BTU ბიზნესისა და ტექნოლოგიების უნივერსიტეტი

python

პროგრამირება

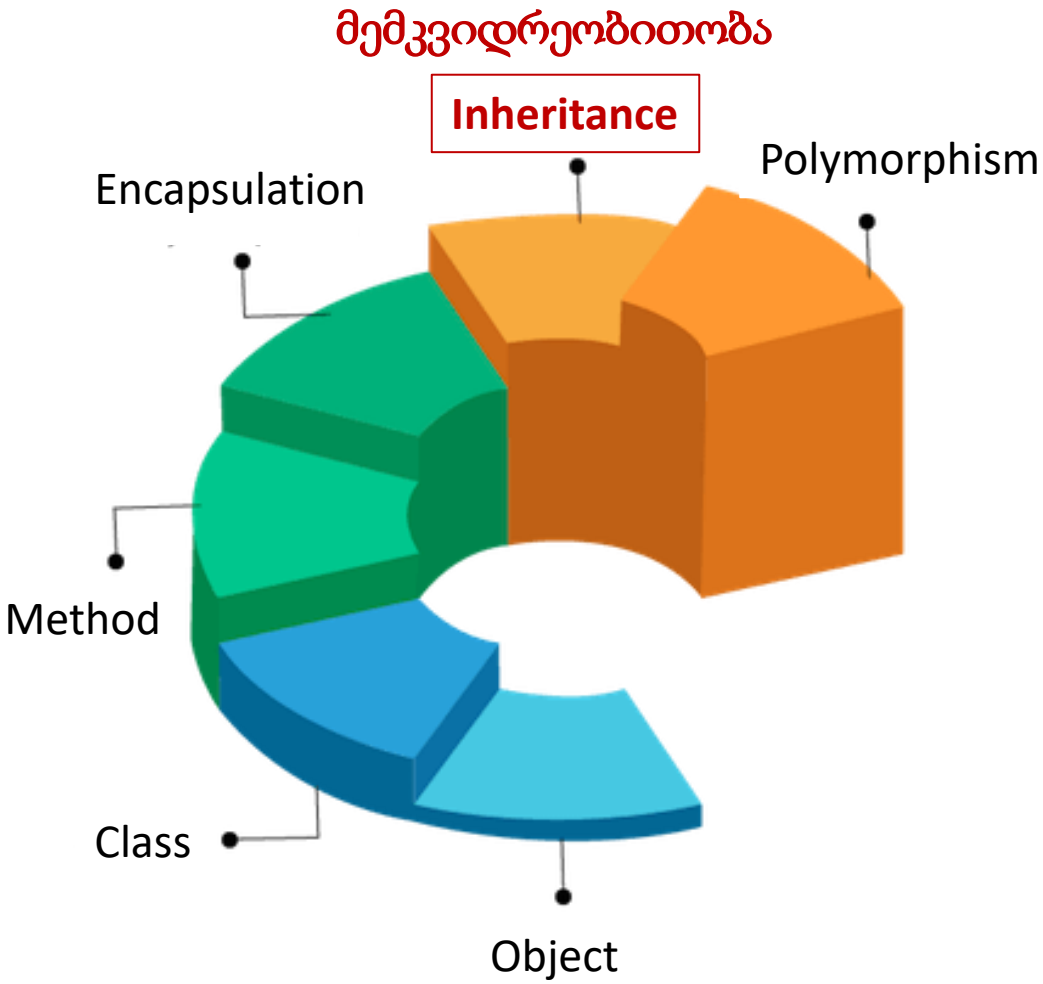
Python

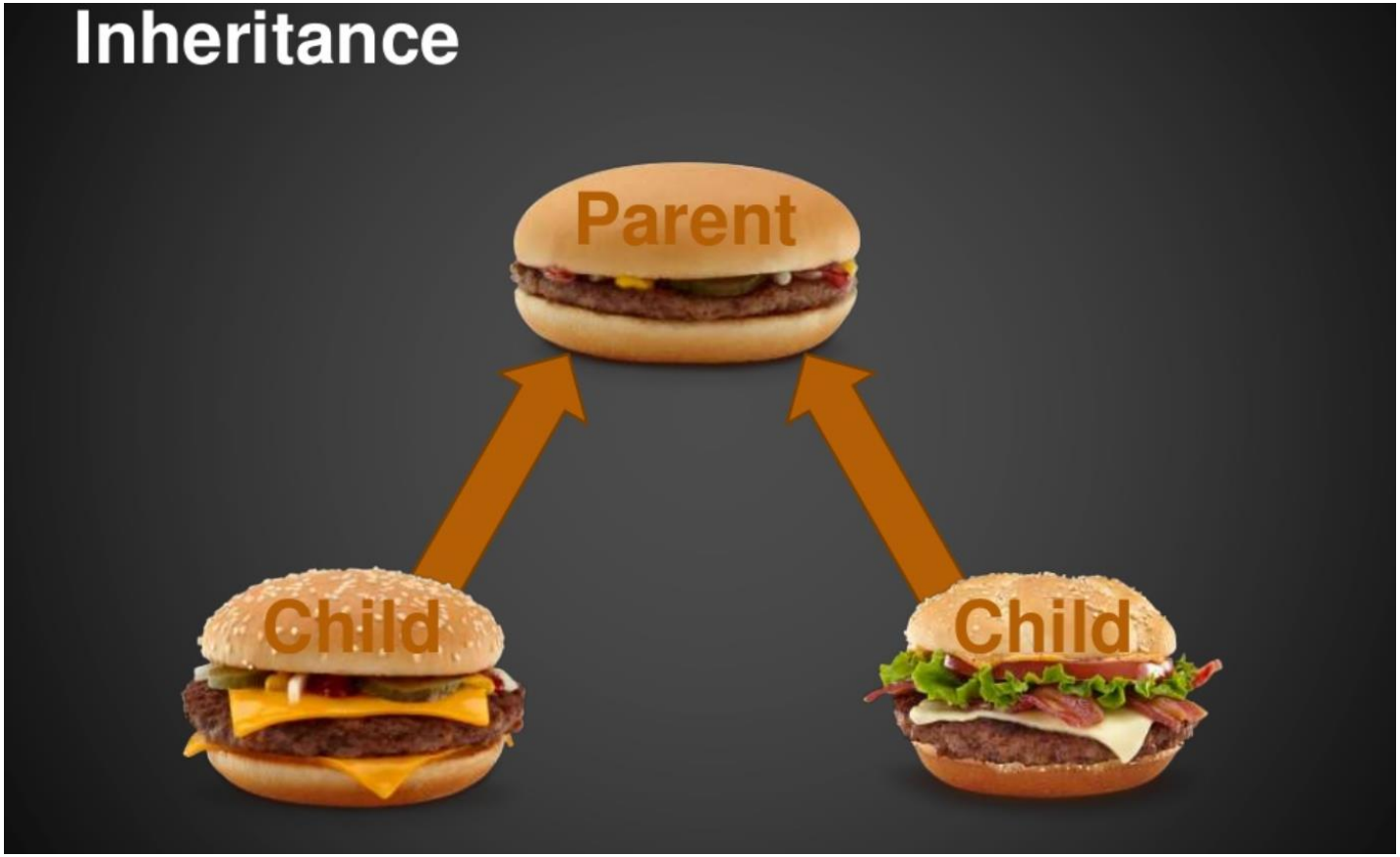
ლექცია 2: კლასების მემკვიდრეობითობა; მეთოდების გადაფარვა

