PPS-Unit 4(DICTIONARIES)

Dictionaries:

Dictionaries store collection of items in the form of key-value pairs. Keys and values are for each item is separated with a colon ‘:’. Each element is separated from the other by a comma. The comma separated list of items is enclosed by braces. Dictionaries are mutable, which means that you can update or delete an item from a dictionary and can also add new items to a dictionary.

Creating a Dictionary

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

type(cars)
```

Accessing Dictionary items:

Items within a dictionary can be accessed by passing key as index.

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

model = cars['model']

print(model)
```

You can also access all the keys and values within a dictionary using keys and values function as shown below:

```
print(cars.keys())     # Accessing keys from dictionary

print(cars.values())       # Accessing values from dictionary
```

You can also get items from a dictionary in the form of key-value pairs using items function.

```
print(cars.items()) # Accessing items from dictionary
```

Iterating over Dictionary Items, Keys and Values

The ‘items’, ‘keys’ and ‘values’ functions return sequences that can be iterated using for loops.

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

for item in cars.items():

    print(item)

# for keys

for key in cars.keys():

    print(key)

# for Values

cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

for value in cars.values():

    print(value)
```

Adding Item to a dictionary

You simply have to pass new key in index and assign it some value.

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

cars['capacity'] = 500

print(cars)
```

Updating a Dictionary

To update the dictionary, use the key for which you want to change the meaning as the index and add a new value to it

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

print(cars)

cars['model'] = 2015

print(cars)
```

Deleting Dictionary Items

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

print(cars)

del cars['model']

print(cars)
```

You can also delete all the items in a dictionary using clear function.

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

print(cars)

cars.clear()

print(cars)
```

Finding Dictionary Length

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

print(len(cars))
```

Checking the existence of an Item in Dictionary

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

print('color' in cars)
```

print('model' not in cars)

Copying Dictionaries

To copy one dictionary to the other, you can use copy function

cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}

cars2 = cars.copy()

print(cars2)

A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.

Lists

Creating a List:

The easiest way to make a list is to enclose a comma-separated list of things within square brackets and assign it to a variable,

colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']

Accessing List Elements

we will access the 2nd element of the list colors:
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
print(colors[1])

you can also access **range of items** from a list using slice operator.
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
sublist = colors[2:4]
print(sublist)

Appending elements to a list

The append function can be used to append elements to a list. The item to append is passed as parameter to the ‘append’ function.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
print(colors)
colors.append('Orange')
print(colors)
```

The remove function is used to remove element from a list.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
print(colors)
colors.remove('Blue')
print(colors)
```

List elements can also be deleted using index numbers.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
print(colors)
del colors[2]
print(colors)
```

Concatenating List

```
nums1 = [2, 4, 6, 8, 10]
nums2 = [1, 3, 5, 7, 9]
result = nums1 + nums2
print(result)
```

Finding length of List

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
print(len(colors))
```

Sorting a List

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
colors.sort()
print(colors)
nums = [12, 4, 66, 35, 7]
nums.sort()
print(nums)
```

Python List Comprehension Syntax

Syntax: newList = [expression(element) for element in oldList if condition]

Parameter:

- expression: Represents the operation you want to execute on every item within the iterable.
- element: The term “variable” refers to each value taken from the iterable.
- iterable: specify the sequence of elements you want to iterate through.(e.g., a list, tuple, or string).
- condition: (Optional) A filter helps decide whether or not an element should be added to the new list.

Return:*The return value of a list comprehension is a new list containing the modified elements that satisfy the given criteria.*
Python List comprehension provides a much more short syntax for creating a new list based on the values of an existing list.

List Comprehension in Python Example

```
numbers = [1, 2, 3, 4, 5]
squared = [x ** 2 for x in numbers]
print(squared)
```

Iteration with List Comprehension

```
# Using list comprehension to iterate through loop
List = [character for character in [1, 2, 3]]

# Displaying list
print(List)
```

Even list using List Comprehension

```
list = [i for i in range(11) if i % 2 == 0]
print(list)
```

Matrix using List Comprehension

```
matrix = [[j for j in range(3)] for i in range(3)]

print(matrix)
```

List Comprehensions translate the traditional iteration approach using for loop into a simple formula hence making them easy to use.

```
# Using list comprehension to iterate through loop
List = [character for character in 'Geeks 4 Geeks!']

# Displaying list
print(List)
```

Output

```
['G', 'e', 'e', 'k', 's', ' ', '4', ' ', 'G', 'e', 'e', 'k', 's', '!']
```

Python List Comprehension using If-else:

```
lis = ["Even number" if i % 2 == 0
       else "Odd number" for i in range(8)]
print(lis)
```

Advantages of List Comprehension:

- More time-efficient and space-efficient than loops.
- Require fewer lines of code.
- Transforms iterative statement into a formula.

