**Упражнение 4: Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект**

**1. зад.** Да се намери евклидовото разстояние между два точки в триизмерно пространство **с,** като се създаде клас point и обекти. Да се създаде конструктор (init ()) и две функции distancefromorgin() и distance().

**Примерно решение:**

# To find the Euclidean distance between two pints in a three dimensional space using class and objects. Class called point consists of a constructor (init ()) and two functions distancefromorgin() and distance()

import math

class point():

def \_\_init\_\_(self,a,b,c):

self.x=a

self.y=b

self.z=c

def distancefromorigin(self):

return ((self.x \*\* 2) + (self.y \*\* 2) +(self.z \*\* 2)) \*\* 0.5

def distance(self, point2):

xdiff = self.x-point2.x

ydiff = self.y-point2.y

zdiff = self.z-point2.z

dist = math.sqrt (xdiff\*\*2 + ydiff\*\*2+ zdiff\*\*2)

return dist

x1,y1,z1= (input("Enter the coordinates of a first point P1(x1,y1,z1): ")).split()

x1,y1,z1 =[int(x1),int(y1), int(z1)]

x2,y2,z2= (input("Enter the coordinates of a second point P2(x2,y2,z2): ")).split()

x2,y2,z2 =[int(x2),int(y2),int(z2)]

# P1 and P2 are objects of point class

p1 = point(x1,y1,z1)

p2 = point(x2,y2,z2)

print('Distance from origin to P1:', p1.distancefromorigin())

print('Distance from origin to P2:', p2.distancefromorigin())

print('Distance from P1 to P2:',p1.distance(p2))

**Резултат:**

Enter the coordinates of a first point P1(x1,y1,z1): 1 2 3

Enter the coordinates of a second point P2(x2,y2,z2): 2 3 4

Distance from origin to P1: 3.7416573867739413

Distance from origin to P2: 5.385164807134504

Distance from P1 to P2: 1.7320508075688772

**2. зад.** Да се създаде клас Person, да се дефинира специален метод \_\_init\_\_ с аргументи - self, name и age. Да се създадат два обекта, да се получи достъп до атрибутите на обекта.

**Примерно решение:**

class Person:

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

self.name = name

self.age = age

k = Person("John", 36)

print(k.name)

print(k.age)

**Резултат:**

John

36

**Забележка:** Ключовата дума **self** in в Python се използва за всички екземпляри в даден клас. Използвайки ключовата дума self, човек може лесно да получи достъп до всички екземпляри, дефинирани в рамките на даден клас, включително неговите методи и атрибути. **init** е един от запазените методи в Python. В ООП той е известен като конструктор. Методът init може да бъде извикан, когато обект е създаден от класа и е необходим достъп за инициализиране на атрибутите на класа.

**3. зад.** Да се създаде клас Employee. Да се дефинира специален метод \_\_init\_\_ (конструктор на клас или метод за инициализация, който Python извиква, когато създава нов екземпляр на класа), чрез аргумента self да се дефинират и зададат атрибути - name и salary. Да се дефинират: променлива empCount в класа, чиято стойност да се споделя между всички екземпляри на този клас; метод displayCount с аргумент self, за извеждане на общият брой служители. Да се създадат екземпляри на класа (като се извика класа, да се използва име на класа и да се предаде displayEmployee (за извеждане на информация за всеки служител – име и заплата). Да се създадат два обекта, да се получи достъп до атрибутите на обекта. Да се изведе информация за всеки служител, както и общият им брой.