ლიკა სვანაძე

lika.svanadze@btu.edu.ge

Python OOP system




Inheritance
Parent
Child
Child

მემკვიდრეობითობა

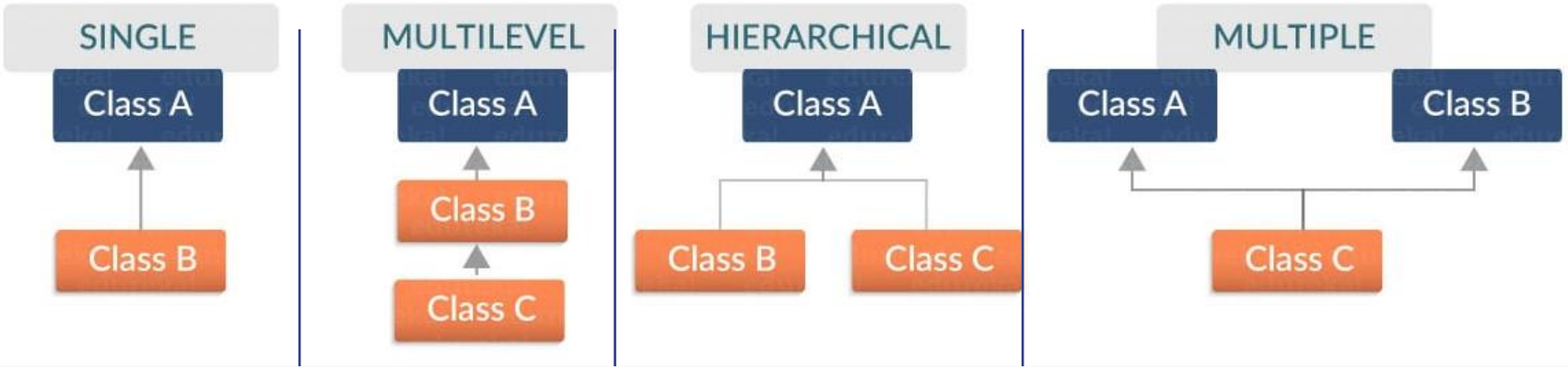
ზოგჯერ საჭირო ხდება შევქმნათ კლასის ქვე-კლასი. მემკვიდრეობითობა ობიექტზე ორინეტირებულ პროგრამირებაში გამოიყენება, როდესაც ერთი კლასის ნიშან-თვისებები გვსურს გადავცეთ სხვა კლასს. ამ შემთხვევაში **საწყის კლასს**, რომლის ბაზაზეც ვქმნით ახალ კლასს, ვუწოდებთ **მშობელ კლასს** (იგივე რაც super class, parent class, base class), **ხოლო ქვე-კლასს** ეწოდება **შვილობილი** კლასი (იგივეა რაც sub class, child class, derive class).

შვილობილი კლასს გააჩნია ყველა ის თვისება, რაც დამახასიათებელია მშობელი კლასისთვის. შესაბამისად, მშობელი კლასის ფუნქციები, ავტომატურად გადადის შვილობილ კლასზეც.

Python-ში ნებისმიერი კლასი წარმოადგენს **object** კლასის ქვე-კლასს. მაგ. int, float, bool ჩაშენებული კლასები არიან object კლასის ქვე-კლასები.

Python2-ში, კლასის აღწერისას ფრჩხილებში უნდა მიეთითოს object კლასი, თუ იგი არ წარმოადგენს რომელიმე სხვა კლასის შვილობილ კლასს. Python3-ში ამის მითითება საჭირო აღარ არის (ავტომატურად იგულისხმება, რომ ნებისმიერი კლასი არის object კლასის შვილობილი კლასი).

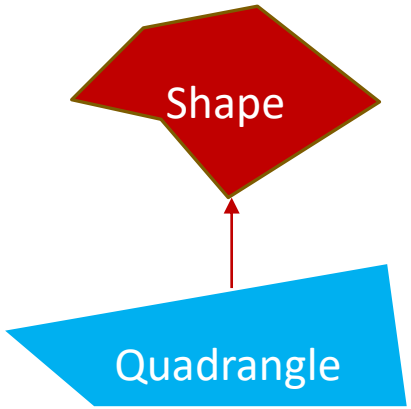
მემკვიდრეობითობის ტიპები



მარტივი მემკვიდრეობითობა - Single Inheritance (1)