Dictionary Manipulation Techniques:

Here's an example of how to define a dictionary in Python:

```
# Define a dictionary

person = {

    "name": "John",

    "age": 30,

    "gender": "Male"

}
```

In this example, the dictionary person contains three key-value pairs. The keys are "name", "age", and "gender", and the values are "John", 30, and "Male", respectively.

You can access the values of a dictionary by using the keys, like this:

```
# Accessing values

print(person["name"])    # Output: John

print(person["age"])     # Output: 30

print(person["gender"])  # Output: Male
```

To add a new key-value pair to a dictionary, you can simply assign a value to a new key

```
# Adding a new key-value pair

person["occupation"] = "Engineer"

print(person)  # Output: {'name': 'John', 'age': 30, 'gender': 'Male', 'occupation': 'Engineer'}
```

To modify an existing value in a dictionary

```
# Modifying an existing value

person["age"] = 35

print(person)  # Output: {'name': 'John', 'age': 35, 'gender': 'Male', 'occupation': 'Engineer'}
```

To remove a key-value pair from a dictionary, you can use the del keyword

```
# Removing a key-value pair

del person["gender"]

print(person)  # Output: {'name': 'John', 'age': 35, 'occupation': 'Engineer'}
```

You can also use various dictionary methods to manipulate dictionaries, such as keys(), values(), items(), get(), pop(), clear(), and others. These methods allow you to access the keys, values, and items (key-value pairs) of a dictionary, get the value of a key (with a default value if the key does not exist), remove a key-value pair from a dictionary and return its value, and clear all key-value pairs from a dictionary, respectively.

```
# Using dictionary methods

print(person.keys())    # Output: dict_keys(['name', 'age', 'occupation'])

print(person.values())  # Output: dict_values(['John', 35, 'Engineer'])

print(person.items())   # Output: dict_items([('name', 'John'), ('age', 35), ('occupation',
'Engineer')])

print(person.get("gender", "Unknown"))  # Output: Unknown

print(person.pop("occupation"))         # Output: Engineer

print(person)  # Output: {'name': 'John', 'age': 35}

person.clear()

print(person)  # Output: {}
```

Python Object Oriented Programming

Python is a versatile programming language that supports various programming styles, including object-oriented programming (OOP) through the use of **objects** and **classes**.
An object is any entity that has **attributes** and **behaviors**. For example, a parrot is an object. It has

- **attributes** - name, age, color, etc.
- **behavior** - dancing, singing, etc.

Similarly, a class is a blueprint for that object.

```
class Parrot:

    # class attribute
    name = ""
    age = 0

# create parrot1 object
parrot1 = Parrot()
parrot1.name = "Blu"
parrot1.age = 10

# create another object parrot2
parrot2 = Parrot()
parrot2.name = "Woo"
parrot2.age = 15

# access attributes
print(f"{parrot1.name} is {parrot1.age} years old")
print(f"{parrot2.name} is {parrot2.age} years old")
```

Output

```
Blu is 10 years old
Woo is 15 years old
```

In the above example, we created a class with the name `Parrot` with two attributes: `name` and `age`.
Then, we create instances of the `Parrot` class. Here, `parrot1` and `parrot2` are references (value) to our new objects.

We then accessed and assigned different values to the instance attributes using the objects name and the `.` notation.