**Примерно решение:**

class Employee:

empCount = 0

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

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print ("Total Employee %d" % Employee.empCount)

def displayEmployee(self):

print ("Name : ", self.name,", Salary: ", self.salary)

emp1 = Employee("Zara", 2000)

emp1.displayEmployee()

emp2 = Employee("Manni", 5000)

emp2.displayEmployee()

emp2.displayCount()

**Резултат:**

Name : Zara , Salary: 2000

Name : Manni , Salary: 5000

Total Employee 2

**4. зад.** Да се създадат два класа – person, да се дефинира специален метод \_\_init\_\_ с аргументи - self, n и а (за име и години на персоната). Класът да се наследява от клас employee, да се дефинира специален метод \_\_init\_\_ с аргументи self, n, a, d, s (име, години, титла, заплата). Да се дефинира метод show за извеждане на информация за служителя. Да се създаде клас student, да се дефинира специален метод \_\_init\_\_ и чрез аргумента self да се дефинират и зададат атрибути - id, rno; клас resident, който наследява класовете person и student, да се дефинира специален метод \_\_init\_\_ с аргументи self, n, a, id, rno; да се създаде метод show, за извеждане на информацията за служителя – име, години, титла и заплата. Да се създадат обекти, да се получи достъп до атрибутите на обекта. Да се изведе информация за всеки служител.

**Примерно решение:**

class person:

def \_\_init\_\_(self,n,a):

self.name = n

self.age = a

# employee is the class derived from person using single inheritance

class employee(person):

def \_\_init\_\_(self, n, a, d, s):

person.\_\_init\_\_(self, n, a)

self.designation = d

self.salary = s

def show(self):

print("Employee Details:")

print(" Name: ", self.name,"\n Age:", self.age, "\n Designation:", self.designation, "\n Salary:", self.salary)

{

e = person("Arun",35)

r = employee("John", 30, “admin”, 5000)

r.show()

e.show() - ще изведе грешка

}

# student is a base class

class student:

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

self.studentId = id

self.roomno = rno

# resident is a class derived from person and student using multiple inheritance

class resident(person, student):

def \_\_init\_\_(self, n, a, id,rno):

person.\_\_init\_\_(self, n, a)

student.\_\_init\_\_(self, id, rno)

def show(self):

print("Resident Details:")

print(" Name:", self.name,"\n Age: ",self.age, "\n Id:", self.studentId,"\n Room no.:", self.roomno)

# Creating objects of employee and resident classes

e = employee("Arun",35,"Data analyst",50000)

r = resident("John", 30, 201900025,203)

e.show()

r.show()

**Резултат:**

Employee Details:

Name: Arun

Age: 35

Designation: Data analyst

Salary: 50000

Resident Details:

Name: John

Age: 30

Id: 201900025

Room no.: 203

**Забележка: method overloading.** В Python можете да дефинирате метод по такъв начин, че да има няколко начина да го извикате. В зависимост от дефиницията на функцията, тя може да бъде извикана с нула, един, два или повече параметъра.

**5. зад.** Да се създаде клас distance за измерване на разстоянието във футове и инчове.

**Примерно решение:**

# distance is a class. Distance is measured in terms of feet and inches

class distance:

def \_\_init\_\_(self,f,i):

self.feet=f

self.inches=i

# overloading of binary operator > to compare two distances

def \_\_gt\_\_(self,d):

if(self.feet>d.feet):

return(True)

elif((self.feet==d.feet) and (self.inches>d.inches)):

return(True)

else:

return(False)

# overloading of binary operator + to add two distances

def \_\_add\_\_(self, d):

i=self.inches + d.inches

f=self.feet + d.feet

if(i>=12):

i=i-12

f=f+1

return distance(f,i)

# displaying the distance

def show(self):

print("Feet= ",self.feet, "Inches= ",self.inches)

a,b= (input("Enter feet and inches of distance1: ")).split()

a,b =[int(a),int(b)]

c,d= (input("Enter feet and inches of distance2: ")).split()

c,d =[int(c),int(d)]

d1 = distance(a,b)

d2 = distance(c,d)

if(d1>d2):

print("Distance1 is greater than Distance2")

else:

print("Distance2 is greater or equal to Distance1")

d3=d1+d2

print("Sum of the two Distance is:")

d3.show()

**Резултат:**

Enter feet and inches of distance1: 8 4

Enter feet and inches of distance2: 6 9

Distance1 is greater than Distance2

Sum of the two Distance is:

Feet= 15 Inches= 1

**6. зад.** Да се създаде клас Parent и наследник Child. В Parent да се създаде метод\_\_init\_\_ (конструктор) и метод get\_value. Child наследява метода get\_value.