მაგალითი

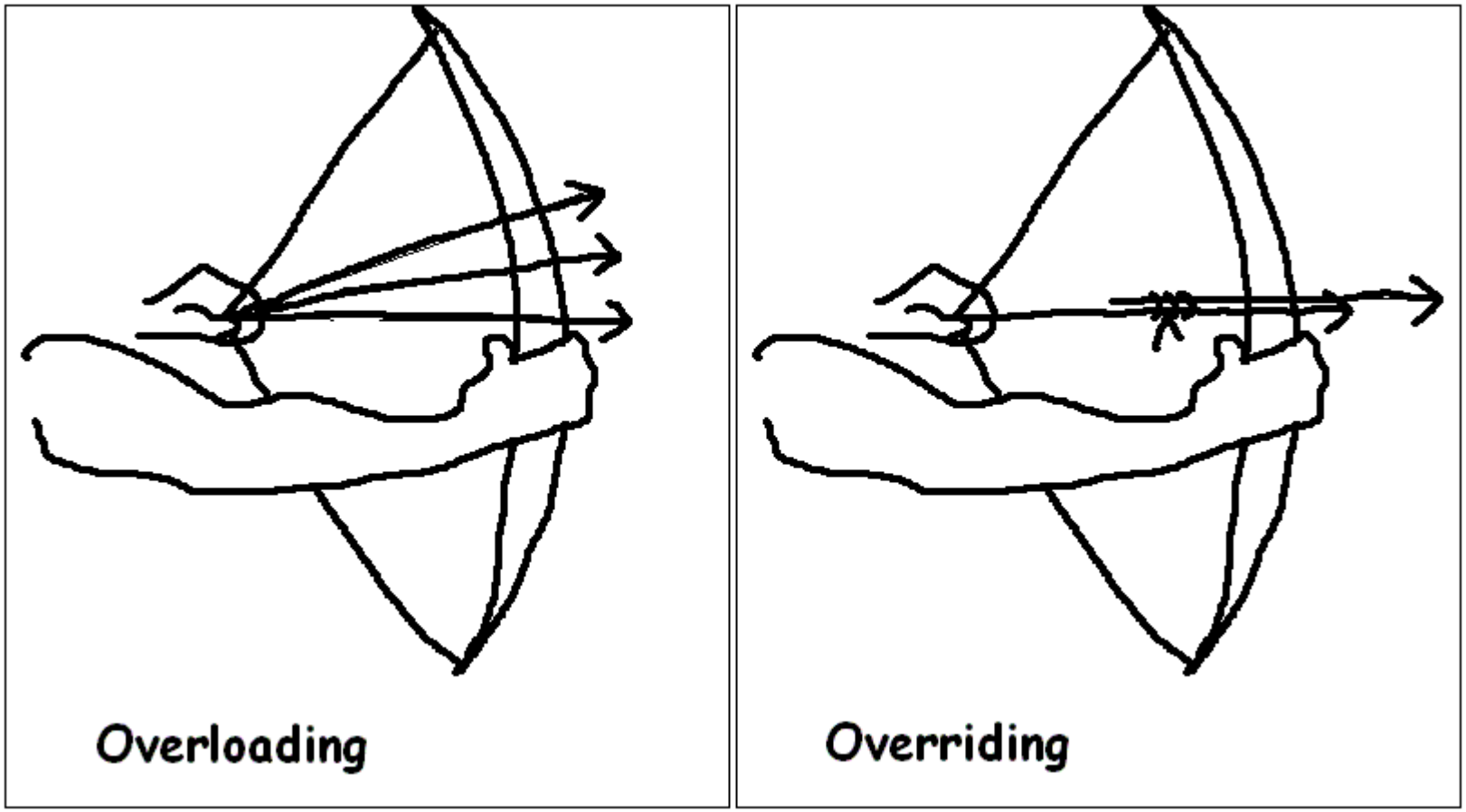
```
class Shape:
    def __init__(self, color):
        self.color = color

    def say_hi(self):
        print(f'I am a shape with {self.color} color')

class Quadrangle(Shape):
    pass

s1 = Shape('red')
s1.say_hi()
q1 = Quadrangle('blue')
q1.say_hi()
```



შედეგი:

```
I am a shape with red color
I am a shape with blue color
```

Overloading, Overriding



Single Inheritance (2) - Method Overriding

* Quadrangle კლასი წარმოადგენს Shape კლასის შვილობილ კლასს, რაც ნიშნავს რომ მეთოდები, რომლებიც აღწერია Shape კლასში შეიძლება გამოყენებული იქნას Quadrangle კლასის ობიექტებისთვისაც.
* თუ ორივე კლასში არის ერთი და იგივე სახელის მეთოდი (ერთი და იგივე პარამეტრებით), შვილობილი კლასის ობიექტისთვის პრიორიტეტს წარმოადგენს შვილობილ კლასში აღწერილი მეთოდი - **Method Overriding** (გადაფარვა) - *დეტალურად განხილულია შემდეგ სლაიდზე.*

მაგალითი

```
class Shape:
    def __init__(self, color):
        self.color = color

    def say_hi(self):
        print(f'I am a shape with {self.color} color')

class Quadrangle(Shape):
    def say_hi(self):
        print(f'I am a quadrangle with {self.color} color')

s1 = Shape('red')
s1.say_hi()
q1 = Quadrangle('blue')
q1.say_hi()
```

შედეგი:

```
I am a shape with red color
I am a quadrangle with blue color
```

Method Overriding - მეთოდების გადაფარვა

* მემკვიდრეობითობის თვისების გათვალისწინებით, შვილობილ კლასს ავტომატურად გადაეცემა მისი მშობელი კლასის ყველა მეთოდი. თუმცა, ხანდახან საჭირო ხდება, მეთოდის ფუნქციონალის **ცვლილება** ან **განვრცობა** შვილობილი კლასისთვის. ესეთ შემთხვევაში, შვილობილ კლასში ვამატებთ მეთოდს იგივე სახელით. პროცესს, რომლის დროსაც ხდება იგივე სახელით მეთოდის იმპლემენტაცია შვილობილ კლასში უწოდებენ **მეთოდების გადაფარვას**. მეთოდების გადაფარვა გამოიყენება მაშინ, როდესაც ქვე-კლასისთვის გვსურს მშობელი კლასის მეთოდის მოდიფიცირება (გაფართოება).

super() მეთოდი (1)

* იმავე მაგალითში, Quadrangle კლასისთვის დაწერეთ კონსტრუქტორი, რომელიც მოიცავს Shape კლასის კონსტრუქტორს და დამატებითი სხვა ცვლადების ინიციალიზაციასაც.

