you can access the __current attribute as _Counter___current like this:

```
counter = Counter()
print(counter._Counter__current)
```

Hash Tables :
Data requires a number of ways in which it can be stored and accessed. One of the most important implementations includes Hash Tables. In Python, these Hash tables are implemented through the built-in data type i.e, dictionary.
EXAMPLE:

```
my_dict={'Dave' : '001' , 'Ava': '002' , 'Joe': '003'}
                    print(my_dict)
                     type(my_dict)
```

OUTPUT:

{‘Dave’: ‘001’, ‘Ava’: ‘002’, ‘Joe’: ‘003’}
dict