**Примерно решение:**

class Parent(object):

def \_\_init\_\_(self):

self.value = 5

def get\_value(self):

return self.value

class Child(Parent):

pass

c=Child()

print (c.get\_value())

**Резултат:**

5

**Забележка:** Python изтрива ненужните обекти (вградени типове или екземпляри на клас) автоматично, за да освободи пространството в паметта. Процесът, чрез който Python периодично възстановява блокове памет, които вече не се използват, се нарича събиране на боклук.

a = 40 # Create object <40>

b = a # Increase ref. count of <40>

c = [b] # Increase ref. count of <40>

del a # Decrease ref. count of <40>

b = 100 # Decrease ref. count of <40>

c[0] = -1 # Decrease ref. count of <40>

**Забележка:** Обикновено не се забелязва, когато събирачът за боклук унищожи неизползван екземпляр и възстанови мястото му. **Но един клас може да реализира специалния метод del(), наречен деструктор, който се извиква, когато екземплярът е на път да бъде унищожен. Този метод може да се използва за почистване на всички ресурси извън паметта, използвани от даден екземпляр.**

**Забележка:** Обикновено се указва атрибут на обект на клас чрез свързване на стойност към идентификатор в тялото на класа.

**7. зад.** Примери за указване на атрибут на обект на клас чрез свързване на стойност към идентификатор в тялото на класа.

**class C1**(object):

x = 23

print(C1.x) *# prints:* ***23***

Обектът на класа C1 има атрибут с име x, обвързан със стойността 23, и C1.x се отнася до този атрибут. Можете също да обвързвате или развързвате атрибути на клас извън тялото на класа.

**class C2**(object): **pass**

C2.x = 23

print(C2.x)

*# prints:* ***23***

**class C3**(object):

x = 23

y = x + 22 *# must use just x,* ***not*** *C3.x*

**Забележка:** Въпреки това, в изрази в методи, дефинирани в тялото на класа, препратките към атрибутите на класа трябва да използват напълно квалифицирано име, а не просто име. Например:

**class C4**(object): x = 23

**def** amethod(self):

print(C4.x) *# must use C4.x or self.x,* ***not*** *just x!*

**class C5**(object):

**def** hello(self):

print('Hello')

**Забележка:** Слотовете осигуряват специален механизъм за намаляване на размера на обектите. Това е концепция за оптимизиране на паметта на обекти.

**Синтаксис:**

class myClass(object):

# defining the slots

\_\_slots\_\_ = (par1, par2)

def \_\_init\_\_(self, \*args, \*\*kwargs):

# initializing the values

self.par1 = value1

self.par2 = value2

**8 зад.** Да се дефинира клас gfg без аргументи, дa се дефинира\_\_slots\_\_ с два аргумента и да се инициализират, да се създаде специален метод \_\_init\_\_ с аргумент self; да се инициализират атрибутите. Да се създаде обект от клас gfg, да се получи достъп до слота и атрибутите му.

**Примерно решение:**

# defining the class.

class gfg:

# defining the slots.

\_\_slots\_\_ =('course', 'price')

def \_\_init\_\_(self):

# initializing the values

self.course ='DSA Self Paced'

self.price = 3999

# create an object of gfg class

a = gfg()

# print the slot

print(a.\_\_slots\_\_)

# print the slot variable

print(a.course, a.price)

**Резултат:**

('course', 'price')

DSA Self Paced 3999

**9 зад.** Да се дефинира клас gfg без аргументи, дa се дефинира\_\_slots\_\_ с два аргумента и да се инициализират, да се създаде специален метод \_\_init\_\_ с аргумент self; да се инициализират атрибутите. Да се създаде обект от клас gfg, , да се получи достъп до слота и атрибутите му. Да се зададат нови стойности на аргументите на слота и да се отпечатат.

