Яндекс Лицей

Урок Polymorphism

Օբյեկտ-կողմնորոշված ծրագրավորման (ՕԿԾ) ներածություն: Պոլիմորֆիզմ

1 Պոլիմորֆիզմ

2 Օբյեկտի տիպի ստուգում

Անոտացիա

Այդ դասին մենք կուսումնասիրենք տարբեր բնույթի օբյեկտների հետ փոխազդեցության նմանատիպ միջոցներ տրամադրելու հնարավորությունները:

1. Պոլիմորֆիզմ

Նախորդ դասին մենք արդեն պատկերացում ստացանք օբյեկտ-կողմնորոշված ծրագրավորման մասին՝ յուրացրեցինք դասերի և մեթոդների սահմանումը, օբյեկտներում ատրիբուտների ավելացումը: Պարզ է, որ դասերը, օբյեկտները, մեթոդները, ատրիբուտները բավականին հարմար են և գեղեցիկ, սակայն ինչո՞ւմ է կայանում դրանց առավելությունը ֆունկցիաների համեմատ: Չէ՞ որ որոշ օբյեկտներ հնարավոր էր փոխանցել ֆունկցիաներին և դրանց հետ կատարել նույն գործողությունները, ինչ մեթոդների օգնությամբ: Այդ դեպքում ո՞րն է լրացուցիչ շարահյուսություն և կանոններ սահմանելու իմաստը: Հիմնական առավելությունը կայանում է նրանում, որ օբյեկտ-կողմնորոշված մոտեցումը թույլ է տալիս գրել կոդ, որն աշխատում է տարբեր դասերի նմուշահատերի հետ: Երբեմն կոդը կարող է աշխատել նաև դեռ չստեղծված դասերի հետ:

Պոլիմորֆիզմ

Կոդի՝ տվյալների տարբեր տեսակների հետ աշխատելու հատկությունը կոչվում է **պոլիմորֆիզմ**.

Մենք արդեն բազմիցս օգտվել ենք տարբեր ֆունկցիաների և օպերատորների այդ հատկությունից՝ նույնիսկ չմտածելով այդ մասին: Օրինակ, + օպերատորը պոլիմորֆային է

```
print(1 + 2)          # 3
print(1.5 + 0.2)      # 1.7
print("abc" + "def")  # abcdef
```

+ օպերատորի ներքին իրագործումը ամբողջ թվերի, լողացող կետով թվերի և տողերի համա… …ս տարբերվում է: Իրականում դրանք երեք տարբեր գործողություններ են՝ Python-ի ինտերպրետատորը

Чаты

կատարման համար ընտրում է դրանցից մեկը՝ օպերանդներից կախված: Թեև մեր դեպքում ընտրությունն ակնհայտ է, քանի որ օպերանդները պարզապես հաստատուններ են:

Բարդացնենք խնդիրը:

```
def f(x, y):
    return x + y


print(f(1, 2))          # 3
print(f(1.5, 0.2))      # 1.7
print(f("abc", "def"))  # abcdef
```

Ինտերպրետատորին չհաջողվեց խաբել, քանի որ Python-ը դինամիկ տիպավորմամբ լեզու է: Նման լեզուներում ցանկացած արժեք իր մեջ պարունակում է տեղեկատվություն տիպի մասին: Հենց այդ տեղեկատվությունն ինտերպրետատորին օգնեց ընտրել + գործողության իրագործման ճիշտ տարբերակը (ինչպես նաև `print` ֆունկցիայի ճիշտ տողային ներկայացումը): Սակայն մենք գիտենք, որ Python-ում տվյալների տիպը հանդիսանում է օբյեկտի դաս և գործողությունն ընտրելիս օգտագործվում է օբյեկտի դասի մասին հենց այդ տեղեկատվությունը: Հաջորդ դասին մենք կվերադառնանք + օպերատորին և կուսումնասիրենք մեր սեփական դասերի համար դրա իրագործման եղանակները:

