BTU ბიზნესისა და ტექნოლოგიების უნივერსიტეტი

python

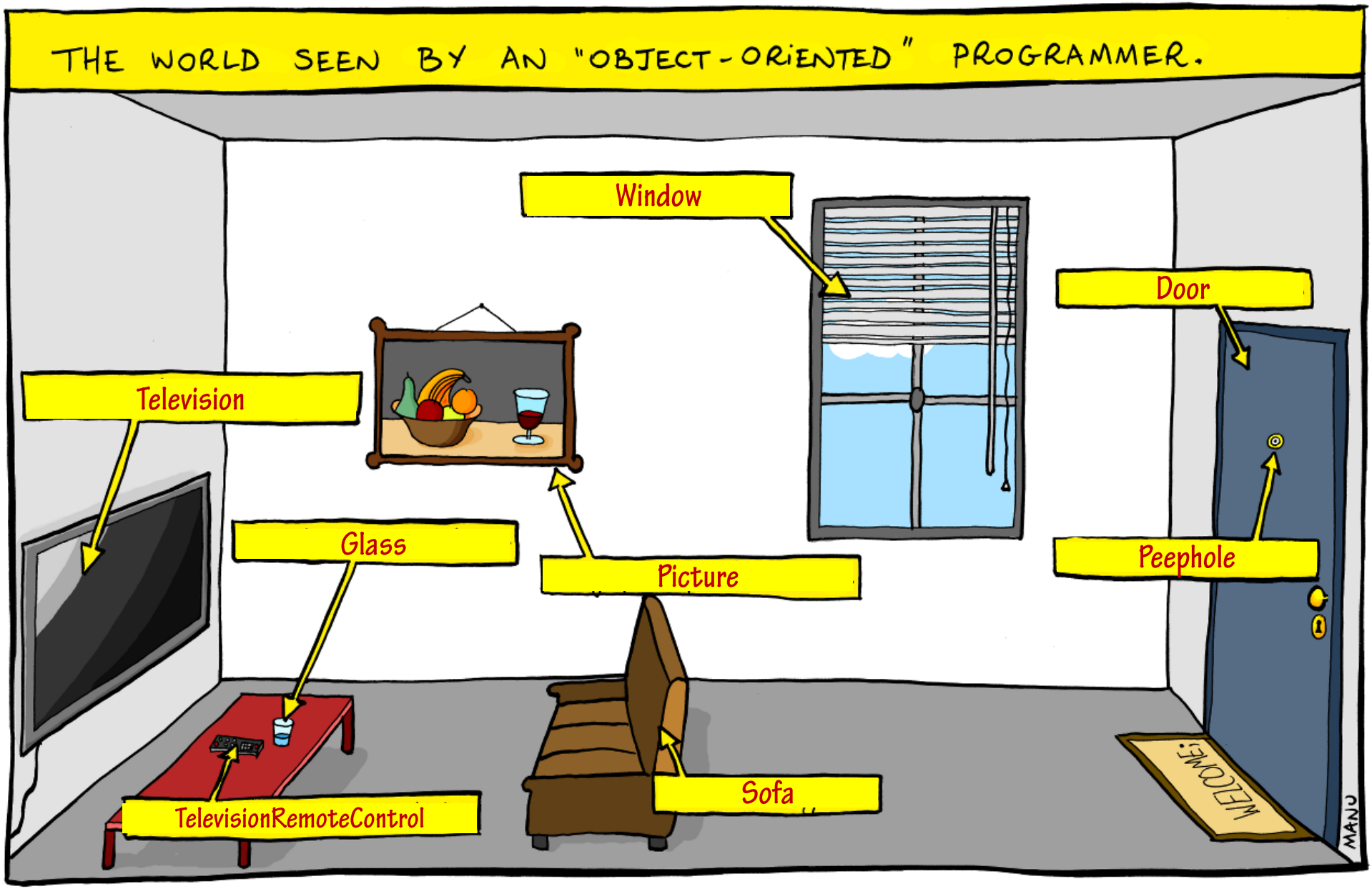
პროგრამირება

Python

ლექცია 1: ობიექტზე ორიენტირებული დაპროგრამირების (OOP) პრინციპები და ძირითადი კონცეფციები; მარტივი კლასებისა და ობიექტების შექმნა; მონაცემთა აბსტრაქცია და ინკაპსულაცია

ლიკა სვანაძე

lika.svanadze@btu.edu.ge

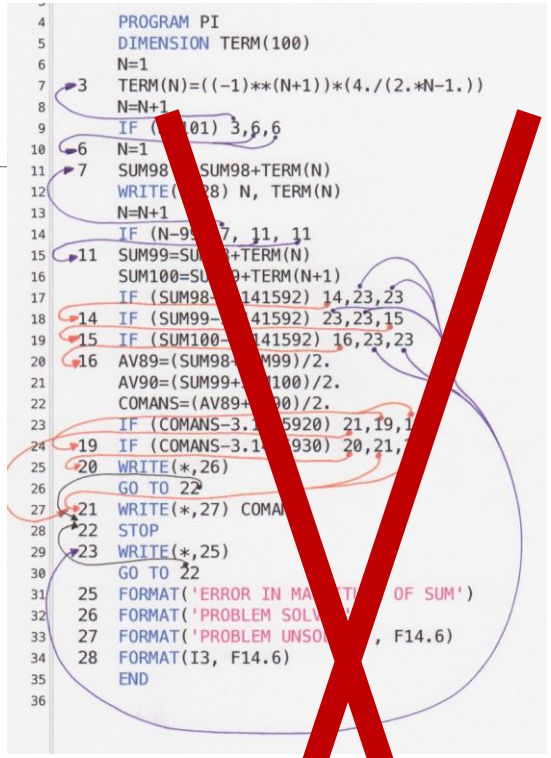

THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
Window
Door
Television
Glass
Picture
Peephole
TelevisionRemoteControl
Sofa
WELCOME
MANU

OOP - Object Oriented Programming

OOP - ობიექტზე ორიენტირებული პროგრამირება არის მიდგომა (მეთოდოლოგია), რომლის მეშვეობით პროგრამის შემუშავება ხორციელდება **კლასებისა** და **ობიექტების** გამოყენებით.

რატომ OOP?

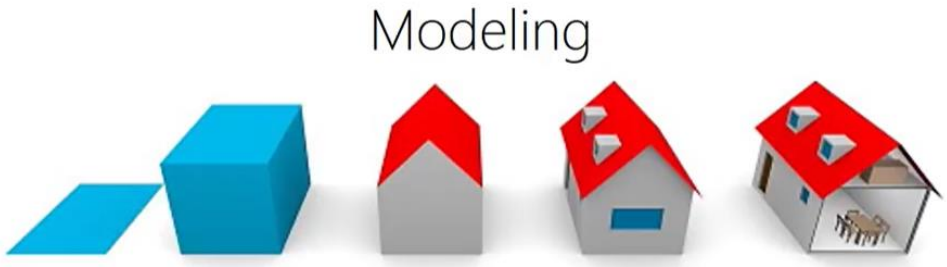
* სტრუქტურული და მოდულებად დაყოფილი კოდი
* რეალურ გარემოზე (პროექტზე) მორგებული
* მარტივად ცვლილებების განხორციელება
* ახალი მოდულების მარტივად ინტეგრირება



Spagetti Code

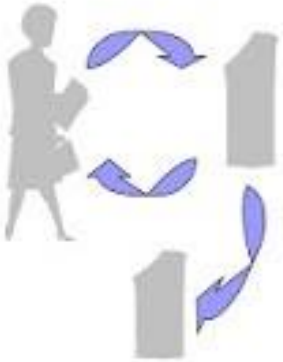
KIS (Keep It Simple)

DRY (Don’t Repeat Yourself)