მაგალითი

```
class Shape:
    def __init__(self, color):
        self.color = color

    def say_hi(self):
        print(f'I am a shape with {self.color} color')

class Quadrangle(Shape):
    def __init__(self, a, b, c, d, color):

        super().__init__(color)
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    def say_hi(self):
        print(f'I am a quadrangle with {self.color} color')

q1 = Quadrangle(4, 2, 1, 6, 'blue')
q1.say_hi()
```

მშობელი კლასის ინიციალიზაციის მეთოდის გამოძახება იგივეა რაც: `Shape.__init__(self,color)`

შედეგი:

```
I am a quadrangle with blue color
```

super() მეთოდი (2)

* super() მეთოდის მეშვეობით შესაძლებელია მშობელი კლასის მეთოდებზე წვდომა. super() მეთოდი აბრუნებს მშობელი კლასის ზოგად ობიექტს.

```
super().__init__(color)
```

იგივეა, რაც

```
Shape.__init__(self,color)
```

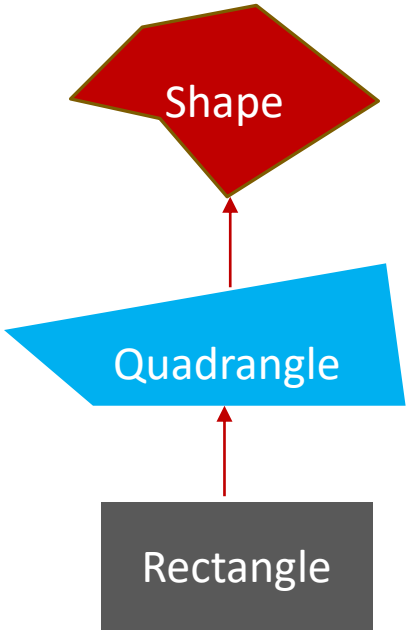
მოქმედება სრულდება ანალოგიურად როგორც Shape კლასის ობიექტზე (პარამეტრებში არ ჭირდება self-ის მითითება).

მოქმედება სრულდება მშობელი კლასის სახელწოდების მითითებით. შესაბამისად self უნდა გადაეცეს პირველ პარამეტრად

Multilevel (მრავალდონიანი) Inheritance

მაგალითი

```
class Shape:
    def __init__(self, color):
        self.color = color

    def say_hi(self):
        print(f'I am a shape with {self.color} color')

class Quadrangle(Shape):
    def __init__(self, a, b, c, d, color):
        super().__init__(color)
        self.a, self.b, self.c, self.d = a, b, c, d

    def say_hi(self):
        print(f'I am a quadrangle with {self.color} color')

class Rectangle(Quadrangle):
    def __init__(self, a, b, color):
        Shape.__init__(self, color)
        self.a = a
        self.b = b


s1 = Shape('red')
s1.say_hi()
q1 = Quadrangle(4, 2, 1, 6, 'blue')
q1.say_hi()
r1 = Rectangle(1, 5, 'black')
r1.say_hi()
```



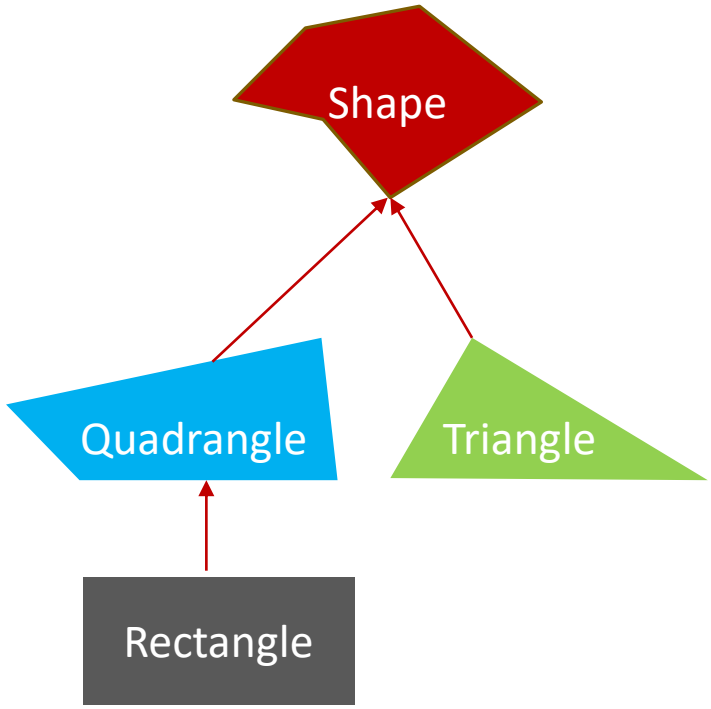
შედეგი:

```
I am a shape with red color
I am a quadrangle with blue color
I am a quadrangle with black color
```

იერარქიული მემკვიდრეობითობა

მაგალითი

```
class Shape:
    def __init__(self, color):
        self.color = color

    def say_hi(self):
        print(f'I am a shape with {self.color} color')

class Quadrangle(Shape):
    def __init__(self, a, b, c, d, color):
        super().__init__(color)
        self.a, self.b, self.c, self.d  = a, b, c, d

    def say_hi(self):
        print(f'I am a quadrangle with {self.color} color')

class Triangle(Shape):
    pass

class Rectangle(Quadrangle):
    def __init__(self, a, b, color):
        Shape.__init__(self, color)
        self.a = a
        self.b = b


s1 = Shape('red')
s1.say_hi()
q1 = Quadrangle(4, 2, 1, 6, 'blue')
q1.say_hi()
r1 = Rectangle(1, 5, 'black')
r1.say_hi()
t1 = Triangle('green')
t1.say_hi()
```



შედეგი:

```
I am a shape with blue color
I am a quadrangle with red color
I am a quadrangle with red color
I am a shape with green color
```

isinstance(), issubclass()

არსებობს 2 ფუნქცია, რომელიც გამოიყენება დაკავშირებულ კლასებთან სამუშაოდ. ესენია:

* isinstance(obj, cl) მეთოდი - რომელსაც გადაეცემა ორი პარამეტრი: obj ობიექტი და cl კლასი. მეთოდი ამოწმებს, ეკუთვნის თუ არა obj ობიექტი cl კლასს ან არის თუ არა cl კლასი მისი მშობელი (წინაპარი) კლასი. შედეგი არის True ან False.

```
print(isinstance(q1, Shape))      #Output: True
print(isinstance(q1, Quadrangle))  #Output: True
print(isinstance(q1, object))    #Output: True
print(isinstance(5, int))        #Output: True
print(isinstance(5, object))       #Output: True
```

* issubclass(cl1, cl2) მეთოდი - რომელსაც გადაეცემა ორი პარამეტრი cl1 და cl2 კლასი. მეთოდი ამოწმებს, არის თუ არა cl1 კლასი cl2-ის მემკვიდრე კლასი. შედეგი არის True ან False.