Այժմ եկե՛ք հիշենք `__init__`. մեթոդի մասին: Այն կատարվում է դասի յուրաքանչյուր նոր օրինակ ստեղծելիս և սկզբնարժեքավորում է նոր օրինակի հատկությունները: Առաջին արգումենտը՝ `self`-ը այն ստանում է ինտերպրետատորից: Մնացած արգումենտները դասին փոխանցվում են կլոր փակագծերում՝ օրինակը ստեղծելիս:

```
class Book:
    def __init__(self, name, author):
        self.name = name
        self.author = author

    def get_name(self):
        return self.name

    def get_author(self):
        return self.author


book = Book('Պատերազմ և խաղաղություն', 'Տոլստոյ Լ. Ն.')
print('{}, {}'.format(book.get_name(), book.get_author()))
# Պատարեզմ և խաղաղություն, Տոլստոյ Լ. Ն.
```

`book = Book('Պատերազմ և խաղաղություն', 'Տոլստոյ Լ.Ն.')` կոդի միջոցով օրինակ ստեղծելիս կկանչվի `__init__`, մեթոդը, որը կստեղծի `name`, `author` ատրիբուտները: Հատկությունները կարելի է կարդալ անմիջապես օբյեկտից (օրինակ `book.name`) կամ օգտագործել դրա համար սահմանված մեթոդները: Երկրորդ տարբերակը նախընտրելի է, քանի որ թույլ է տալիս դասից օգտվող ծրագրավորողներին պաշտպանել դասի իրագործման հնարավոր փոփոխություններից:

Այժմ մենք պատրաստ ենք սահմանելու մեր սեփական դասերը, որոնց օգնությամբ կուսումնասիրենք պոլիմորֆիզմը:

Ուսումնասիրենք «Շրջան» և «Քառակուսի» դասերի իրագործումը՝ մակերեսի և պարագծի հաշվարկման նպատակով

```
from math import pi


class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return pi * self.radius ** 2

    def perimeter(self):
        return 2 * pi * self.radius


class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

    def perimeter(self):
        return 4 * self.side
```

Մենք սահմանել ենք `Circle` և `Square`, դասերը, որոնց նմուշահատերը կարող են հաշվել շրջանագծերի և քառակուսիների մակերեսները և պարագծերը: Կարևոր է, որ երկու դասերն ունեն նույն ինտերֆեյսը՝ մակերեսի հաշվարկի մեթոդները կոչվում են `area`, իսկ պարագծի հաշվարկի մեթոդները `perimeter`. : Այդ մեթոդներն ունեն նաև նույն քանակի պարամետրեր (տվյալ դեպքում՝ միայն `self`)-ը): Երկու մեթոդներն էլ աշխատանքի արդյունքում վերադարձնում են թիվ, որը սակայն կարող է լինել տարբեր տիպի (ամբողջ և իրական):

Այժմ կարող ենք սահմանել `print_shape_info`պոլիմորֆային ֆունկցիա, որը կարտածի տեղեկություններ պատկերի մասին

```
def print_shape_info(shape):
    print("Area = {}, perimeter = {}.".format(
        shape.area(), shape.perimeter()))


square = Square(10)
print_shape_info(square)
# Area = 100, perimeter = 40.

circle = Circle(10)
```

```
print_shape_info(circle)
# Area = 314.1592653589793, perimeter = 62.83185307179586.
```