Procedural vs. Object-Oriented

- Procedural

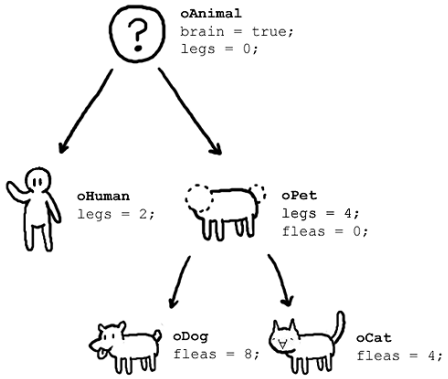


Withdraw, deposit, transfer

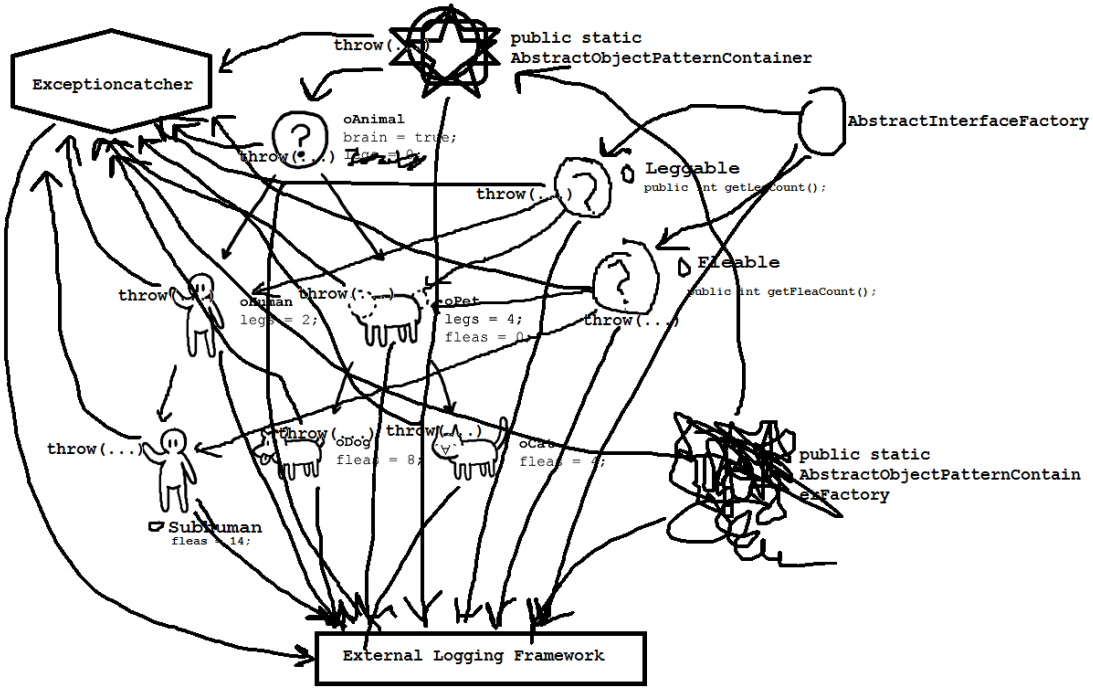
- Object Oriented

Customer, money, account

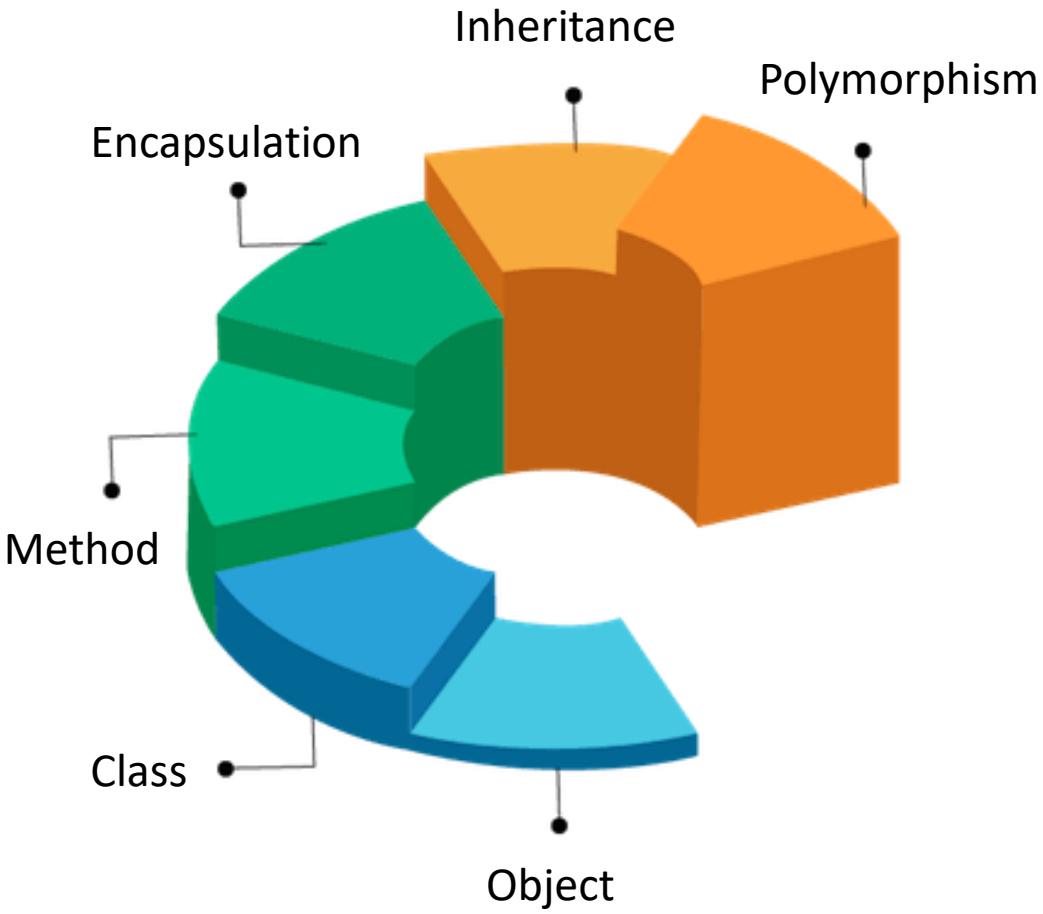
What OOP users claim



What actually happens



Python OOP system



Class, Object

- პითონში გვაქვს სხვადასხვა ტიპის **მონაცემები** (data), მაგ.

```
142        3.14          "Hello"        [2, 5, 7, 1]     {1:1, 9:81, 7:49}
```

- მონაცემები იგივეა რაც **ობიექტები**, რომელსაც აქვს **ტიპი** (int, float, str, list, სხვ.)

- მაგ. 142 არის **int ტიპის მონაცემი**

  "Hello" არის **str ტიპის მონაცემი**

  numbs = [2, 5, 7, 1] - numbs არის list ტიპის მონაცემი (ობიექტი)

- პითონში არსებული **ტიპები** წარმოადგენს **კლასს**, ხოლო **მონაცემები** - ამ კლასის ობიექტს.

```
>>> type(3.14)
<class 'float'>
>>> type(7)
<class 'int'>
>>> type("Hi")
<class 'str'>
```

Class, Object

* კლასი (Class) გამოიყენება ახალი ტიპის ობიექტების აღსაწერად და მათზე მოქმედებების ჩასატარებლად. იგი განზოგადებულად აღწერს გარკვეული ტიპის ობიექტების მახაისათებლებს (ატრიბუტებს) და ფუნქციონალს (მეთოდებს). ტერმინები - **კლასი (Class)**, **ტიპი (Type)**, **მონაცემთა ტიპი (Data type)** გამოიყენება ერთი და იგივე მნიშვნელობით. მაგ. `int, dict, str` წარმოადგენს კლასს (ტიპს, მონაცემთა ტიპს).

* ობიექტი (Object) წარმოადგენს კლასის კონკრეტულ შემთხვევას (instance). ობიექტი აუცილებლად უნდა ეკუთვნოდეს რომელიმე კლასს. ტერმინები - ობიექტი (**Object**) და ეგზემპლარი (**instance**) გამოიყენება ერთი და იგივე მნიშვნელობით.

Class, Object

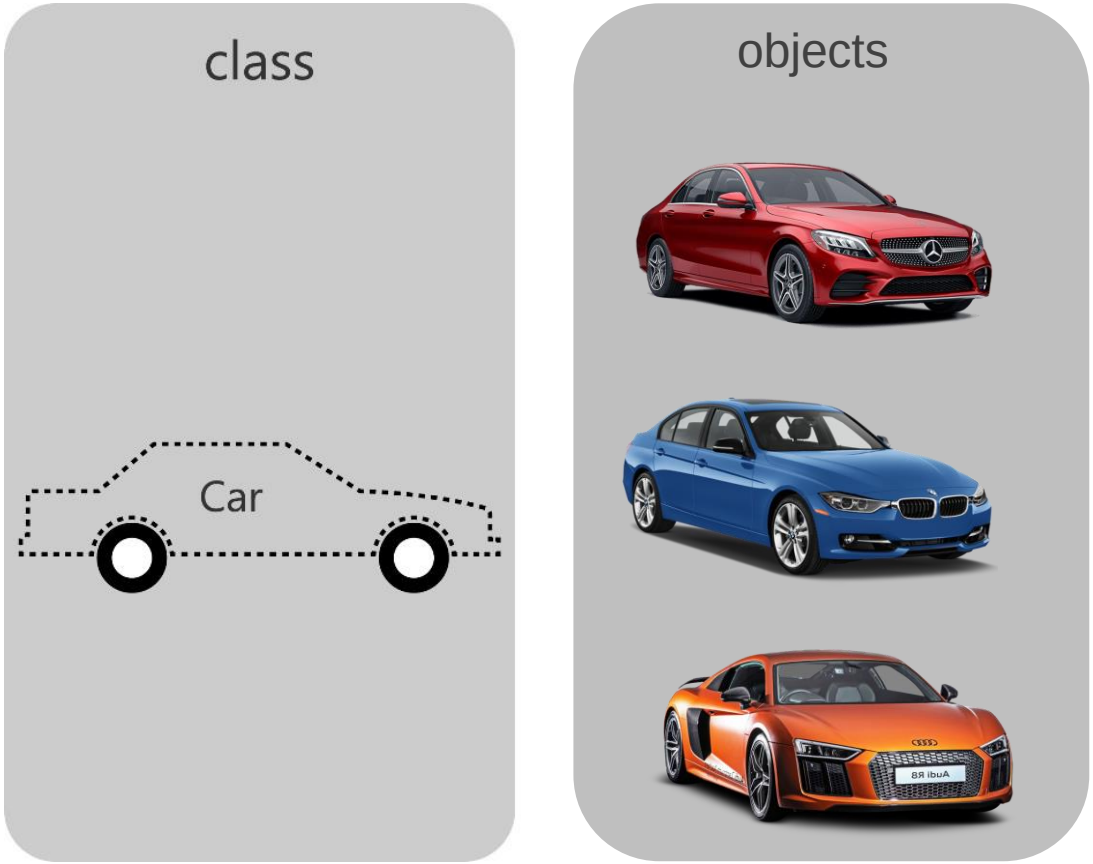
* კლასს გააჩნია **ატრიბუტი** და **მეთოდი (ფუნქცია)**. ატრიბუტი წარმოადგენს კლასის მახასიათებლებს (თვისებებს), ხოლო მეთოდი აღწერს გარკვეულ მოქმედებას/ქცევას (behavior), რომელიც ამ ტიპის ობიექტისთვის არის დამახასიათებელი.