**Примерно решение:**

# defining the class.

class gfg:

# defining the slots.

\_\_slots\_\_ =('course', 'price')

def \_\_init\_\_(self):

# initializing the values

self.course ='oops'

self.price = 5999

# create an object of gfg class

a = gfg()

# print the slot

print(a.\_\_slots\_\_)

# print the slot variable

print(a.course, a.price)

# change the value of the variable

a.course ='System Design'

# print the slot variable

print(a.course, a.price)

# change the value of the variable

a.price = 9999

# print the slot variable

print(a.course, a.price)

**Резултат:**

('course', 'price')

oops 5999

System Design 5999

System Design 9999

**Забележка:** Капсулирането е и основната концепция в ООП. Той предотвратява директния достъп до атрибутите на обекта от извикващият код. По отношение на капсулирането директно в езика за програмиране Python, може да скриете атрибутите на класа, като ги направите частни и ограничите достъпа до тях чрез специални методи, които също се наричат ​​свойства.

Нека се промени дефинирания клас от предходния слайд, като се дефинират свойства в него:

**10 зад.** Да се създаде клас Person, да се дефинира специален метод \_\_init\_\_ с аргументи - self и name (за името и годините на потребителя). Да се създаде метод set\_age с аргументи self и age, който да извършва проверка за коректността на зададените години – те да са в интервала [1;110], иначе да се изведе грешка. Да се създаде метод get\_age с аргумент self, който връща age. Да се създаде метод get\_name с аргумент self, който връща name. Да се създаде метод display\_info с аргумент self за извеждане на името и възрастта на потребителя. Да се създаде обект. Да се извикват метода display\_info и методa set\_age, като се присвои стойност в зададения интервал за годините му и извън него.

**Примерно решение:**

class Person:

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

self.\_\_name = name # установяване на име

self.\_\_age = 1 # установяване на възраст

def set\_age(self, age):

if 1 < age < 110:

self.\_\_age = age

else:

print("Недопустима възраст")

def get\_age(self):

return self.\_\_age

def get\_name(self):

return self.\_\_name

def display\_info(self):

print(f"Име: {self.\_\_name}\tВъзраст: {self.\_\_age}")

tom = Person("Tom")

tom.display\_info() # Име: Tom Възраст: 1

tom.set\_age(-3486) # Недопустима възраст

tom.set\_age(25)

tom.display\_info() # Име: Tom Възраст: 25

**11 зад.** Нека се пренапише класа Person от задача 10, като се използват анотации.

**Примерно решение:**

class Person:

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

self.\_\_name = name # установяване на име

self.\_\_age = 1 # установяване на възраст

@property

def age(self):

return self.\_\_age

@age.setter

def age(self, age):

if 1 < age < 110:

self.\_\_age = age

else:

print("Недопустима възраст")

@property

def name(self):

return self.\_\_name

def display\_info(self):

print(f"Име: {self.\_\_name}\tВъзраст: {self.\_\_age}")

tom = Person("Tom")

tom.display\_info() # Име: Tom Възраст: 1

tom.age = -3486 # Недопустима възраст

print(tom.age) # 1

tom.age = 36

tom.display\_info() # Име: Tom Възраст: 36

**12. зад. Наследяване**

**Вариант 1:**

class Person:

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

self.\_\_name = name # име

@property

def name(self):

return self.\_\_name

def display\_info(self):

print(f"Name: {self.\_\_name} ")

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

self.\_\_name = name # име на работника

@property

def name(self):

return self.\_\_name

def display\_info(self):

print(f"Name: {self.\_\_name} ")

def work(self):

print(f"{self.name} works")

**Забележка:** Класът Employee във Вариант 1 има същите атрибути и методи като класа Person, тъй като служителят е човек. Така че в класът Employee е добавен само методът works, останалата част от кода повтаря функционалността на класа Person. Но за да не се дублира функционалността на един клас в друг, в този случай е по-добре да се използва наследяване. Ако се наследи класа Employee от класа Person реализацията ще е:

**Вариант 2:**

class Person:

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

self.\_\_name = name # име

@property

def name(self):

return self.\_\_name

def display\_info(self):

print(f"Name: {self.\_\_name} ")

class Employee(Person):

def work(self):

print(f"{self.name} works")

tom = Employee("Tom")

print(tom.name) # Tom

tom.display\_info() # Name: Tom

tom.work() # Tom works

**Забележка:** Трябва да се има предвид, че Employee НЯМА частни атрибути от тип \_\_name. Например, НЕ може да получи достъп до частния атрибут в работния метод self.\_\_name:

def work(self):

print(f"{self.\_\_name} works") # ! Error

class Employee(Person):

def work(self):

print(f"{self.name} works")

tom = Employee("Tom")

print(tom.name) # Tom

tom.display\_info() # Name: Tom

tom.work() # Tom works

**13. зад. Множествено наследяване**

**Забележка:** Една от отличителните черти на езика Python е поддръжката за множествено наследяване, т.е. един клас може да бъде наследен от няколко класа:

# клас Employee

class Employee:

def work(self):

print("Employee works")

# клас Student

class Student:

def study(self):

print("Student studies")

#WorkingStudent наследява класовете Employee и Student

class WorkingStudent(Employee, Student):

pass

# клас Working Student

tom = WorkingStudent()

tom.work() # Employee works

tom.study() # Student studies

**Вариант 2:**

class Employee:

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

self.\_\_name = name

@property

def name(self):

return self.\_\_name

def work(self):

print(f"{self.name} works")

class Student:

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

self.\_\_name = name

@property

def name(self):

return self.\_\_name

def study(self):

print(f"{self.name} studies")

class WorkingStudent(Employee, Student):

pass

tom = WorkingStudent("Tom")

tom.work() # Tom works

tom.study() # Tom studies

def study(self):

print(f"{self.name} studies")

class WorkingStudent(Employee, Student):

pass

tom = WorkingStudent("Tom")

tom.work() # Tom works

tom.study() # Tom studies

**14. зад. Предефиниране на функционалността на базовия клас.**

Класът Employee може да възприеме функционалността на клас Person

**Примерно решение:**

class Person:

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

self.\_\_name = name # име

@property

def name(self):

return self.\_\_name

def display\_info(self):

print(f"Name: {self.\_\_name} ")

**Забележка:** Но какво ще стане, ако трябва да се промени нещо от тази функционалност? Например да се добави нов атрибут към служител чрез конструктора, който ще съхранява компанията, в която работи, или да се промени реализацията на метода display\_info. Python позволява да замени функционалността на базов клас.

**Примерно решение след промяната:**

class Employee(Person):

def work(self):

print(f"{self.name} works")

class Employee(Person):

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

        super().\_\_init\_\_(name)

        self.company = company

    def display\_info(self):

        super().display\_info()

        print(f"Company: {self.company}")

    def work(self):

        print(f"{self.name} works")

tom = Employee("Tom", "Microsoft")

tom.display\_info()  # Name: Tom

                     # Company: Microsoft

**Забележка:** Тук се добави нов атрибут към клас Employee – self.company, който се съхранява от компанията на служителя. Съответно методът \_\_init\_\_() приема три параметъра: вторият за задаване на името и третия за настройка на компанията. Но ако конструкторът е дефиниран в базовия клас с помощта на метода \_\_init\_\_ и е необходимо да се промени логиката на конструктора в производния клас, тогава в конструктора на производния клас трябва да се извика конструктора на базовия клас. Т.е. в конструктора Employee трябва да се извика конструктора на клас Person.

