class Shapa:  
 pay = 1000  
 discount = 0.8  
 def \_\_init\_\_(self, name, pay, discount):  
 self.name = name  
 self.pay = pay  
 self.discount = discount  
  
 def get\_pay(self):  
 return self.pay \* self.discount  
  
 def get\_discount(self):  
 return Shapa.discount \* Shapa.pay  
  
name = Shapa("vano", 800, 0.7)  
print(name.get\_pay())  
print(name.get\_discount())

ჯერ ვიძახებთ ინსტანსის დიქსოოუნთს შემდეგ კლასის დისქოუნთს

რადგან self ნიშნავს ინსტანსის კლასს ჯერ ეძებს ობიექტის ინსტანსში ხოლო შემდეეგ თუ ობიექტის ინსტანსში არიყო გადადის კლასის ინსტანზე

class Shapa:  
 pay = 1000  
 discount = 0.8  
 def \_\_init\_\_(self, name, pay, discount):  
 self.name = name  
  
 def get\_pay(self):  
 return self.pay \* self.discount  
  
name = Shapa("vano", 800, 0.7)  
print(name.get\_pay())

რადგან ობიექტს არ გააჩნია ინსტანსი გადაწოდებული არგუმენტი დააიგნორა და აიღო კლასის ინსტანსიდან

**@classmethod** - იმისთვის რომ ობიექტმა კლასიდან აიღოს ინსტანსი დაარა ინიტიდან თავზე უნდა გავუწეროთ ეს მეთოდი

@classmethod  
def get\_discount(cls):  
 return cls.discount \* cls.pay

ამშემთხვევაში არა ინიტის ინსტანსს არამედ კლასის ინსტანს წამოიღებს ხოლო cls\_ს ვწერთ self\_ის მაგივრად რადგან ეს უფრო მიღებული პრაქტიკაა

**@staticmethod** - თუ ვქმნით კლასში ისეთ ფუნქციას რომელსაც ობიექტის ანნკლასის ინსტანსებს არ გადავცემთ შესაბბამისად პარამეტრიც არ დაჭირდება თუმცა პარამეტრს თუ არ გავუწერთ დაერორდება და ამ მეთოდს თუ გავუწერთ თავზე უკბე პარამეტრი აღარ იქნება საჭირო

class Shapa:  
 pay = 1000  
 discount = 0.8  
 def \_\_init\_\_(self, name, pay, discount):  
 self.name = name  
 self.pay = pay  
 self.discount = discount

@staticmethod  
 def rest\_days ():  
 day = input("Enter the day: ")  
 if day == "Monday":  
 return "you need to work"  
 else:  
 return "you need to rest"  
  
  
name = Shapa("vano", 800, 0.7)  
print(name.rest\_days())

**Protecdid ცვლადები**

პროტექდიდ ცვლადები არის დაცული ცვლადები ანუ მასზე გარედდან წვდომა არ შეიძლება და იწერება ცვლადის წინ ერთი ტირით

class Protect:  
 def \_\_init\_\_(self,name, age):  
 self.name = name  
 self.\_age = age

protect = Protect("vano",18)  
print(protect.name)  
print(protect.\_age)

გვაქვს პაბლიკ ინსტანსი ანუ name და პროტექტიდ ინსტანსი ანუ \_age

მიუხედავად იმისა რომ არ შეიძლება გარედან წვდომა მაინც დაპრინტავს პრინტი რადგან ასეთი ცვლადები არის დეველოპერებისთვის და მასზე გარედან წვდომა ან შეცვლა არარის რეკკომენდირებული თუმცა შესაძლებელია

ხოლო თუ ასეთი ცვლადები გვიდნა შევცვალოთ მაშინ კლასშივე ვწერთ მეთოდს რომელიც ამ ცვლადს დააბრუნებს და ამ მეთოდის გამოძახება და შეცვლა შესაძლებელი იქნება

class Protect:  
 def \_\_init\_\_(self,name, age):  
 self.name = name  
 self.\_age = age  
  
 def get\_age(self):  
 return self.\_age  
  
protect = Protect("vano",18)  
print(protect.name)  
print(protect.get\_age())