* მაგ. `numbs = [2, 5, 7, 1].` list ტიპის ობიქტისთვის შესაძლებელია გამოვიყენოთ მეთოდი (ფუნქცია) append, count და სხვ.

მაგ. `numbs.count(1)` - ამ შემთხვევაში count არის list კლასის მეთოდი, რომელიც მხოლოდ list ტიპის ობიექტებთან (მონაცემებთან) მუშაობს.

* გარდა ჩაშენებული კლასებისა, პროგრამისტს შეუძლია თავად შექმნას სასურველი კლასი, თავად განსაზღვროს კლასის საჭირო ატრიბუტები და მეთოდები.

Class & Object



* კლასს გააჩნია **ატრიბუტი** და **მეთოდი (ფუნქცია)**. ატრიბუტი წარმოადგენს კლასის მახასიათებლებს (თვისებებს), ხოლო მეთოდი აღწერს გარკვეულ მოქმედებას/ქცევას (behavior), რომელიც ამ ტიპის ობიექტისთვის არის დამახასიათებელი.

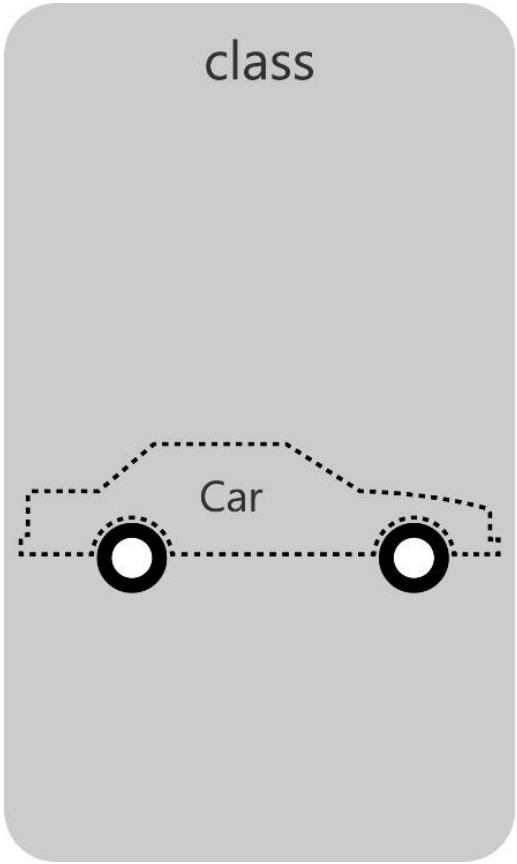
Class & Object

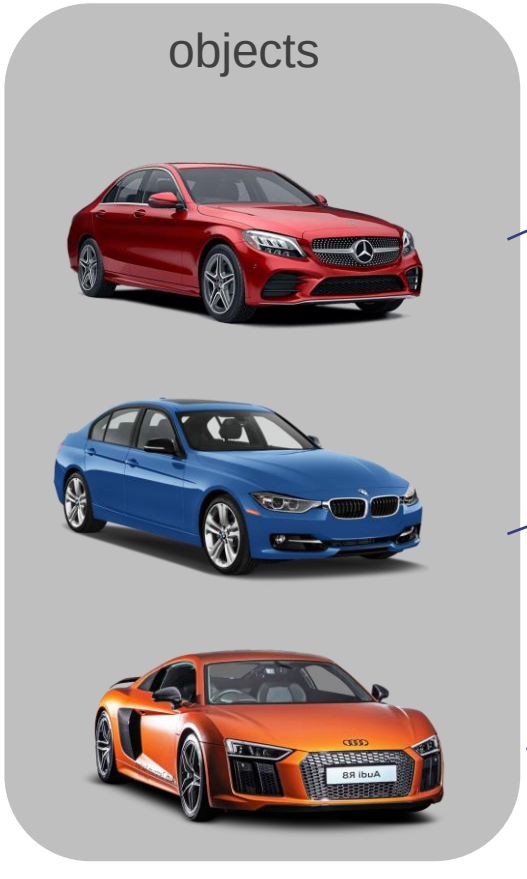
Attributes:

color
model
makeYear
fuelType

Methods:

sell()
buy()
rent()
insurance()





red
Mercedes
2015
Gas

blue
BMW
2013
Gas

orange
Audi
2009
Diesel

* კლასს გააჩნია **ატრიბუტი** და **მეთოდი (ფუნქცია)**. ატრიბუტი წარმოადგენს კლასის მახასიათებლებს (თვისებებს), ხოლო მეთოდი აღწერს გარკვეულ მოქმედებას/ქცევას (behavior), რომელიც ამ ტიპის ობიექტისთვის არის დამახასიათებელი.

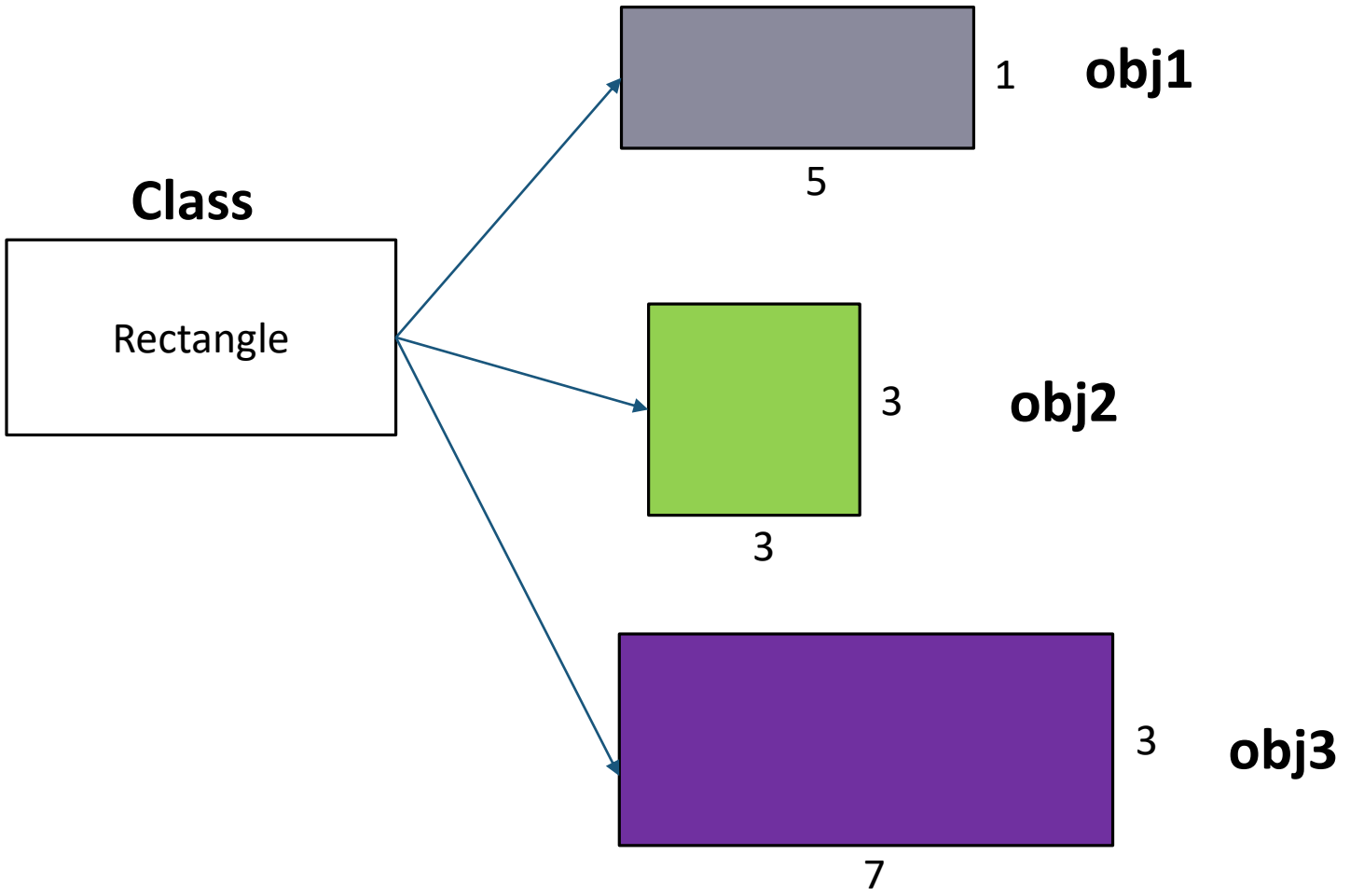
Class & Object

Attributes:

width
length
color

Methods:

perimeter()
area()



Class-ის აღწერა

* **მაგალითი**: შექმენით კლასი Rectangle ატრიბუტებით სიგრძე და სიგანე.
* კლასის აღწერისას გამოიყენება რეზერვირებული სიტყვა class.
* მსგავსად ფუნქციისა, კლასში შესაძლებელია იყოს docstring (ინფორმაცია კლასის შესახებ), რომლის წაკითხვა შესაძლებელია __doc__ ატრიბუტის მეშვეობით. დეტალური ინფორმაცია იხილეთ მომდევნო სლაიდებზე.

```
class Rectangle:
    '''Docstring text here'''

    def __init__(self, width, length):
        self.width = width
        self.length = length
```

* კლასებისთვის არსებობს წინასწარ განსაზღვრული **განსაკუთრებული მეთოდები (special methods)**, რომელთა დასახელება იწყება და მთავრდება ორი ქვედა ტირით. მაგ. `__init__()`, `__str__()`, `__sub__()` და სხვა.
* მეთოდი __init__() წარმოადგენს **კონსტრუქტორს**, რომლის შესრულება ხდება ავტომატურად, როდესაც ამ კლასის ტიპის ობიექტი შეიქმნება. კონსტრუქტორის დანიშნულებაა კლასის ატრიბუტებისთვის საწყისი მნიშვნელობების მინიჭება. __init__() მეთოდს პარამეტრად გადაეცემა self, რომელიც მიუთითებს ობიექტს რომლისთვისაც ხდება ამ მეთოდის გამოყენება. self-ის ანალოგი C++-ში არის this.