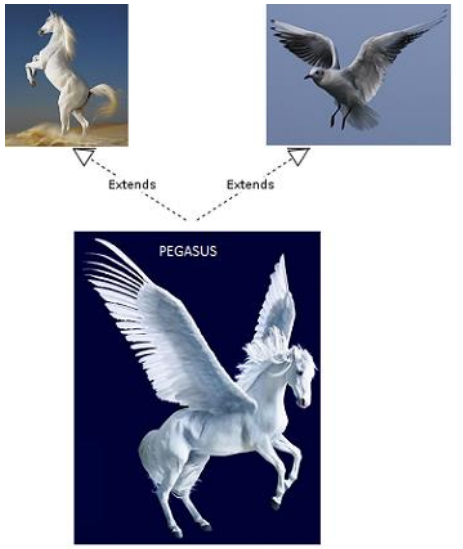
```
print(issubclass(Shape, object))    #Output: True
print(issubclass(Quadrangle, Shape))  #Output: True
print(issubclass(Shape, Quadrangle))   #Output: False
print(issubclass(int, object))      #Output: True
print(issubclass(bool, int))       #Output: True
```

მრავლობითი (Multiple) მემკვიდრეობითობა

Python-ში არსებობს მრავალჯერადი მემკვიდრეობითობა, რომლის მეშვეობითაც კლასი შეიძლება იყოს რამდენიმე (ორი ან მეტი) კლასის მემკვიდრე.

მაგალითი 1:

მაგალითი 2:



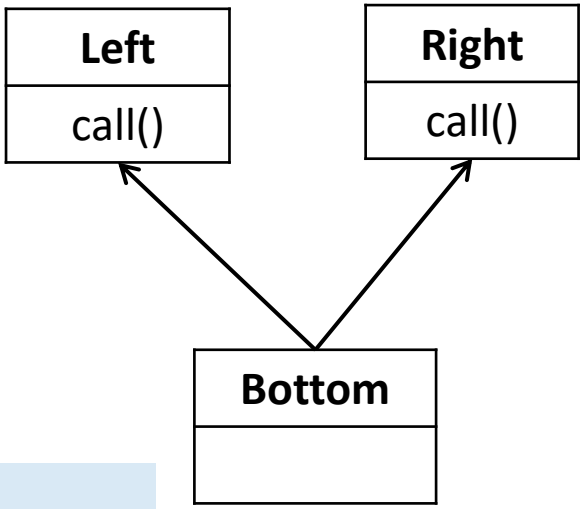
მრავლობითი მემკვიდრეობითობა ხშირად არ გამოიყენება პითონში (იხ. Diamond პრობლემა მომდევნო სლაიდებზე), თუმცა იგი უმეტესად გამოიყენება **mixin**-სთვის. mixin ეწოდება კლასს, რომელიც შინაარსობრივად არ არის შექმნილი როგორც ცალკეული კლასი, წარმოადგენს სხვა კლასის მშობელ კლასს, რომელიც შესძენს მას დამატებით ფუნქციონალს.

მრავლობითი (Multiple) მემკვიდრეობითობა

მრავალჯერადი მემკვიდრეობითობის სინტაქსია:

```
class SubclassName(BaseClass1, BaseClass2, BaseClass3, ...):
        pass
```

შვილობილი კლასის აღწერისას ()-ში მიეთითება საბაზისო (მშობელი) კლასის სახელწოდებები პრიორიტეტების თანმიმდევრობით.



მაგალითი

```
class Left:
    def call(self):
        print("Left")

class Right:
    def call(self):
        print("Right")

class Bottom(Left, Right):
    pass

obj = Bottom()
obj.call()
```

შედეგი:

```
Left
```

Method Resolution Order (MRO)

mro() ფუნქციის გამოყენებით შესაძლებელია მშობელი კლასების ჩამოთვლა პრიორიტეტების კლების მიხედვით. გაითვალისწინეთ ,mro() ფუნქცია გამოიყენება კლასის სახელწოდებასთან. შედაგად მიიღება List-ის ტიპის მონაცემი.