**Забележка:** Изразът super() се използва за препращане към базовия клас. И така, в конструктора Employee се извършва извикването:

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

**Забележка:** Този израз представлява извикване към конструктора на класа Person, на който се предава името на работника. И това е логично. В крайна сметка името на работника е зададено в конструктора на класа Person. В самия конструктор Employee се задава само свойството на компанията. Освен това в класа Employee, методът display\_info() е отменен, като към него се добавя продукцията на компанията на служителя. Методът може да се дефинира по следния начин:

def display\_info(self):

print(f"Name: {self.name}")

print(f"Company: {self.company}")

**Забележка:** Тогава изходният ред за име ще повтори кода от класа Person. Ако тази част от кода съвпада с метода от класа Person, тогава няма смисъл да се повтаря, така че отново, като се използва изразът super(), ще стане обръщане към реализацията на метода display\_info в класа Person:

def display\_info(self):

super().display\_info() # обръщение към метод display\_info в клас Person

print(f"Company: {self.company}")

**Забележка:** След това може да се извика конструктора Employee, за да се създаде обект от този клас и да се извика метода display\_info:

tom = Employee("Tom", "Microsoft")

tom.display\_info()

**Забележка:** При работа с обекти може да се наложи извършване на определени операции в зависимост от вида им. А с вградената функция isinstance() може да провери типа на обекта. Тази функция приема два параметъра:

isinstance(object, type)

**Забележка:** Първият параметър представлява обекта, а вторият параметър представлява типа, за който ще се тества. Ако обектът представлява посочения тип, тогава функцията връща True.

**15зад.** Да се създаде йерархия от класове Person и класове Employee и Student наследяващи Person:

**Примерно решение:**

class Person:

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

self.\_\_name = name # име

@property

def name(self):

return self.\_\_name

def do\_nothing(self):

print(f"{self.name} does nothing")

# клас работник

class Employee(Person):

def work(self):

print(f"{self.name} works")

# клас студент

class Student(Person):

def study(self):

print(f"{self.name} studies")

def act(person):

if isinstance(person, Student):

person.study()

elif isinstance(person, Employee):

person.work()

elif isinstance(person, Person):

person.do\_nothing()

tom = Employee("Tom")

bob = Student("Bob")

sam = Person("Sam")

act(tom) # Tom works

act(bob) # Bob studies

act(sam) # Sam does nothing

**16зад.** **Атрибути на класа и статични методи**

В допълнение към атрибутите на обекти в клас, могат да се дефинират атрибути на клас. Такива атрибути се дефинират като променливи на ниво клас.

**Примерно решение:**

class Person:

type = "Person"

description = "Describes a person"

print(Person.type) # Person

print(Person.description) # Describes a person

Person.type = "Class Person"

print(Person.type) # Class Person

**Забележка:** Тук в класа Person са дефинирани два атрибута: type, който съхранява името на класа, и description, който съхранява описанието на класа. За да се позовем на атрибутите на клас, може да се използва името на класа, например: Person.type, и, подобно на атрибутите на обект, могат да се получат и променят техните стойности.

**Подобни атрибути са общи за всички обекти от класа.**

**Примерно решение:**

class Person:

type = "Person"

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

self.name = name

tom = Person("Tom")

bob = Person("Bob")

print(tom.type) # Person

print(bob.type) # Person

# изменяме атрибута на класса

Person.type = "Class Person"

print(tom.type) # Class Person

print(bob.type) # Class Person

Атрибутите на класа могат да се използват за ситуации, в които трябва да се дефинират някои общи данни за всички обекти. Например:

class Person:

default\_name = "Undefined"

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

if name:

self.name = name

else:

self.name = Person.default\_name

tom = Person("Tom")

bob = Person("")

print(tom.name) # Tom

print(bob.name) # Undefined

В този случай атрибутът default\_name съхранява името по подразбиране. И ако празен низ за името се предаде на конструктора, тогава стойността на атрибута на класа default\_name се предава на атрибута name. За обръщение към атрибут на клас в метод, може да се използва името на класа:

self.name = Person.default\_name

**17зад.** **Атрибут на класа**. **Възможно е атрибут на клас и атрибут на обект да имат едно и също име.** Ако стойността не е зададена за атрибут на обект в кода, тогава стойността на атрибута на класа може да се използва за него.