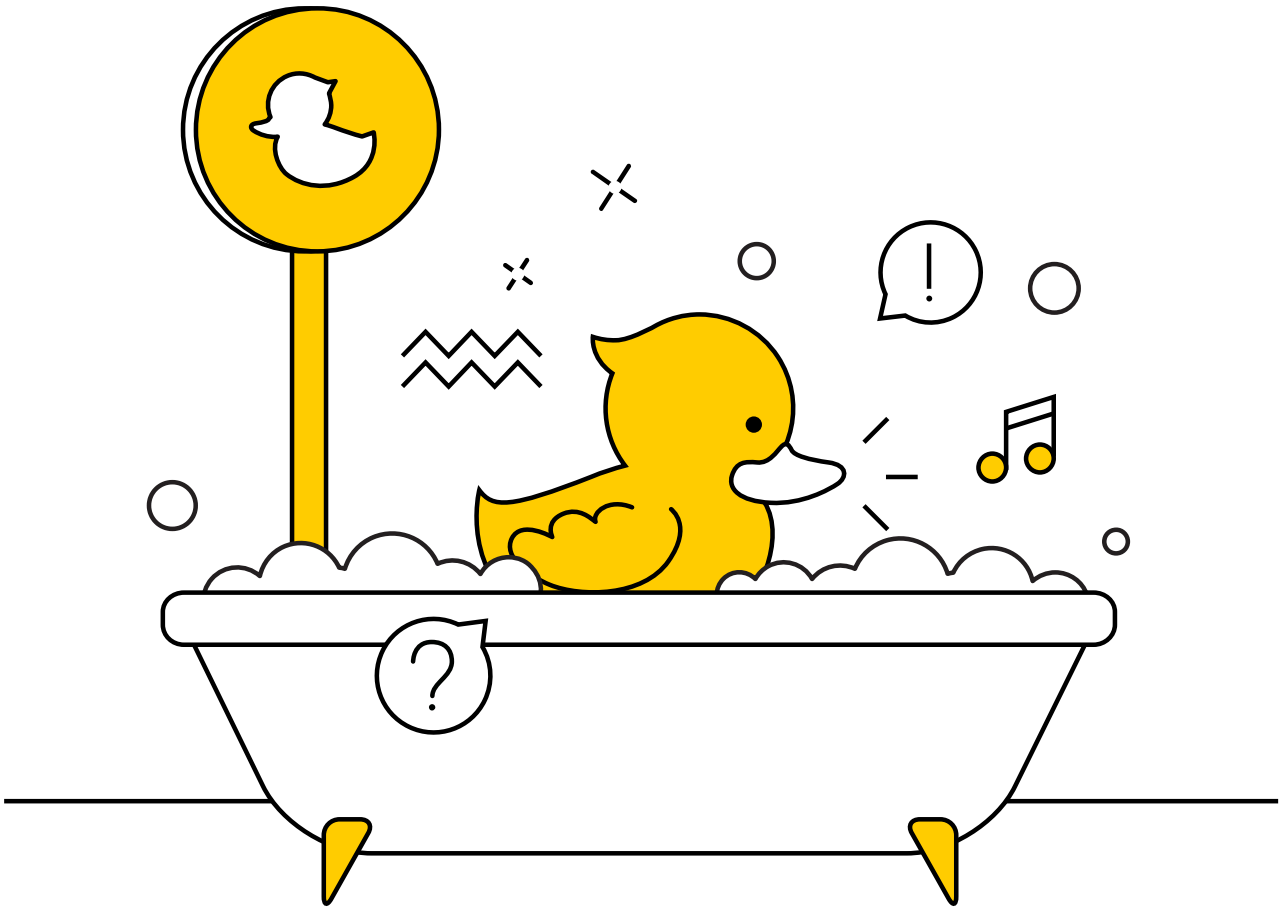
Եթե `print_shape_info` ֆունկցիայի արգումենտը `Square`դասի օրինակն է, ապա կատարվում են այդ դասում սահմանված մեթոդները, իսկ եթե `Circle` դասի օրինակն է, ապա կատարվում են `Circle`-ի մեթոդները:

```
print(dir(square))  # Նմուշահատի հատկությունները և հատուկ մեթոդները:

print(dir(Square))  # Դասի հատկությունները և հատուկ մեթոդները:
```

Բադային տիպավորում

Տվյալ կոդն օգտագործում է այն փաստը, որ Python-ում ընդունված է այսպես կոչված **«բադային տիպավորումը»**: Նման անվանումն առաջացել է հետևյալ հումորային արտահայտությունից՝ «եթե ինչ-որ բան բադի նման է, լողում և կռնչում է բադի նման, ապա ամենայն հավանականությամբ դա հենց բադ է»:



Python-ով գրված ծրագրերի իմաստով դա նշանակում է, որ եթե ինչ-որ օբյեկտ ապահովում է իրենից պահանջվող բոլոր գործողությունները, ապա դրա հետ կաշխատեն այդ գործողությունների միջոցով՝ անկախ այդ օբյեկտի իրական տիպից: Այսպիսով, մեր `print_shape_info` ֆունկցիան կարո

տեղեկություններ `area` և `perimeter` մեթոդներ ունեցող ցանկացած օբյեկտի մասին (այդ մեթոդների պարամետրերի ցանկում պետք է նշված լինի միայն մեկ պարամետր՝ `self`):

Բադային տիպավորում չապահովող լեզուներում մենք ստիպված կլինեինք ծրագրում ավելացնել ինտերֆեյս՝ որպես առանձին էություն ծրագրավորման լեզվով նկարագրության մակարդակի վրա, ինչպես նաև նշել, որ մեր դասերը պատկանում են այդ ինտերֆեյսին: Python լեզվում դա անելու կարիք չկա, սակայն ինտերֆեյսներ, այնուամենայնիվ, գոյություն ունեն:

Կարևոր է

Որպեսզի պոլիմորֆիզմն աշխատի, դրան անհրաժեշտ է հետևել ինչպես **շարահյուսության մակարդակով** (մեթոդների նույն անուններ և պարամետրերի հավասար քանակ), այնպես էլ **իմաստային մակարդակով** (նույն անուններով մեթոդները կատարում են նույն գործողությունները, մեթոդների պարամետրերն ունեն նույն իմաստը):

Եկե՛ք սահմանենք `Circle` -ին և `Square`-ին նման ինտերֆեյս ունեցող ևս մեկ դաս, օրինակ `Rectangle`(ուղղանկյուն): Եթե ամեն ինչ ճիշտ անենք, ապա `print_shape_info` ֆունկցիան կկարողանա աշխատել դրա օրինակների հետ՝

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

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


rect = Rectangle(10, 15)
print_shape_info(rect)  # Area = 150, perimeter = 50.
```

Եվս մեկ անգամ ուշադրություն դարձրեք, որ բադային տիպավորումը թույլ է տալիս նախապես գրել ֆունկցիա, որը կարող է աշխատել ցանկացած դասի բոլոր նմուշահատերի հետ՝ նույնիսկ դեռ գոյություն չունեցող: Կարևոր է միայն, որ այդ դասերն ապահովեն ֆունկցիային անհրաժեշտ ինտերֆեյսը:

Կարևոր է

Եվս մեկ դիտողություն ինկապսուլյացիայի մասին: Բանն այն է, որ ի սկզբանե, որպես կանոն, գոյություն ունի ոչ թե երկու դաս, ինչպես մեր օրինակում, այլ միայն մեկ դաս: Թող դա լինի `square`. դասը: Եթե դրանում չինկապսուլացնենք `side` հատկությունը և նախապես չսահմանենք մակերեսի և պարագծի հաշվարկման ինտերֆեյսը, ապա ոչ մի պոլիմորֆիզմ չենք ունենա: Կարևոր է հիշել, որ ինկապսուլյացիան սահմանում է դասի ինտերֆեյսի հասկացությունը և հիմք է ստեղծում պոլիմորֆիզմի համար:

2. Օբյեկտի տիպի ստուգում

Օբյեկտների հետ աշխատելիս անհրաժեշտ է լինում կատարել այս կամ այն գործողությունները՝ դրանց տիպից կախված: `isinstance()` ներկառուցված ֆունկցիայի օգնությամբ մենք կարող ենք ստուգել օբյեկտի տիպը:

Ֆունկցիա isinstance

Այդ ֆունկցիան ընդունում է երկու պարամետր `isinstance(object, type)`

Առաջին պարամետրը ներկայացնում է օբյեկտը, իսկ երկրորդը՝ այն տիպը, որի պատկանելիության առումով կատարվում է ստուգումը: Եթե օբյեկտը պատկանում է նշված տիպին, ապա ֆունկցիան վերադարձնում է True արժեքը:

```
for person in people:
    if isinstance(person, Student):
        print(person.university)
    elif isinstance(person, Employee):
        print(person.company)
    else:
        print(person.name)
    print()
```