მაგალითი

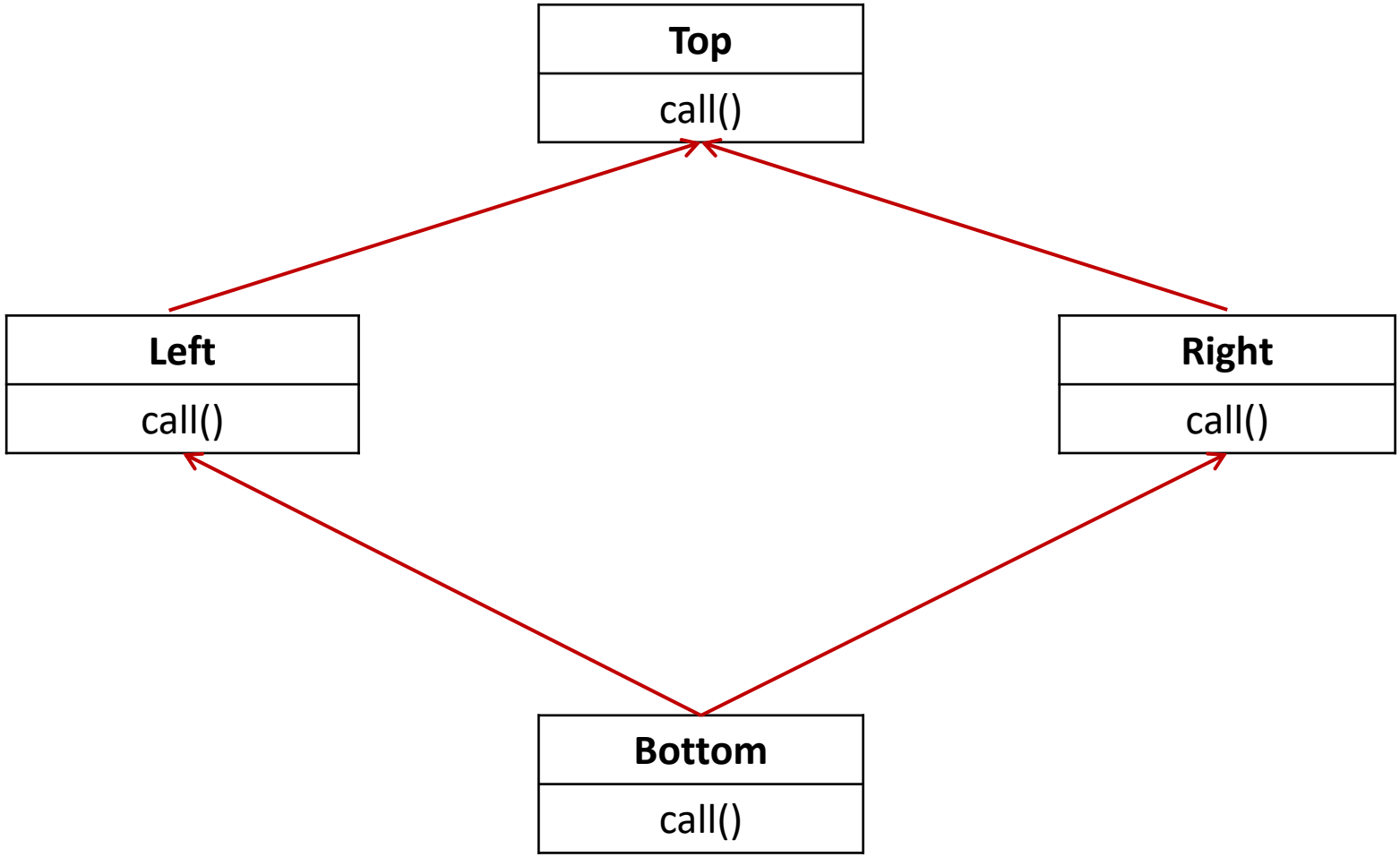
```
class Left:
    def call(self):
        print("Left")


class Right:
    def call(self):
        print("Right")

class Bottom(Left, Right):
    pass


obj = Bottom()

print(Bottom.mro())
```

შედეგი:

```
[
<class '__main__.Bottom'>,
<class '__main__.Left'>,
<class '__main__.Right'>,
<class 'object'>
]
```

მშობელი კლასების ჩამონათვალი პრიორიტეტების მიხედვით

Diamond problem - მეთოდების გადაფარვა მრავლობითი მემკვიდროებითობის დროს



hasattr()

hasattr() მეთოდის სინტაქსია: `hasattr(object, name)`

- hasattr() მეთოდი აბრუნებს True-ს თუ object ობიექტს აქვს name ატირბუტი. წინააღმდეგ შემთხვევაში აბრუნებს False.
- hasattr() მეთოდს გადაეცემა შემდეგი პარამეტრები:
  - **object** - ობიექტი, რომლის კონკრეტული ატრიბუტის არსებობა უნდა შემოწმდეს
  - **name** - ატრიბუტის სახელი სტრიქონის სახით

მაგალითი

```
class Student:
    def __init__(self, first, last):
        self.first = first
        self.last = last


st1 = Student("George", "Abashidze")
print("Person has first name? ", hasattr(st1, "first"))
print("Person has age? ", hasattr(st1, "age"))
```

შედეგი

```
Person has first name?  True
Person has age?  False
```

getattr()

getattr() მეთოდის სინტაქსია: `getattr(object, name, default)`

- getattr() მეთოდი აბრუნებს object ობიექტის name ატრიბუტის მნიშვნელობას. თუ name ატრიბუტი არ მოიძებნა, დააბრუნებს default-ად მითითებულ მნიშვნელობას.
- getattr() მეთოდს გადაეცემა შემდეგი პარამეტრები:
  - **object** - ობიექტი, რომლის კონკრეტული ატრიბუტის მნიშვნელობა უნდა დაბრუნდეს
  - **name** - ატრიბუტის სახელი სტრიქონის სახით
  - **default** - მნიშვნელობა, რომელსაც დააბრუნებს getattr() ფუნქცია იმ შემთხვევაში, თუ ობიექტისთვის name ატრიბუტი არ მოიძებნა. შესაძლოა მესამე პარამეტრი არ იყოს მითითებული getattr() ფუნქციაში.

მაგალითი

```
class Student:
    def __init__(self, first, last):
        self.first = first
        self.last = last


st1 = Student("George", "Abashidze")
print(getattr(st1,"first"))   #Output: George
print(getattr(st1,'gender', 'Male')) #Output: Male
print(getattr(st1,'gender'))  #Output: AttributeError
```

setattr()

setattr() მეთოდის სინტაქსია: `setattr(object, name, value)`

- setattr() მეთოდი **object** ობიექტის **name** ატირბუტს ანიჭებს **value** მნიშვნელობას. ფუნქცია არ აბრუნებს რაიმე მნიშვნელობას (აბრუნებს None მნიშვნელობას).
- setattr() მეთოდს გადაეცემა სამი პარამეტრი:
  - **object** - ობიექტი, რომლის კონკრეტულ ატრიბუტს მნიშვნელობა უნდა მიენიჭოს
  - **name** - ატრიბუტის სახელი სტრიქონის სახით
  - **value** - მნიშვნელობა, რომელიც უნდა მიენიჭოს **object** ობიექტის **name** ატრიბუტს