**Примерно решение:**

class Person:

name = "Undefined"

def print\_name(self):

print(self.name)

tom = Person()

bob = Person()

tom.print\_name() # Undefined

bob.print\_name() # Undefined

bob.name = "Bob"

bob.print\_name() # Bob

tom.print\_name() # Undefined

Тук методът print\_name използва атрибута name на обекта, но никъде в кода не е зададен този атрибут. Но на ниво клас атрибутът name е зададен. Следователно, първия път, когато методът print\_name бъде извикан, той ще използва стойността на атрибута на класа:

tom = Person()

bob = Person()

tom.print\_name() # Undefined

bob.print\_name() # Undefined

След това обаче може да се промени атрибута set на обекта:

bob.name = "Bob"

bob.print\_name() # Bob

tom.print\_name() # Undefined

Освен това вторият обект - tom ще продължи да използва атрибута клас. И ако се промени атрибута на класа, стойността tom.name също ще се промени:

tom = Person()

bob = Person()

tom.print\_name() # Undefined

bob.print\_name() # Undefined

Person.name = "Some Person" # промяна на стойността на атрибута на класа

bob.name = "Bob" # установяване на атрибута на обекта

bob.print\_name() # Bob

tom.print\_name() # Some Person

**18зад.** **Статични методи**

В допълнение към обикновените методи, един клас може да дефинира статични методи. Такива методи се предшестват от анотацията @staticmethod и се отнасят до класа като цяло. Статичните методи обикновено дефинират поведение, което е независимо от конкретен обект:

**Примерно решение:**

class Person:

\_\_type = "Person"

@staticmethod

def print\_type():

print(Person.\_\_type)

Person.print\_type() # Person – достъп до статичен метод чрез името на класа

tom = Person()

tom.print\_type() # Person – обръщение към статичен метод чрез името на обекта

class Person:

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

self.name = name # установяване на име

self.age = age # установяване на възраст

def display\_info(self):

print(f"Name: {self.name} Age: {self.age}")

tom = Person("Tom", 23)

print(tom)

Когато се стартира, програмата се показва следното:

<\_\_main\_\_.Person обект на 0x10a63dc00>

Това не е много информативна информация за обекта. Разбира се, може да заобиколи това, като се дефинира допълнителен метод в класа Person, който показва данните на обекта - в примера по-горе това е методът display\_info. Но има и друг изход – дефиниране на метода \_\_str\_\_() в класа Person (две долни черти от всяка страна):

class Person:

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

self.name = name # установяване на име

self.age = age # установяване на възраст

def display\_info(self):

print(self)

# print(self.\_\_str\_\_()) # или по този начин

def \_\_str\_\_(self):

return f"Name: {self.name} Age: {self.age}"

tom = Person("Tom", 23)

print(tom) # Name: Tom Age: 23

tom.display\_info() # Name: Tom Age: 23

Методът \_\_str\_\_трябва да върне низ. И в този случай се връща основната информация за лицето. Ако трябва да се използва тази информация в други методи на класа, тогава може да се използва израза self.\_\_str\_\_()

Изходът на конзолата ще бъде различен:

Name: Tom Age: 23Name: Tom Age: 23

**Задължителни задачи:**

1. Създайте клас Singer, с аргументи - name, age, fans, money. Създайте методи ShowInfo, който да показва информация за изпълнителя; CreateSong, PlayConcert, които да променят някои от свойствата на класа . Да се създадат поне два обекта на класа и да се тестват различните методи.

2. Задача 1 да се допълни, като към класът Singer се добави капсулация на атрибутите и методи за get и set. Какви проверки биха могли да се добавят в тези методи?

3. Към задача 2 да се добави поне 1 скрит (private) метод към класа Singer. По какъв начин използвахте този метод?

**Задължителна домашна работа:**

1. Да се декларира клас Time (астрономическо време). Членове-данни да са: час, минути, секунди. Методите на класа да включват функции за: въвеждане елементите на класа (да се предвиди контрол на входните данни, ако данните не са допустими, да се въвежда 0); извеждането да е в универсален формат – чч.мм.сс AP/MP. Да се дефинират методи на класа. Да се състави главна програма за тестване на дефинираните методи.