გეთ ეიჯ ფუნქცია აბრუნებს ამ ცვლადს და ფუნქციას ვიძახებთ

**private ცვლადები**

ფრავით ცვლადებს თ გამოვიძახებთ ასე გავა ერორზე იწერება ორი ქვედატირით

class Protect:  
 def \_\_init\_\_(self,name, age, pay):  
 self.name = name  
 self.\_age = age  
 self.\_\_pay = pay  
  
protect = Protect("vano",18, 500)  
print(protect.\_\_pay)

მაგრამ თუ გარედან შევცვლით და შემდეგ გამოვიძახებთ მაშინ გამოიძახება და არ გავა ერორზე

class Protect:  
 def \_\_init\_\_(self,name, age, pay):  
 self.name = name  
 self.\_age = age  
 self.\_\_pay = pay  
  
  
protect = Protect("vano",18, 500)  
protect.\_\_pay = 2000  
print(protect.\_\_pay)

**@property** თუ გვინდა ასეთი ცვლადების ატრიბუტები გადავაქციოთ შეცვლადად ვუწერთ ამ მეთოდს და რადგან ატრიბუტია და ფაქტიურად ცვლადს ვიძახებთ მაშინ ფრჩხილები აღარაა საჭირო

class Protect:  
 def \_\_init\_\_(self,name, age, pay):  
 self.name = name  
 self.\_age = age  
 self.\_\_pay = pay  
  
 @property  
 def pay(self):  
 return self.\_\_pay  
  
protect = Protect("vano",18, 500)  
print(protect.pay)

**@.setter** - არის სეთერი ხოლო @property არის გეთერი. ეს სეთერი აკეთებს შემდეგს რომ იღებს ფრაივათ ცვლადს და რასაც მივანიჭებთ იმად შეცვლის მას

class Protect:  
 def \_\_init\_\_(self,name, age, pay):  
 self.name = name  
 self.\_age = age  
 self.\_\_pay = pay  
  
 @property  
 def pay(self):  
 return self.\_\_pay  
  
 @pay.setter  
 def pay(self,pay):  
 self.\_\_pay = pay  
  
protect = Protect("vano",18, 500)  
print(protect.pay)  
protect.pay = 2000  
print(protect.pay)

რადგან სეთერია უფრჩხილებო გამოძახება იძახებს ფროფერთის ხოლო სეთერის გამოძახება არ ხდება არამედ სეთერს ეწერება მმნიშვნელობა იცვლება და შემდეგ ისევ იძახება ფროფერთი

**ფუნქტორები**

თუ კლას გამოვიძახებთ პირდაპირ დაერორდება რადგან არაა callable

class Multiplayer:  
 def \_\_init\_\_(self, a):  
 self.a = a  
mult = Multiplayer(2)  
print(mult())

**\_\_call\_\_** მეთოდის გაწერით შეგვეძლება ამ მეთოდის გამოძახება და მოითხოვს გადაცემული პარამეტრს

class Multiplayer:  
 def \_\_init\_\_(self, a):  
 self.a = a  
   
 def \_\_call\_\_(self, b):  
 return self.a \* b  
  
mult = Multiplayer(2)  
print(mult(5))

ახლა უკვე გამოიძახა კლასიც მიენიჭა კლასს 2 ხოლო call\_ს 5 რადგან კლასის ინსტანსი არის 2 და გამოძახებით კლასის ინსტანს ვამრავლებთ მეთოდის ინსტანსზე დაპრინტავს შედეგს 10\_ს

class Multiplayer:  
 def \_\_init\_\_(self, a):  
 self.a = a  
  
 def \_\_call\_\_(self, b):  
 return self.a \* b  
  
mult = Multiplayer(2)  
triple = Multiplayer(5)  
  
print(mult(5))  
print(triple(5))