Object-ის შექმნა

* კლასის აღწერის შემდეგ, საჭიროა კლასის ტიპის ობიექტის შექმნა (კლასის გარეთ).

```
obj1 = Rectangle(3,5)
print(type(obj1))
print(obj1.width)
print(obj1.length)
```

შედეგი:

```
<class '__main__.Rectangle'>
3
5
```

* შექმენით Rectangle კლასის მეორე ობიექტი სახელწოდებით obj2.

method-ის შექმნა

- მეთოდი - წარმოადგენს ფუნქციას, რომელიც მოთავსებულია (აღწერილია) კლასში. იგი მუშაობს ზუსტად ისე, როგორც ფუნქცია ერთი განსხვავებით: მეთოდის პირველი არგუმენტი ყოველთვის არის self - ანუ არსებული კლასის კონკრეტული ობიექტი (instance).
- აღვწეროთ იმავე კლასში perimeter მეთოდი, რომელიც გამოითვლის ოთხკუთხედის პერიმეტრს:

შესრულება

```
class Rectangle:
    ''' Docstring text here'''
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def perimeter(self):
        return 2*(self.width+self.length)
```

მეთოდის აღწერა

method-ის გამოძახება

* გამოვთვალოთ obj1-ის პერიმეტრი.

შესრულება

```
class Rectangle:
    ''' Docstring text here'''
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def perimeter(self):
        return 2 * (self.width + self.length)



obj1 = Rectangle(3,5)

print(obj1.perimeter())

print(Rectangle.perimeter(obj1))
```

შედეგი:

```
16
16
```

კლასის მეთოდის გამოძახება შესაძლებელია ორნაირად

ატრიბუტის ან ობიექტის წაშლა

- ატრიბუტის ან ობიექტის წაშლა შესაძლებელია del ბრძანებით.

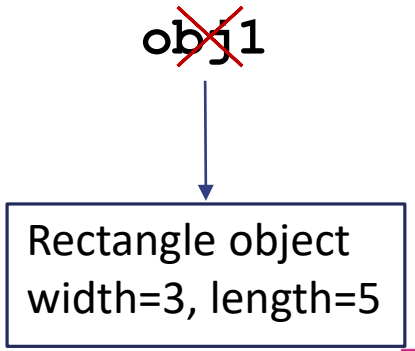
```
class Rectangle:
    ''' Docstring text here'''
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def perimeter(self):
        return 2 * (self.width + self.length)

obj1 = Rectangle(3,5)
print(obj1.length)
del obj1.length
print(obj1.length)
```

შედეგი:

```
5
AttributeError: 'Rectangle' object has
no attribute 'length'
```

- ობიექტის წაშლა ხორციელდება შემდეგნაირად:

```
del obj1
del Rectangle
```



ატრიბუტის მნიშვნელობის შეცვლა

* ობიექტის ატრიბუტის მნიშვნელობა შესაძლებელია შეიცვალოს კლასის გარეთ:

```
class Rectangle:
    ''' Docstring text here'''
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def perimeter(self):
        return 2 * (self.width + self.length)

obj1 = Rectangle(3,5)
obj1.width += 1
print(obj1.width)
```

შედეგი:

```
4
```

* უმჯობესია, თუ ატრიბუტის მნიშვნელობის შეცვლა მოხდება კლასში ახალი მეთოდის მეშვეობით:

```
class Rectangle:
    ''' Docstring text here'''
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def perimeter(self):
        return 2 * (self.width + self.length)

    def increase_width(self):
        self.width += 1

obj1 = Rectangle(3,5)
obj1.increase_width()

print(obj1.width)
```

შედეგი:

```
4
```

მეთოდის Default values (გაჩუმებით მნიშვნელობა)

* კლასში მეთოდის აღწერისას ცვლადებს შესაძლებელია მიეთითოს საწყისი მნიშვნელობა გაჩუმებით (default value). მაგ. __init__() მეთოდში, გაჩუმებით მინიჭების შემთხვევაში, ახალი ობიექტის შემოღებისას შეგვიძლია აღარ გადავცეთ საწყისი მნიშვნელობები. ქვემოთ განხილულ მაგალითში, თუ ობიექტის შემოღებისას არ გადავცემთ პარამეტრებს, ავტომატურად აღებული იქნება მნიშვნელობებად 1 და 2, ანუ მართკუთხედის სიგანე გაჩუმებით არის 1, ხოლო სიგრძე - 2.

მაგალითი

```
class Rectangle:
    def __init__(self, width=1, length=2):
        self.width = width
        self.length = length


obj1 = Rectangle()
print(obj1.width)
print(obj1.length)

obj2 = Rectangle(3,7)
print(obj2.width)
print(obj2.length)
```

შედეგი:

```
1
2
3
7
```

__str__() მეთოდი

* **__str__()** მეთოდი წარმოადგენს კლასის წინასწარ განსაზღვრულ ფუნქციას, რომელიც აბრუნებს თქვენთვის სასურველ ტექსტს (სტრიქონს) ობიექტის შესახებ. მისი შესრულება ხდება ავტომატურად მაშინ, როდესაც მივმართავთ (ვბეჭდავთ) კონკრეტულ ობიექტს (ატრიბუტების გარეშე).

* **მაგალითი**: შექმენით Point კლასი კოორდინატთა სისტემაზე ნებისმიერი წერტილის მდებარეობის აღსაწერად. მოახდინეთ მისი ინიციალიზაცია, რომელსაც გაჩუმებით გადაეცემა კოორდინატთა სათავე. დაამატეთ **__str__()** მეთოდი, რომელიც ნებისმიერ ობიექტს წარმოადგენს შემდეგნაირად: (x,y). მაგ: (5,3). კლასის გარეთ, შემოიტანეთ ორი ახალი ობიექტი, თქვენთვის სასურველი კოორდინატებით. *შენიშვნა: **format** ფუნქციის სინტაქსი იხილეთ მომდევნო სლაიდზე.*

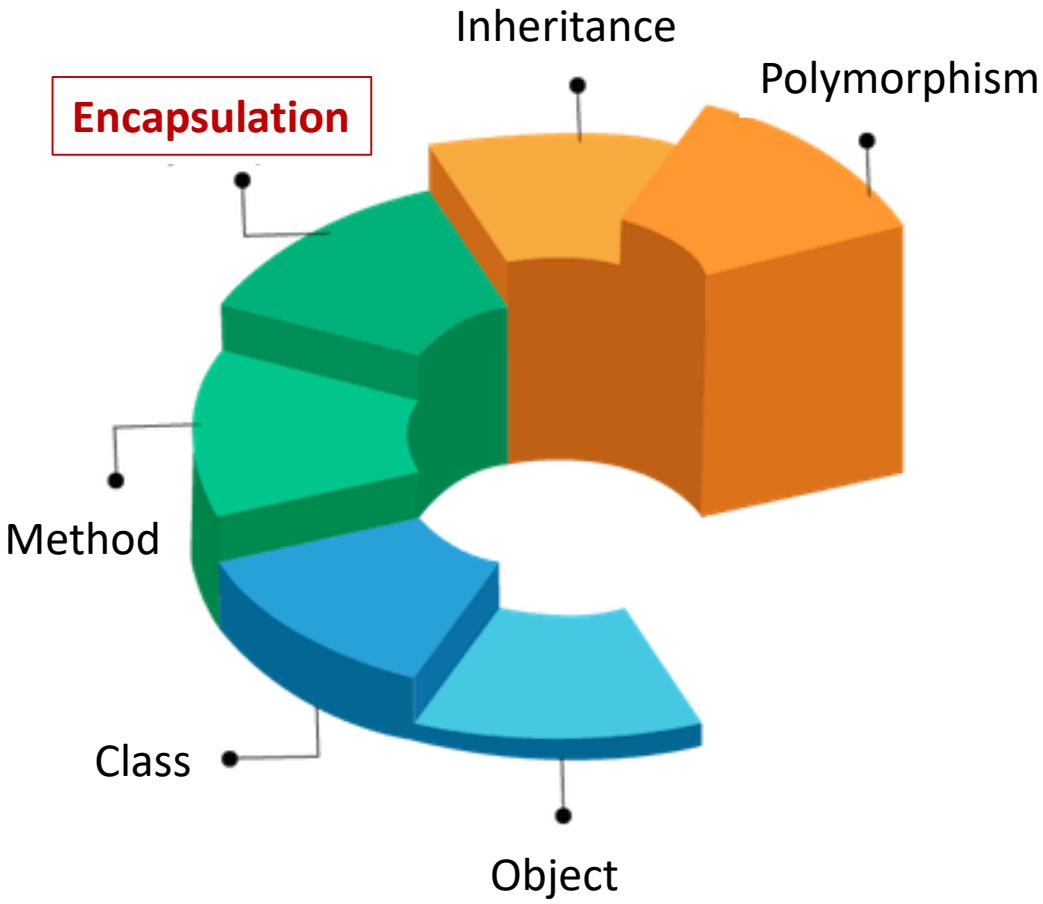
მაგალითი

```
class Point:
    def __init__(self, x=0, y=0):
        self.x=x
        self.y=y

    def __str__(self):
        return "({},{})".format(self.x,self.y)

p1 = Point(4,3)
print(p1)
p2 = Point()
print(p2)
```