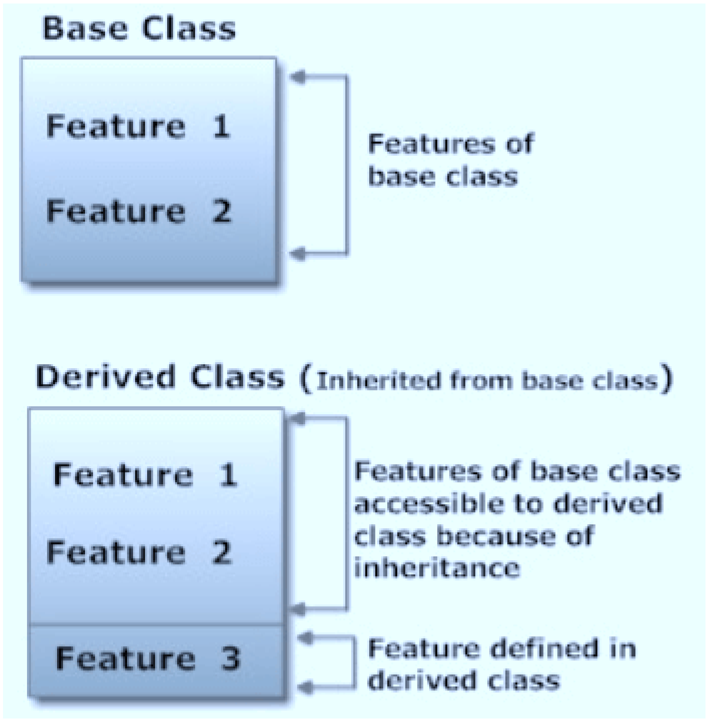
Python Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it.

The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

How Inheritance works

Inheritance is a feature used in object-oriented programming; it refers to defining a new class with less or no modification to an existing class. The new class is called **derived class** and from one which it inherits is called the **base**. Python supports inheritance; it also supports **multiple inheritances**. A class can inherit attributes and behavior methods from another class called subclass or heir class.


Base Class
Feature 1
Feature 2
Features of base class
Derived Class (Inherited from base class)
Feature 1
Feature 2
Feature 3
Features of base class accessible to derived class because of inheritance
Feature defined in derived class

Example 2: Use of Inheritance in Python

```
# base class
class Animal:

    def eat(self):
        print( "I can eat!")

    def sleep(self):
        print("I can sleep!")

# derived class
class Dog(Animal):

    def bark(self):
        print("I can bark! Woof woof!!")

# Create object of the Dog class
dog1 = Dog()

# Calling members of the base class
dog1.eat()
dog1.sleep()

# Calling member of the derived class
dog1.bark();
```

Output

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

Here, `dog1` (the object of derived class `Dog`) can access members of the base class Animal. It's because `Dog` is inherited from `Animal`.

```
# Calling members of the Animal class
dog1.eat()
dog1.sleep()
```

Python Encapsulation:

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of attributes and methods inside a single class.

It prevents outer classes from accessing and changing attributes and methods of a class. This also helps to achieve **data hiding**.
In Python, we denote private attributes using underscore as the prefix i.e single _ or double __. For example,

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

Output

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

In the above program, we defined a `Computer` class.

We used `__init__()` method to store the maximum selling price of `Computer`. Here, notice the code

```
c.__maxprice = 1000
```

Here, we have tried to modify the value of `__maxprice` outside of the class. However, since `__maxprice` is a private variable, this modification is not seen on the output.

As shown, to change the value, we have to use a setter function i.e `setMaxPrice()` which takes price as a parameter.

Polymorphism

Polymorphism is another important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

Let's see an example,

```
class Polygon:
    # method to render a shape
    def render(self):
        print("Rendering Polygon...")

class Square(Polygon):
    # renders Square
    def render(self):
        print("Rendering Square...")

class Circle(Polygon):
    # renders circle
    def render(self):
        print("Rendering Circle...")

# create an object of Square
s1 = Square()
s1.render()

# create an object of Circle
c1 = Circle()
c1.render()
```

Output

```
Rendering Square...
Rendering Circle...
```

In the above example, we have created a superclass: `Polygon` and two subclasses: `Square` and `Circle`. Notice the use of the `render()` method.

The main purpose of the `render()` method is to render the shape. However, the process of rendering a square is different from the process of rendering a circle.
Hence, the `render()` method behaves differently in different classes. Or, we can say `render()` is polymorphic.

Key Points to Remember:

- Object-Oriented Programming makes the program easy to understand as well as efficient.
- Since the class is sharable, the code can be reused.
- Data is safe and secure with data abstraction.
- Polymorphism allows the same interface for different objects, so programmers can write efficient code.