**Python – 4**

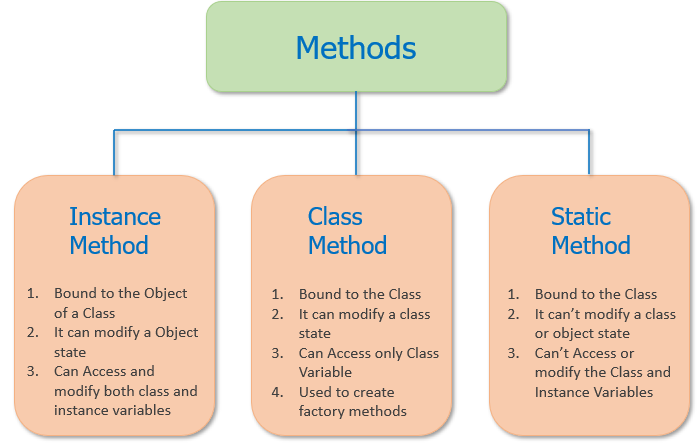
**Excercise Answers**

**1. Demonstrate the usage of staticmethod and classmethod with explanations of when are they useful**

reference : https://pynative.com/python-class-method-vs-static-method-vs-instance-method/

[Instance method](https://pynative.com/python-instance-methods/) performs a set of actions on the data/value provided by the instance variables. If we use instance variables inside a method, such methods are called instance methods.

* [Class method](https://pynative.com/python-class-method/) is method that is called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.
* [Static method](https://pynative.com/python-static-method/) is a general utility method that performs a task in isolation. This method doesn’t have access to the instance and class variable.



from datetime import datetime

current\_year = datetime.today().year

class Car:

base\_price = 100000

def \_\_init\_\_(self, windows, doors, power):

self.windows = windows

self.doors = doors

self.power = power

def what\_base\_price(self):

print("base price: {}".format(self.base\_price))

@classmethod

def revised\_base\_price(cls, inflation):

cls.base\_price = cls.base\_price + cls.base\_price \* inflation

@staticmethod

def check\_year():

if current\_year <= 2022:

return True

else:

return False

car1 = Car(4, 5, 3000)

if(Car.check\_year()):

pass

else:

Car.revised\_base\_price(.10)

print(car1.base\_price)

**2. How is access control on attributes of a class (public, private, etc.) different in Python vs other programming languages?**

Python doesn’t have any mechanisms, that would effectively restrict you from accessing a variable or calling a member method. All of this is a matter of [culture](https://stackoverflow.com/questions/1641219/does-python-have-private-variables-in-classes) and convention.

**Public**

class employee:

    def \_\_init\_\_(self, name, sal):

        self.name=name

        self.salary=sal

You can access employee class’s attributes and also modify their values, as shown below.

>>>e1=Employee("Kiran",10000)

>>> e1.salary

10000

>>> e1.salary=20000

>>> e1.salary

20000

**Protected**

>>> class employee:

    def \_\_init\_\_(self, name, sal):

        self.\_name=name  # protected attribute

        self.\_salary=sal # protected attribute

This changes virtually nothing, you’ll still be able to access/modify the variable from outside the class. You can still perform the following operations:

>>> e1=employee("Swati", 10000)

>>> e1.\_salary

10000

>>> e1.\_salary=20000

>>> e1.\_salary

20000

**Private**

>>> class employee:

    def \_\_init\_\_(self, name, sal):

        self.\_\_name=name  # private attribute

        self.\_\_salary=sal # private attribute

Let’s try to access using private attribute directly –

>>> e1=employee("Bill",10000)

>>> e1.\_\_salary

AttributeError: 'employee' object has no attribute '\_\_salary'

to rectify above error

>>> e1=Employee("Bill",10000)

>>> e1.\_Employee\_\_salary

10000

Let’s put everything altogether for better understanding

class Hello:

def init(self, name):

self.public\_variable = 10

self.\_\_private\_variable = 30

def public\_method(self):

print(self.public\_variable)

print(self.\_\_private\_variable)

print('public')

self.\_\_private\_method()

def \_\_private\_method(self):

print('private')

Now let’s invoke these methods and variables-

>>> hello = Hello('name')

>>> print(hello.public\_variable)

10

>>> hello.public\_method()

10

30

public

private

>>> print(hello.\_\_private\_variable)

not allowed outside the class

AttributeError: 'Hello' object has no attribute '\_\_private\_variable'

>>> hello.\_\_private\_method

not allowed outside the class

AttributeError: 'Hello' object has no attribute '\_\_private\_method'

**3. What is the difference between a class attribute and an instance attribute?**

Instance Attributes are unique to each object, (an instance is another name for an object). Here, any Dog object we create will be able to store its name and age. We can change either attribute of either dog, without affecting any other dog objects we’ve created.

class Dog:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

This \_\_init\_\_ is called the initializer. It is automatically called when we instantiate the class. It’s job is to make sure the class has any attributes it needs. It’s sometimes also used to make sure that the object is in a valid state when it’s instantiated, like making sure the user didn’t enter a negative age for the dog.

We have to include the self parameter so that our initializer has a reference to the new object being instantiated.