შედეგი:

```
(4,3)
(0,0)
```

Python OOP system



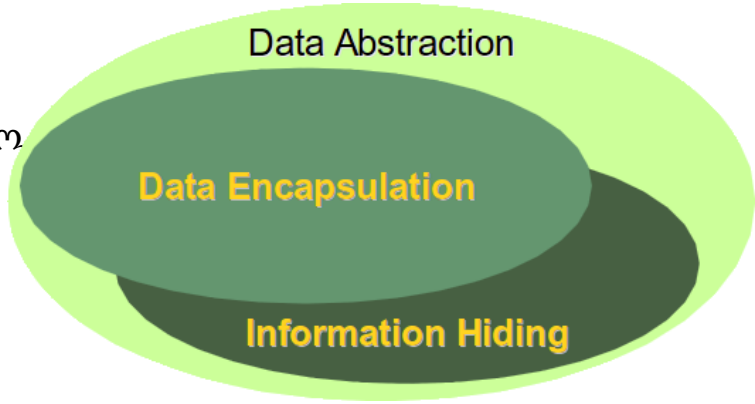
Abstraction, Encapsulation, Hiding

ტერმინები **Data Abstraction**, **Data Encapsulation** და **Information Hiding** ზოგჯერ ერთი და იგივე მნიშვნელობით გამოიყენება. თუმცა, მათ შორის არის მნიშვნელოვანი განსხვავება.

მონაცემთა აბსტრაქცია გულისხმობს **მონაცემთა ინკაფსულაციასა** და გარკვეული ინფორმაციის **დამალვას (Hiding)** ერთდროულად.

ინკაფსულაცია წარმოადგენს მონაცემების შეკვრას (კაფსულირებას) მეთოდების გამოყენებით.

პროექტზე მუშაობის დროს აუცილებლად უნდა გაითვალისწინოთ სამივე კომპონენტი, რომ საბოლოო პროდუქტი იყოს მაქსიმალურად მორგებული მომხარებლის საჭიროებაზე და იყოს მაქსიმალურად user-friendly.



Abstraction = Encapsulation + Hiding

Information hiding - არის მიდგომა, რომლის დროსაც კლასის გარკვეული მონაცემები არ არის ხილვადი კლასის გარეთ, რაც ნიშნავს რომ მათი გარედან შემთხვევით შეცვლა არ შეიძლება.

აბსტრაქცია - Abstraction

აბსტრაქცია წარმოადგენს ობიექტზე ორიენტირებული პროგრამირების მნიშვნელოვან ნაწილს. იგი გამოიყენება სხვადასხვა პროგრამირების ენაში, მათ შორის Python-შიც.

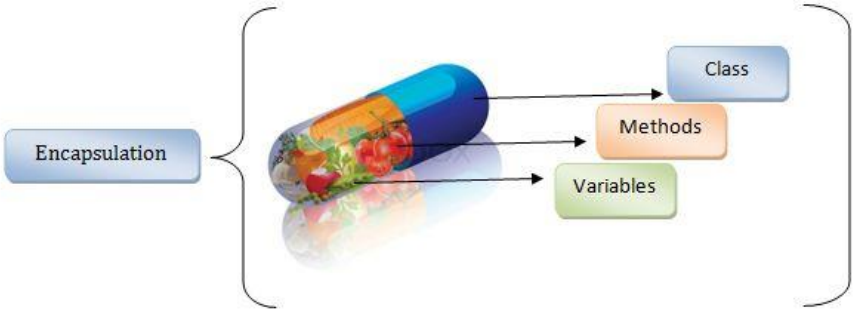
აბსტრაქცია არის მიდგომა (კონცეფცია), რომლის დროსაც მომხმარებელი ინფორმირებულია თუ როგორ მუშაობს კონკრეტული სისტემა, და არ არის საჭირო დეტალურად იცოდეს თუ რა ნაწილებისგან შედგება იგი. შესაბამისად, მხოლოდ საჭირო და მნიშვნელოვან ფუნქციონალზე ხდება ფოკუსირება, ხოლო სხვა დანარჩენი შიდა კომპონენტი რომელიც არ არის მომხმარებლისთვის მნიშვნელოვანი, დამალულია.

Real Life Example of Abstraction

ინკაფსულაცია - Encapsulation

ინკაფსულაცია არის მექანიზმი, რომლის მეშვეობითაც ხდება აბსტრაქციის მიდგომის გამოყენება.
ინკაფსულაცია გულისხმობს გარკვეული ნიშნით დაკავშირებული კომპონენტების თავმოყრას ერთ გარემოში, როგორც ერთიანი სისტემა. კლასები წარმოადგენს ინკაფსულაციის ყველაზე კარგ მაგალითს.
ინკაფსულაცია მიიღწევა კლასის ატრიბუტებზე 2 სახის მეთოდის გამოყენებით: **getter** და **setter**.

მეთოდები, რომლებიც აბრუნებენ ატრიბუტების მნიშვნელობებს, უწოდებენ **getter** მეთოდებს. ხოლო მეთოდები, რომლებიც ცვლიან ატრიბუტების მნიშვნელობას, უწოდებენ **setter** მეთოდებს.



getter და setter მეთოდები

ატრიბუტებზე პირდაპირი წვდომა და მათი მნიშვნელობების ცვლილება შესაძლებელია, თუმცა სასურველია იგი მოხდეს **getter** და **setter** მეთოდების გამოყენებით.

მაგალითი

```
class Student:

    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    def get_name(self):
        return self.firstname

    def set_name(self, text):
        self.firstname = text

st1 = Student("Giorgi", "Abashidze")
print(st1.firstname) # პირდაპირი წვდომა ცვლადზე - არაა რეკომენდირებული
print(st1.get_firstname()) # ცვლადზე წვდომა getter მეთოდით
st1.firstname = "Davit"  # ცვლადის მნიშვნელობის შეცვლა მინიჭებით- არაა რეკომენდირებული
st1.set_firstname("Davit") # ცვლადის მნიშვნელობის შეცვლა setter მეთოდით
```

getter მეთოდი

setter მეთოდი

*მას შემდეგ რაც, განვსაზღვრავთ **getter** და **setter** მეთოდებს კონკრეტული ატრიბუტისთვის, სასურველია ატრიბუტი გავხადოთ Private (მიუწვდომელი კლასის გარედან). წვდომის რეჟიმები იხილეთ შემდეგ სლაიდზე.*

ინფორმაციის დამალვა -Information Hiding

აბსტრაქციის კონცეფცია მოიცავს ინფორმაციის დამალვასაც, რაც გულისხმობს, რომ კლასის გარკვეული ცვლადები და მეთოდები არ უნდა იყოს წვდომადი კლასის გარედან, რომ არ მოხდეს მათი შემთხვევით ცვლილება. ობიექტზე ორიენტირებულ პროგრამირებაში არსებობს ატრიბუტებზე წვდომის 3 რეჟიმი: Public, Protected, Private. Python-ში არ ხდება სიტყვების (Public, Protected ან Private) მითითება ცვლადის/მეთოდის წინ. Python-ში ატრიბუტის სახელის წინ ეთითება ერთი ან ორი "ქვედა ტირე", რომლის მეშვეობითაც ხდება წვდომის ტიპის განსაზღვრა.

Access Modifiers - კლასის ელემენტებზე წვდომა

ატრიბუტის/მეთოდის დასახელება	წვდომის ტიპი	აღწერა
name	Public	ატრიბუტი/მეთოდი წვდომადია ნებისმიერი ადგილიდან (კლასის შიგნით და გარეთ).
_name	Protected	ატრიბუტი/მეთოდი წვდომადია მხოლოდ არსებულ კლასში და მის მემკვიდრე კლასებში. ***შენიშვნა:*** *Python-ში პირდაპირ წვდომა მაინც არის ატრიბუტზე შესაძლებელი კლასის გარეთ. პროგრამისტმა უნდა გაითვალისწინოს, რომ ცვლადზე/მეთოდზე წვდომა/მოდიფიკაცია, რომელიც იწყება ქვედა ტირით არ არის მიზანშეწონილი, რადგან იგი მიღებულია რომ არის Protected ტიპის.*
__name	Private	ატრიბუტი წვდომადია მხოლოდ ამავე კლასში. იგი კლასის გარეთ არ ჩანს და მასზე წვდომა კლასის გარედან არ არის შესაძლებელი

ინფორმაციის დამალვა - Information Hiding

Python-ში გაჩუმებით ყველა ცვლადი არის Public ტიპის.