მაგალითი

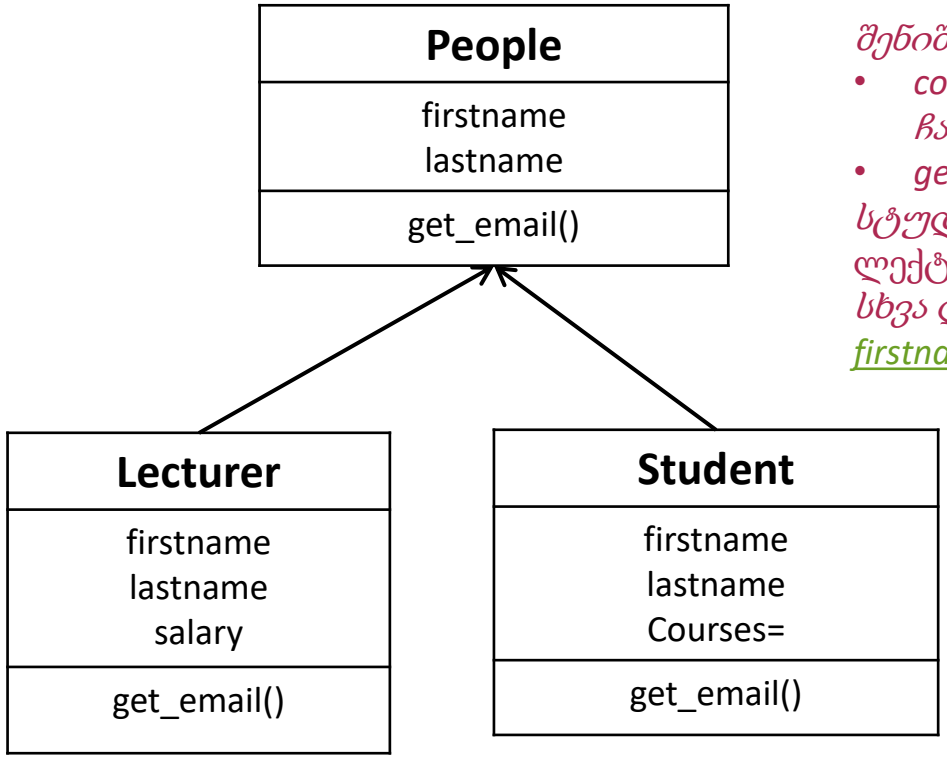
```
class Student:
    def __init__(self, first, last):
        self.first = first
        self.last = last


st1 = Student("George", "Abashidze")
print("Before:", st1.first)
setattr(st1, 'first', 'David')
print("After:", st1.first)
```

შედეგი

```
Before: George
After: David
```

სავარჯიშოები:

1. ***სავარჯიშო University:*** აღწერეთ კლასი People, Student, Lecturer-შემდეგი სქემის მიხედვით (გამოიყენეთ მემკვიდრეობითობა). შვილობილი კლასის ინიციალიზაციაში გამოიძახეთ მშობელი კლასის ინიციალიზაცია. სურვილისამებრ შეგიძლიათ დაამატოთ სხვა ფუნქციონალი. კლასების აღწერის შემდეგ შემოიღეთ სხვადასხვა ობიექტები და გააკეთეთ მათზე მანიპულაციები.



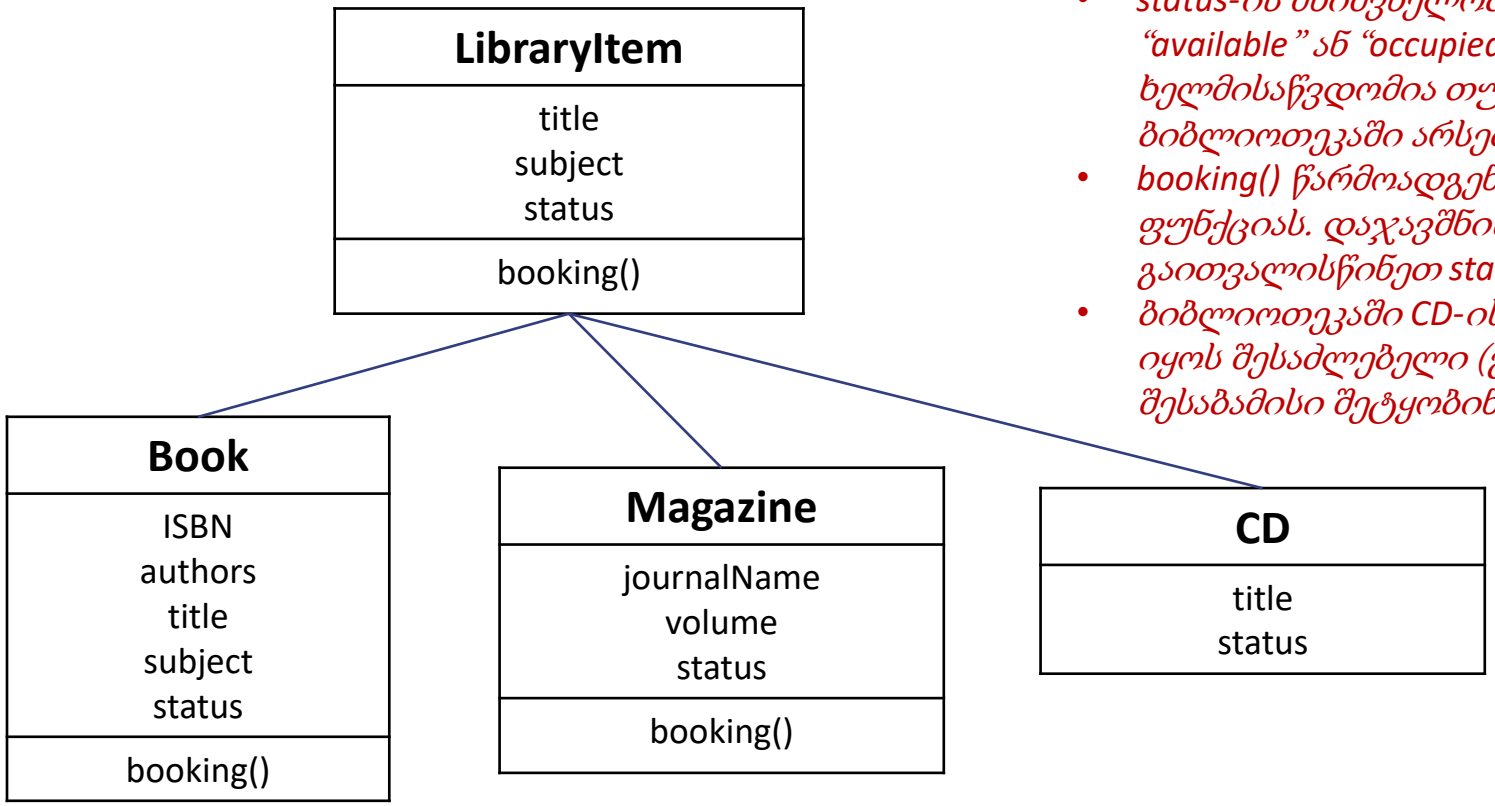
შენიშვნა:

- *courses ატრიბუტი წარმოადგენს საგნების ჩამონათვალი list-ის სახით;*
- *get_email() აგენერირებს იმეილს შემდეგნაირად:*

სტუდენტებისთვის - firstname.lastname.1@btu.edu.ge
ლექტორებისთვის - firstname.lastname@btu.edu.ge
სხვა დანარჩენისთვის კი - firstname.lastname.uni@btu.edu.ge

სავარჯიშოები:

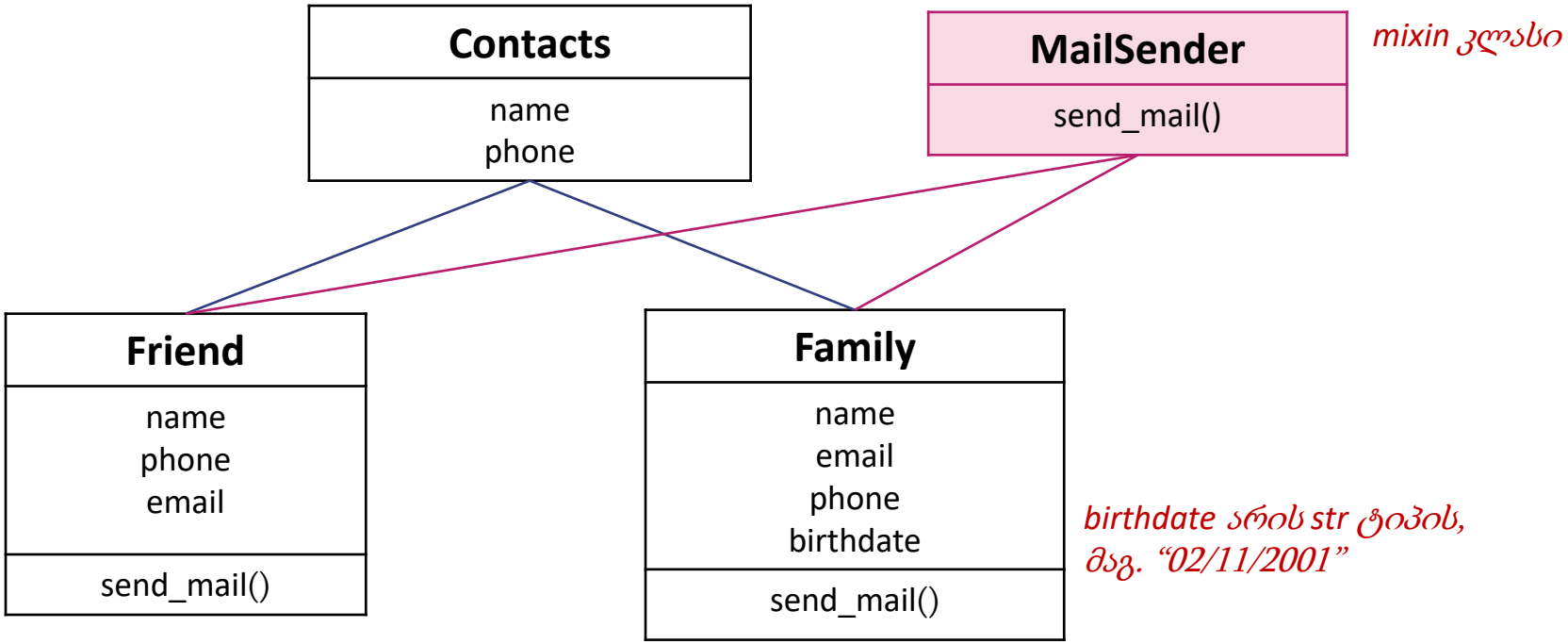
2. ***სავარჯიშო Library:*** შემდეგი სტრუქტურის მიხედვით შემქენით კლასები, განსაზღვრეთ მემკვიდრეობითობა, კლასის ატრიბუტები და მეთოდები (შეგიძლიათ სავარჯიშოს ფუნქციონალი თავად განავრცოთ):



- *status-ის მნიშვნელობა შეიძლება იყოს "available" ან "occupied", რაც მიუთითებს ხელმისაწვდომია თუ დაჯავშნილი ბიბლიოთეკაში არსებული მასალა.*
- *booking() წარმოადგენს დაჯავშნის ფუნქციას. დაჯავშნის დროს გაითვალისწინეთ status-ის მნიშვნელობა.*
- *ბიბლიოთეკაში CD-ის დაჯავშნა არ უნდა იყოს შესაძლებელი (გამოიტანოს შესაბამისი შეტყობინება).*

სავარჯიშოები:

3. ***სავარჯიშო Contact Manager:*** შექმენით კლასი Contacts, რომელიც აღწერს ადამიანების საკონტაქტო მონაცემებს. დაამატეთ 2 ქვე კლასი - Friends და Family შესაბამისი ატრიბუტებით. კლასებს რომელსაც აქვს email ატრიბუტად, მათთვის შესაძლებელი უნდა იყოს მეილის გაგზავნის ფუნქციის გამოყენება. ამისათვის დაამატეთ mixin კლასი MailSender, სადაც აღწერთ მეთოდს send_mail(). ეს მეთოდი ბეჭდავს შესაბამის შეტყობინებას*: "თქვენი მეილი გაიგზავნა ამ მისამართზე: giorgi.giorgadze@gmail.com"*. გაიაზრეთ mixin კლასის იდეა.



სავარჯიშოები:

4. ***სავარჯიშო University with Diamond Problem:*** სავარჯიშო 1-ს დაამატეთ Doctoral_Student კლასი შესაბამისი ატრიბუტებით და get_email() მეთოდით - მოიფიქრეთ როგორი სახის მეილი უნდა დააგენერიროს და რამდენი. ყველა კლასის ობიექტისთვის გჭირდებათ get_email მეთოდი (იხ. სავარჯიშო 1-ის პირობა). გაიაზრეთ diamond problem-ის არსი ამ მაგალითზე. როდის წარმოიშვება პრობლემა? ჩამოაყალიბეთ თქვენთვის ყველაზე ოპტიმალური გამოსავალი თქვენს ალგორითმში.