Class Attributes are unique to each class. Each instance of the class will have this attribute. It’s sometimes used to specify a defualt value that all objects should have after they’ve been instantiated. Here, our class attribute is species

class Dog:

species = 'mammal'

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

**4 & 5. Learn about dataclasses and namedtuple**

reference : https://realpython.com/python-data-classes/

**Dataclasses**

A data class is a class typically containing mainly data, although there aren’t really any restrictions. It is created using the new @dataclass decorator, as follows:

from dataclasses import dataclass

@dataclass

class DataClassCard:

rank: str

suit: str

>>> queen\_of\_hearts = DataClassCard('Q', 'Hearts')

>>> queen\_of\_hearts.rank

'Q'

>>> queen\_of\_hearts

DataClassCard(rank='Q', suit='Hearts')

>>> queen\_of\_hearts == DataClassCard('Q', 'Hearts')

True

**Namedtuple**

A better alternative to datac lass is the [namedtuple](https://dbader.org/blog/writing-clean-python-with-namedtuples). It has long been used to create readable small data structures. We can in fact recreate the data class example above using a namedtuple like this:

from collections import namedtuple

NamedTupleCard = namedtuple('NamedTupleCard', ['rank', 'suit'])

>>> queen\_of\_hearts = NamedTupleCard('Q', 'Hearts')

>>> queen\_of\_hearts.rank

'Q'

>>> queen\_of\_hearts

NamedTupleCard(rank='Q', suit='Hearts')

>>> queen\_of\_hearts == NamedTupleCard('Q', 'Hearts')

True

**6. Learn about inheritance in Python and show with examples your understanding of the below concepts**

* 1. **Basic inheritance**
  2. **Abstract class (see collections.abc module in Python std lib)**

**c. Protocol (see typing.Protocol module in Python std lib)**

**Inheritance**

Inheritance models what is called an is a relationship. This means that when you have a Derived class that inherits from a Base class, you created a relationship where Derived is a specialized version of Base.

### The Object Super Class

>>> class MyClass:

... pass

>>> c = MyClass()

>>> dir(c)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_',

'\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_',

'\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_',

'\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_',

'\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_']

[dir()](https://docs.python.org/3/library/functions.html" \l "dir) returns a list of all the members in the specified object. You have not declared any members in MyClass, so where is the list coming from? You can find out using the interactive interpreter:

>>> o = object()

>>> dir(o)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_',

'\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_',

'\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_',

'\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_',

'\_\_subclasshook\_\_']

Example

# In hr.py

class PayrollSystem:

def calculate\_payroll(self, employees):

print('Calculating Payroll')

print('===================')

for employee in employees:

print(f'Payroll for: {employee.id} - {employee.name}')

print(f'- Check amount: {employee.calculate\_payroll()}')

print('')

class Employee:

def \_\_init\_\_(self, id, name):

self.id = id

self.name = name

class SalaryEmployee(Employee):

def \_\_init\_\_(self, id, name, weekly\_salary):

super().\_\_init\_\_(id, name)

self.weekly\_salary = weekly\_salary

def calculate\_payroll(self):

return self.weekly\_salary

class HourlyEmployee(Employee):

def \_\_init\_\_(self, id, name, hours\_worked, hour\_rate):

super().\_\_init\_\_(id, name)

self.hours\_worked = hours\_worked

self.hour\_rate = hour\_rate

def calculate\_payroll(self):

return self.hours\_worked \* self.hour\_rate

class CommissionEmployee(SalaryEmployee):

def \_\_init\_\_(self, id, name, weekly\_salary, commission):

super().\_\_init\_\_(id, name, weekly\_salary)

self.commission = commission

def calculate\_payroll(self):

fixed = super().calculate\_payroll()

return fixed + self.commission

salary\_employee = hr.SalaryEmployee(1, 'John Smith', 1500)

hourly\_employee = hr.HourlyEmployee(2, 'Jane Doe', 40, 15)

commission\_employee = hr.CommissionEmployee(3, 'Kevin Bacon', 1000, 250)

payroll\_system = hr.PayrollSystem()

payroll\_system.calculate\_payroll([

salary\_employee,

hourly\_employee,

commission\_employee

])

### **Abstract Base Classes in Python**

The Employee class in the example above is what is called an abstract base class. Abstract base classes exist to be inherited, but never instantiated. Python provides the abc module to define abstract base classes.

The [abc](https://docs.python.org/3/library/abc.html" \l "module-abc) module in the Python standard library provides functionality to prevent creating objects from abstract base classes.

You can modify the implementation of the Employee class to ensure that it can’t be instantiated:

# In hr.py

from abc import ABC, abstractmethod

class Employee(ABC):

def \_\_init\_\_(self, id, name):

self.id = id

self.name = name

@abstractmethod

def calculate\_payroll(self):

pass

This change has two nice side-effects:

1. You’re telling users of the module that objects of type Employee can’t be created.
2. You’re telling other developers working on the hr module that if they derive from Employee, then they must override the .calculate\_payroll() abstract method.

## **Composition in Python**

Composition is an object oriented design concept that models a has a relationship. In composition, a class known as composite contains an object of another class known to as component. In other words, a composite class has a component of another class.

class Employee:

    # constructor for initialization

    def \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

    # instance method

    def emp\_data(self):

        print('Name of Employee : ', self.name)

        print('Age of Employee : ', self.age)

class Data:

    def \_\_init\_\_(self, address, salary, emp\_obj):

        self.address = address

        self.salary = salary

        # creating object of Employee class

        self.emp\_obj = emp\_obj

    # instance method

    def display(self):

        # calling Employee class emp\_data()

        # method

        self.emp\_obj.emp\_data()

        print('Address of Employee : ', self.address)

        print('Salary of Employee : ', self.salary)

# creating Employee class object

emp = Employee('Ronil', 20)

# passing obj. of Emp. class during creation

# of Data class object

data = Data('Indore', 25000, emp)

# call Data class instance method

data.display()

**c. Protocols**

need to refer

https://peps.python.org/pep-0544/