მაგალითი

```
class Myclass():
    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"


a = Myclass()

a.pub += " and I can be modified"
print(a.pub)

print(a.__priv)   #გამოიწვევს AttributeError-ს

a._prot += ', but still can be changed'
print(a._prot)
```

შედეგი:

```
I am public and I can be modified

AttributeError: 'Myclass' object has no
attribute '__priv'

I am protected, but still can be changed
```

property() მეთოდი (1)

property() მეთოდი შეიცავს getter, setter, deleter მეთოდებს რომლის მეშვეობითაც შესაძლებელია კლასის ცვლადებზე მუშაობა. property() მეთოდი ქმნის ახალ ატრიბუტს (თვისებას), რომლის გამოყენებით ხდება ამ ატრიბუტის (თვისების) getter და setter მეთოდებზე წვდომა.

property() მეთოდი უნდა აღიწეროს კლასის შიგნით და მისი სინტაქსია:

```
name = property(getter, setter, deleter)
```

- ობიექტის თვისების სახელი, რომლითაც იქნება გამოძახებული
- ფუნქციის სახელი, რომლის შესრულება მოხდება თვისების გამოძახებისას
- ფუნქციის სახელი, რომლის შესრულება მოხდება თვისებაზე მინიჭების ოპერაციის დროს
- ფუნქციის სახელი, რომლის შესრულება მოხდება თვისებაზე del ოპერაციის გამოყენებისას (არასავალდებულო ველი)

მაგალითი იხილეთ მომდევნო სლაიდზე. თვისებებთან სამუშაოდ შესაძლებელია @property დეკორატორის გამოყენება (იხილეთ მომდევნო სლაიდებზე).

property() მეთოდი (2)

მაგალითი

```
class Student:
    def __init__(self, firstname, lastname):
        self.__firstname = firstname
        self.__lastname = lastname

    def get_name(self):
        return self.__firstname

    def set_name(self, text):
        self.__firstname = text

    def del_name(self):
        del self.__firstname

    name = property(get_name, set_name, del_name)

st1 = Student('Giorgi', 'Arveladze')

print(st1.name)

st1.name = "Davit"

del st1.name
```

name თვისების შექმნა. კლასის გარეთ წვდომა შესაძლებელი იქნება name სახელწოდებით

print(st1.name) — გამოიძახებს get_name() მეთოდს.

st1.name = "Davit" — გამოიძახებს set_name() მეთოდს.

del st1.name — გამოიძახებს del_name() მეთოდს.

შედეგი:

```
Giorgi
```

სლაიდები რომლებიც არის ცისფერი ფონის არის დამატებითი ინფორმაცია, რომელიც არ შედის ქვიზებში და გამოცდის საკითხებში. ინტერესის შემთხვევაში შეგიძლია გაარჩიოთ ☺

მაგალითი (ნაწილი 1)

მაგალითი 1: შექმენით კლასი Student, ატრიბუტებით first, last, fullname, email.

```
class Student:

    def __init__(self, first, last):
        self.first = first
        self.last = last
        self.fullname = self.first + ' '+ self.last
        self.email ='{}.{}@btu.edu.ge'.format(first,last)


st1 = Student("Mariam", "Abuladze")
print(st1.first)      #Output: Mariam
print(st1.fullname)   #Output: Mariam Abuladze
st1.first = "Nino"
print(st1.first)      #Output: Nino
print(st1.fullname)   #Output: Mariam Abuladze
```

first ცვლადის მნიშვნელობის შეცვლის შემდეგ, fullname-ში დარჩა მაინც ძველი სახელი, რადგან fullname-ის ინიციალიზაცია მოხდა როდესაც st1 ობიექტი შემოვიღეთ. თუ კლასში გვჭირდება ისეთი ატრიბუტები, რომლებიც სხვა ცვლადის მნიშვნელობებისგან არის შედგენილი (მაგ. fullname, email), უმჯობესია ისინი გარდავქმნათ როგორც მეთოდი, რომელიც დააგენერირებს შესაბამის მნიშვნელობას (იხილეთ შემდეგი სლაიდი).

მაგალითი (ნაწილი 2)

```
class Student:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    def fullname(self):
        return '{} {}'.format(self.first, self.last)

    def email(self):
        return '{}.{}@btu.edu.ge'.format(self.first, self.last)

st1 = Student("Mariam", "Abuladze")
print(st1.fullname())
print(st1.email())
```

მაგალითში fullname და email წარმოდგენილია როგორც მეთოდი და აღარ გვაქვს ატრიბუტების სახით. შესაბამისად მათი გამოძახება უნდა მოხდეს ()-ის მეშვეობით. თუმცა, fullname და email შინაარსობრივად წარმოადგენს კლასის დამახიათებელ კომპენენტს და საურველია რომ მისი გამოძახებისას არ მოგვიწიოს მუდმივად ()-ის გამოყენება. ამ შემთხევავაში ვიყენებთ @property დეკორატორს (იხილეთ შემდეგი სლაიდი).

მაგალითი (ნაწილი 3)

```
class Student:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def fullname(self):
        return '{} {}'.format(self.first, self.last)

    @property
    def email(self):
        return '{}.{}@btu.edu.ge'.format(self.first, self.last)


st1 = Student("Mariam", "Abuladze")
print(st1.fullname)
print(st1.email)
```

@property წარმოადგენს ჩაშენებულ დეკორატორს (ფუნქციას). @property დეკორატორის მეშვეობით კლასის მეთოდის გამოძახება შესაძლებელია როგორც ატრიბუტი, ანუ ()-ის გარეშე. ამ შემთხვევავაში fullname და email მეთოდი წარმოადგენს კლასის თვისებას (property) და მისი გამოძახება ხდება როგორც ატრიბუტის (ფრჩხილების გარეშე). დეტალური ინფორმაცია დეკორატორის შესახებ იხილეთ მომდევნო სლაიდებზე. property() ფუნქცია/მეთოდი დეტალურად აღწერილია მომდევნო სლაიდზე.

მაგალითი (ნაწილი 4)

property setter

დავუშვათ რომ იგივე მაგალითში გვსურს, fullname გამოვიყენოთ მონაცემების შესაცვლელად შემდეგნაირად (რაც გულისხმობს სახელის და გვარის ცვლილებას):

```
st1.fullname = ”Manana Manjgaladze"
```

თუმცა ამ ბრძანების გამოყენება არ შეგვიძლია რადგან fullname რეალურად ჩაწერილია როგორც მეთოდი. @property დეკორატორს აქვს setter და getter ფუნქციების მხარდაჭერა, რაც ნიშნავს რომ შეგვიძლია დავამატოთ ახალი მეთოდი რომელსაც მივუთითებთ **setter დეკორატორს** - **@propertyname.setter** , და შექმნილი მეთოდი გაეშვება fullname-ზე მინიჭების ოპერაციის გამოყენებისას (იხილეთ მაგალითის შედეგი შემდეგ სლაიდზე):

მაგალითი (ნაწილი 5)

property setter

```
class Student:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def fullname(self):
        return '{} {}'.format(self.first, self.last)

    @fullname.setter
    def fullname(self, name):
        result = name.split()
        self.first = result[0]
        self.last = result[1]


    @property
    def email(self):
        return '{}.{}@btu.edu.ge'.format(self.first, self.last)


st1 = Student("Mariam", "Abuladze")
st1.fullname = "Manana Manjgaladze"
print(st1.first)    # Output: Manana
print(st1.last)     # Output: Manjgaladze
```

შედეგი:

```
Manana
Manjgaladze
```

კლასის სტატიკური ცვლადი

- კლასის აღწერისას შესაძლოა ატრიბუტები ცვლადის სახით იყოს წარმოდგენილი __init__() კონსტრუქტორის გარეთ. ასეთ ცვლადს უწოდებენ სტატიკურ ცვლადს, რომელიც საერთოა ყველა ამ კლასის ობიექტისთვის. იგი უფრო მეტადი ასოცირებულია კლასთან, ვიდრე კონრეტულ ობიექტთან.
- ქვემოთ მითითებულ მაგალითში uni ატრიბუტის მნიშვნელობა ყველა ობიექტისთვის (სტუდენტისთვის) იქნება 'BTU'. უმეტსად, სტატიკური ცვლადის გამოყენება/გამოძახება ხდება კლასისთვის და არა ობიექტისთვის (instance).

```
class Student:
    uni = 'BTU'

    def __init__(self, first, last):
        self.first = first
        self.last = last

    def __str__(self):
        return '{} {} სწავლობს {}-ში'.format(self.first, self.last, Student.uni)

st1 = Student('Giorgi', 'Abashidze')
print(st1.uni)
print(Student.uni)
print(st1)
```

შედეგი:

```
BTU
BTU
Giorgi Abashidze სწავლობს BTU-ში
```

სტატიკური მეთოდი - static method

კლასის სტატიკური მეთოდი არის სტანდარტული ფუნქცია, რომელიც უმეტესად დაკავშირებულია კლასთან და არა ობიექტთან. შესაბამისად, მისი მეშვეობით არ ხდება ობიექტის ცვლადებზე მანიპულაციები. მისი გამოძახება უმეტესად ხდება კლასის სახელის მეშვეობით. იგი უმეტესად გამოიყენება utility ფუნქციების დასაწერად ანუ ხშირად გამოყენებადი ფუნქციები, რომლებიც ობიექტებთან არ არის დაკავშირებული. მაგ. მონაცემის ერთი ტიპიდან მეორე ტიპში კონვერტაციის ფუნქცია. ესეთი ფუნქციები შეიძლება დაიწეროს კლასის გარეთაც, თუმცა უმჯობესია რომელიმე კლასთან იყოს ასოცირებული სადაც ხდება მისი გამოყენება.