აქ შეიქმნა მეორე ინსტანსი და მიენიჭა სხვა არგუმენტი

**დეკორატორების შექმნა**

დეკორატორი არის ფუნქცია რომელიც წინასწარ უცვლის ფუნქციას ჩვენს ფუნქციას

დეკორატორის შექმნა ხდება def ქივორდით და გადაეცემ პარამეტრი ძირითადად func

შიგნით იქმნება ისევ ფუნქცია რომელსაც ვარქმევთ ძირითადად wrapper\_ს და გადავცემთ ორ პარამეტრს \*args და \*\*kwargs\_ს

def change\_value (func):  
 def wrapper(\*args,\*\*kwargs):  
 a =args[0] + 3  
 b = args[1] + 5  
 return func(a,b)  
 return wrapper

a იქნება არგსიდან მიღებული მენულე ანუ პირველი არგუმენტი ხოლო b არგსიდან მიღებული მეორე არგუმენტი და პირველ მიღებულ არგუმენტს დაუმატებს 3\_ს ხოლო მეორეს 5\_ს და wrapper\_ი დააბრუნებს func ფუნქციიდან ანუ რასაც დავადებთ ფუნქციას იმ ფუნქციიდან მიღებულ არგუმენტებს პირველს და მეორეს ხოლო change value ფუნქცია დააბრუნებს ამ ვრაპერს

def change\_value (func):  
 def wrapper(\*args,\*\*kwargs):  
 a =args[0] + 3  
 b = args[1] + 5  
 return func(a,b)  
 return wrapper  
  
@change\_value  
def print\_number(x, y):  
 return f"x: is {x}, y: is {y}"

print(print\_number(1, 2))

print\_number\_ს აქვს ორი არგუმენტი 1 და 2, დეკორატორის ვრაპერმა მიიღო ორივე არგუმენტი და პირველი არგუმენტს დაუმატა 3 ხოლო მეორეს 5 და func\_ით ანუ print\_function \_ით დააბრუნა პირველი დამეორე არგუმენტები დამატებული ოღონდ ხოლო რომ დავპრინტე ფუნქცია იქსი უკვე გახდა არა 1 არამედ დამატებული 3 ანუ 4 და იგრეკი არა 2 არამედ დამატებული 5

time მეთოდი

მეთოდით ვიღებთ დროს და ვიყენებთ როგორც გვინდა

**Sleep** - არის დაყოვნება ანუ რამდენი დრო უნდა იყოს დაყოვნებული ფუნქცია რომ გადავიდეს შემდეგ ფუნქციონალზე

import time  
  
def func():  
 start = time.time()  
 print("start here")  
 time.sleep(2)  
 print("end here")  
 end = time.time()  
 return f"mounda {end - start} wami"  
print(func())

გამოვიძახე თუ არა ჩარირთო დრო შემდეგ დავაყოვნე ორი წამით და დასრულდა თუ არა დასრულდა დრო და ჩაინიშნა end ცვლადში და რაც დასრულებაზე იყო დრო იმას გამოვაკელი დაწყების დრო და მივიღე რამდენი წამი დაჭირდა შესრულებას

თუ მინდა რომ გავიგო ყოველმა ფუნქციამთუ კოდმა რა დრო მოანდომა შესრულებას

სათითაოდ რომ არ ვუწერო ეს კოდი შევქმნი დეკორატორს და პირდაპირ დამიწერს დროს

import time  
  
def deco(func):  
 def wrapper(\*args, \*\*kwargs):  
 start = time.time()  
 result = func(\*args, \*\*kwargs)  
 end = time.time()  
 print(f"mounda {end - start} wami")  
 return result  
 return wrapper  
  
@deco  
def func():  
 time.sleep(2)  
  
func()

შეიქმნა ფუნქცია მის ამუშავებას უნდა 2 წამი და დაწყებიდან დასრულებამდე რადგან

მოუნდა ორი წამი დაპრინტავს დეკორატორი პრინტს