კლასის სტატიკური მეთოდის აღწერა ხდება კლასის შიგნით და მას პარამეტრად **არ** გადაეცემა self ობიექტი. static method-ის გამოძახება შესალებელია ობიექტის შემოღების გარეშეც.

სტატიკური მეთოდის შექმნა შესაძლებელია ორი გზით:

1. staticmethod() ჩაშენებული ფუნქციით
2. @staticmethod ჩაშენებული დეკორატორით

განვიხილოთ შესაბამისი მაგალითები მომდევნო სლაიდზე:

სტატიკური მეთოდი - static method

@staticmethod დეკორატორი

```
class Student:
    uni = 'BTU'

    def __init__(self, first, last):
        self.first = first
        self.last = last


    def get_name(self):
        return Student.upper_text(self.first)

    @staticmethod
    def upper_text(text):
        return text.upper()

st1 = Student('Giorgi', 'Abashidze')

print(st1.get_name())
print(Student.upper_text("Hello Everyone"))
print(st1.upper_text("Hello Everyone"))
```

staticmethod ფუნქცია

```
class Student:
    uni = 'BTU'

    def __init__(self, first, last):
        self.first = first
        self.last = last

    def get_name(self):
        return Student.upper_text(self.first)

    def upper_text(text):
        return text.upper()

Student.upper_text = staticmethod(Student.upper_text)

st1 = Student('Giorgi', 'Abashidze')

print(st1.get_name())
print(Student.upper_text("Hello Everyone"))
print(st1.upper_text("Hello Everyone"))
```

შედეგი:

```
GIORGI
HELLO EVERYONE
HELLO EVERYONE
```

format() ფუნქცია

* format() ფუნქცია წარმოადგენს სტრინებთან სამუშაო ფუნქციას. მისი მეშვეობით შესაძლებელია სტრიქონში სხვადასხვა კომპონენტების ჩასმა და შედეგად მიიღება დაფორმატებული სტრიქონი. format() ფუნქციის სინტაქსია:

* str წარმოადგენს სტრიქონს, რომელიც ასევე შეიცავს { } სიმბოლოებს. შედეგად, { }-ის ნაცვლად ჩაიწერება კონკრეტული მნიშვნელობა (value), რომელიც მითითებულია format ()-ში არგუმენტების სახით.

მაგალითი

```
name = "Adam"
value = 2.35

print("Hello {}, your balance is {}".format(name, value))
```

სტრიქონი ფორმატირება

შედეგი:

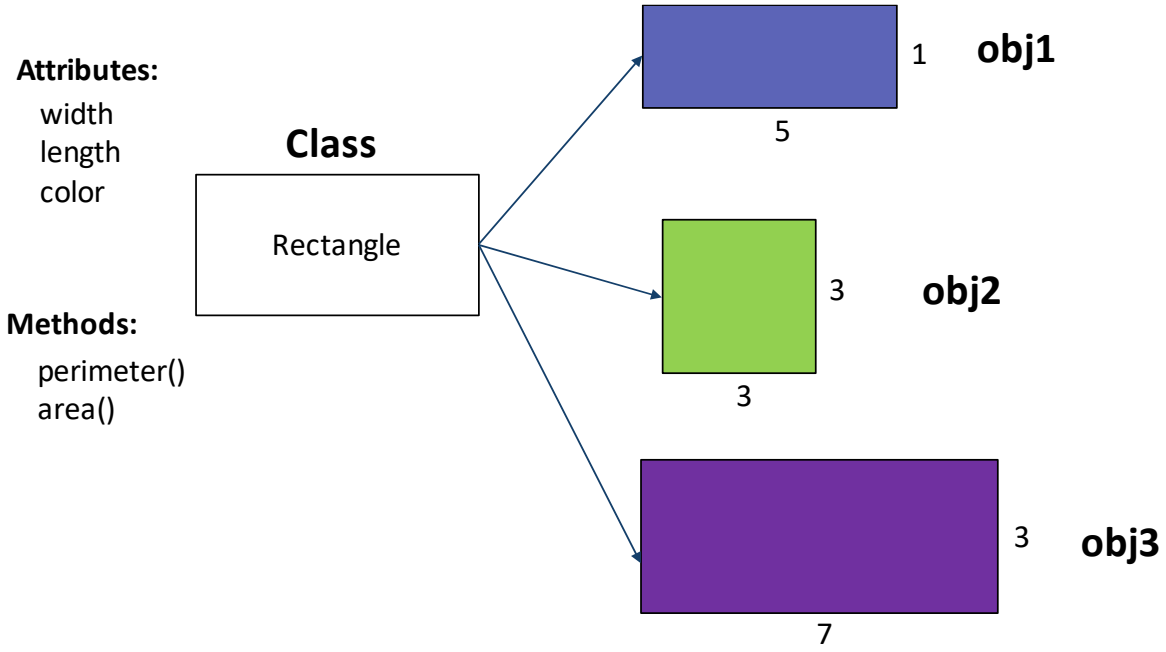
```
Hello Adam, your balance is 2.35
```

dir, help, __doc__

* `dir` ფუნქცია აბრუნებს პარამეტრად გადაცემული ობიექტის ატრიბუტებისა და მეთოდების ჩამონათვალს. მაგ. `print(dir(obj1))`
* `help` ფუნქცია აბრუნებს პარამეტრად გადაცემული ობიექტების ატრიბუტებისა და მეთოდების ჩამონათვალს თავისი აღწერილობით. მაგ. `print(help(obj1))` ან `print(help(obj1.perimeter))`
* `__doc__` ატრიბუტის მეშვეობით შესაძლებელია კლასის ან მეთოდის `docstring`-ის დაბრუნება. მაგ. `print(obj1.__doc__)`. თუ docstring რამდენიმე ხაზზე უნდა დაიწეროს, ამ შემთხვევაში საჭიროა სამმაგი ბრჭყალის გამოყენება (`'''multiline docstring text'''` ან `"""multiline docstring text """`), ხოლო თუ ერთ ხაზზეა დასაწერი, როგორც სამმაგი ასევე ერთი ბრჭყალის გამოყენებაც შესაძლებელია (მაგ. `'one-line docstring text'` ან `"one-line docstring text"`).

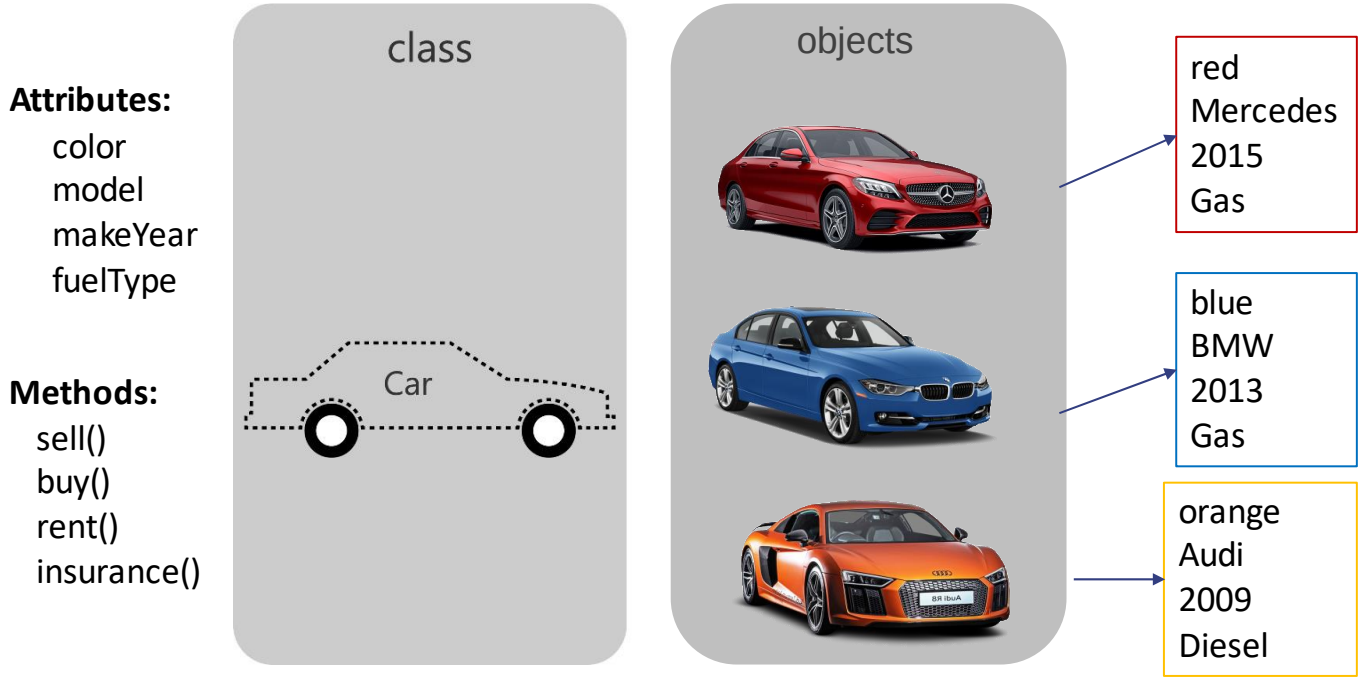
სავარჯიშოები (მარტივი):

1. შექმენით Rectangle კლასი სურათის მიხედვით. დაამატეთ კლასის კონსტრუქტორი შესაბამისი ატრიბუტებით. დაამატეთ მართკუხედის ფართობის და პერიმეტრის გამოსათვლელი ფუნქციები. კლასის გარეთ შემოიღეთ ობიექტები. გამოთვალეთ obj1 ობიექტის ფართობი და დაბეჭდეთ მიღებული შედეგი.



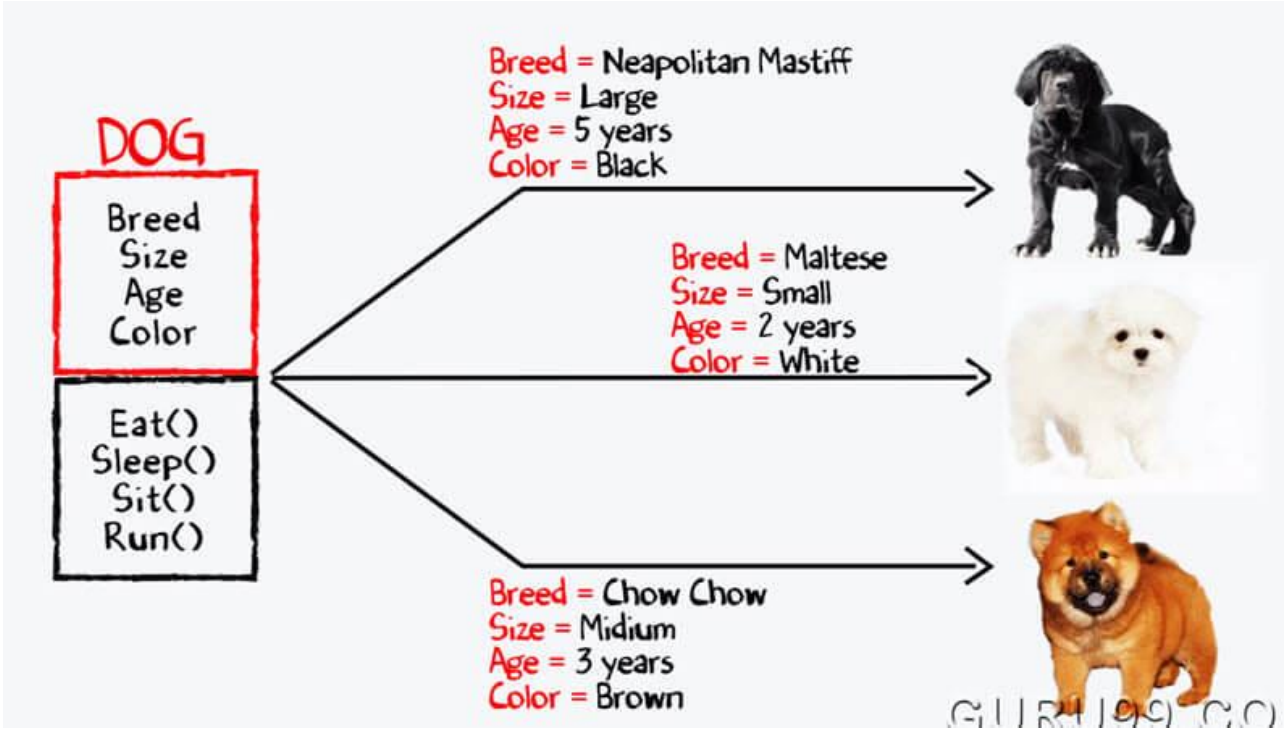
სავარჯიშოები (მარტივი):

2. აღწერეთ კლასი Car ქვემოთ მითითებული ატრიბუტების მიხედვით. შემოიღეთ 3 ობიექტი. კლასში დაამატეთ მეთოდები სურვილისამებრ, რომელიც დაბეჭდავს შესაბამის ტექსტს.



სავარჯიშოები (მარტივი):

3. აღწერეთ კლასი Dog ატრიბუტებით breed, size, age, color. შექმენით 3 ახალი ობიექტი შემდეგი მონაცემების მიხედვით.



სავარჯიშოები (მარტივი):

4. შექმენით კლასი Celsius, რომლის private ატრიბუტია __temperature, მოახდინეთ მისი ინიციალიზაცია კონსტრუქტორში.

   - დაამატეთ get_temp, set_temp და del_temp მეთოდები ატრიბუტზე წვდომისთვის, შეცვლისთვის ან წაშლისთვის.
   - *შექმენით temperature თვისება (property ფუნქციის გამოყენებით - იხილეთ სლაიდი და შეასრულეთ დამოუკიდებლად).*
   - კლასის გარეთ შემოიღეთ Celsius კლასის 2 ობიექტი და გამოიძახეთ შექმნილი მეთოდები.

სავარჯიშო - საშუალო სირთულის

შექმენით კლასი Bank_Account, რომელიც შეიცავს დაფარულ (private) ატრიბუტებს account_name და balance. account_name არის მომხარებლის სახელი, ხოლო balance-ის მნიშვნელობაა თანხა, რომელიც მომხარებლს აქვს საბანკო ანგარიშზე. აღწერეთ კლასის ატრიბუტები კონსტრუქტორის მეშვეობით (ინიციალიზაცია). კონსტრუქტორში balance ატრიბუტს გაჩუმებით (default value) მიანიჭეთ 0.

დამატებით შექმენით შემდეგი მეთოდები კლასში:

1. Account_name-სთვის დაწერეთ getter და setter მეთოდები.

2. დაამატეთ deposit() მეთოდი, რომელიც ასრულებს ანგარიშზე თანხის შეტანას. მას გადაეცემა ერთი პარამეტრი. პარამეტრად გადაცემული მნიშვნელობა ემატება საბანკო ანგარიშზე არსებულ თანხას. ფუნქცია ასევე ბეჭდავს ანგარიშზე არსებულ თანხას. მაგ: “თანხის შეტანა შესრულებულია. ამჟამად თქვენ ანგარიშზე გაქვთ 120 ლარი”

3. დაამატეთ withdraw() მეთოდი, რომელიც ასრულებს ანგარიშიდან თანხის გამოტანას. გადაეცემა ერთი პარამეტრი. პარამეტრად გადაცემული მნიშვნელობა აკლდება საბანკო ანგარიშზე არსებულ თანხას. ფუნქცია ასევე ბეჭდავს ანგარიშზე არსებულ თანხას. მაგ: “თანხის გამოტანა შესრულებულია. ამჟამად თქვენ ანგარიშზე გაქვთ 120 ლარი”

კლასის გარეთ შემოიტანეთ Bank_Account-ის ობიექტი. იხილეთ სურათი, თუ როგორ ხდება ოპერაციების შესრულება და დაწერეთ შესაბამისი კოდი.

```
შეიტანეთ კლიენტის სახელი გვარი: Giorgi Abashidze
რა თანხა გაქვთ ამჟამად ანგარიშზე? 100
შეიტანეთ შესაბამისი კოდი რომელი ოპერაციის შესრულებაც გსურთ:
თანხის შეტანა: 1, თანხის გამოტანა: 2
1
რა თანხის შეტანა გსურთ? 200
თანხის შეტანა შესრულებულია. ამჟამად თქვენ ანგარიშზე გაქვთ 300 ლარი

Process finished with exit code 0
```

სავარჯიშოები (საშუალო სირთულის):

5. აღწერეთ კლასი Fraction (წილადი) ატრიბუტებით top და bottom - სადაც top წარმოადგენს წილადის მრიცხველს, ხოლო bottom წილადის მნიშვნელს. კლასში დაამატეთ __str__() მეთოდი, რომელიც დააბრუნებს Fraction ტიპის ნებისმიერი ობიექტს წილადის ფორმატით. მაგ. “3/5”. კლასში დაამატეთ ფუნქცია, რომელიც დაითვლის ორი წილადის ჯამს და დააბრუნებს შედეგს. ასევე დაამატეთ მეთოდი inverse(), რომელიც დააბრუნებს წილადის შებრუნებულ მნიშვნელობას. მაგ 3/5-ის შებრუნებულია 5/3. კლასის გარეთ შემოიტანეთ ორი კონკრეტული ობიექტი (წილადი) და იპოვეთ მათი ჯამი. დაბეჭდეთ მიღებული შედეგი წილადის ფორმატით. ასევე დაბეჭდეთ, ერთ-ერთი წილადის შებრუნებული მნიშვნელობა.

6. აღწერეთ კლასი Point რომლის ატრიბუტებია x და y. შემოიტანეთ Point ტიპის ობიექტები point1 და point2 რომლის კოორდინატებია (3,5) და (6,9). კლასში დაამატეთ მეთოდი რომელიც დაითვლის სიგრძეს კოორდინატა ცენტრიდან ნებისმიერ წერტილამდე.

სავარჯიშოები (რთული):

6. აღწერეთ კლასი Point. მოახდინეთ მისი ატრიბუტების x და y-ის ინიციალიზაცია კონსტრუქტორის მეშვეობით. დაამატეთ __str__() მეთოდი, რომელიც ჩაწერს Point კლასის ობიექტს შემდეგი ფორმატით “(x,y)”. მაგ. (5,4). კლასში დაამატეთ მეთოდი, რომელიც დაითვლის ორ ნებისმიერ წერტილს შორის მანძილს შემდეგი ფორულით:
$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, სადაც a წერტილის კოორდინატებია (x_1, y_1), ხოლო b წერტილის კოორდინატებია (x_2, y_2).

კლასის გარეთ შემოიტანეთ ნებისმიერი ორი წერტილი a და b თქვენთვის სასურველი კოორდიანტებით input-ის მეშვეობით. დაბეჭდეთ თითოეული a და b წერტილი აღწერილი ფორმატით. დაითვალეთ მათ შორის მანძილი.