Лекція 23

Об'єктно-орієнтоване програмування

(продовження)

Поліморфізм

Парадигма об'єктно-орієнтованого програмування крім спадкування включає ще одну важливу особливість — поліморфізм.

Слово «поліморфізм» можна перевести як «багато форм».

В ООП (об’єктно-орієнтованому програмуванні) цим терміном позначають можливість використання того самого імені операції або методу до об'єктів різних класів, при цьому дії, виконувані з об'єктами, можуть суттєво різнитися. Тому можна сказати, що в одного слова багато форм.

Приклад 1. Метод в одному класі

```
class First:
    def func(self,a,b):
        return a+b
m=First()
result=m.func(25,75)
print(result)
result=m.func("При", "віт")
print(result)
result=m.func(True, True)
print(result)
Результат:100
Привіт
2
```

Використовуємо один і той же метод, який дає різну реакцію при вводі різних даних

Однойменні методи в різних класах

Два різні класи можуть містити метод з назвою **`total`**. Інструкції у цих методах можуть передбачати зовсім різні операції:

Розглянемо приклад двох класів:

Клас FClass містить метод total, який виконує додавання до вхідного параметру числа 10.

Клас SClass містить метод **`total`**, який виконує підрахунок кількості символів у вхідному параметрі.

Залежно від того, до об'єкта якого класу застосовується метод **`total`**, виконуються ті або інші інструкції.

Приклад 2.

```
class FClass:
      n=10
      def total(self,N):
            self.total = int(self.n) + int(N)

class SClass:
      def total(self,s):
            self.total = len(str(s))
f = FClass()
s = SClass()
f.total(45)
s.total(45)
print (f.total) # Вивід: 55
print (s.total) # Вивід: 2
Результат виконання: 55 2
```

Різні класи с різним конструктором

Розглянемо програму

Програма запрошує введення числа користувачем.

Якщо число належить до діапазону від -100 до 100, то створюється об'єкт одного класу One.

Якщо число не належить до діапазону від -100 до 100, то створюється об'єкт класу Two.

В обох класах повинен бути метод-конструктор `__init__`, який у першому класі підносить число до квадрату, а в другому – множить на два.

Приклад 3.

```
class One:
    def __init__(self,a): self.a = a ** 2
    def __str__(self):
        return '[Піднесли до квадрата: %s]' % (self.a)
class Two:
    def __init__(self,a): self.a = a * 2
    def __str__(self):
        return '[Подвоїли: %s]' % (self.a)
a = input ("Введіть число: ")
a = int(a)
if -100 < a < 100: obj = One(a)
else:
    obj = Two(a)
print (obj)
```

Результат виконання:

Введіть число: 12

[Піднесли до квадрата: 144]

Введіть число: 128

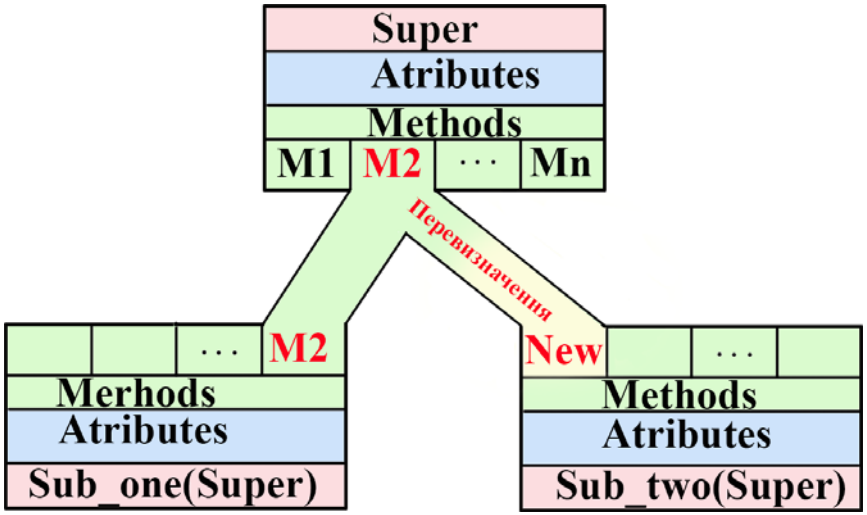
[Подвоїли: 256]

Перевизначення методів

Використання поліморфізму та спадкування класів дозволяє перевизначати методи суперкласів у підкласах.

Наприклад, може виникнути ситуація, коли всі підкласи реалізують визначений метод із суперкласу, і лише один підклас повинен мати його іншу реалізацію.

У такому випадку метод перевизначають в підкласі.



Приклад 4.

```
class Base:
      def __init__(self,n):
            self.numb = n
      def out(self):
            print (self.numb)
class One(Base):
      def multi(self,m):
            self.numb *= m
class Two(Base):
      def inlist(self):
            self.inlist = list(str(self.numb))
      def out(self):
            i = 0
            while i < len(self.inlist):
                  print (self.inlist[i])
                  i += 1
```

```
obj1 = One(45)
obj1.out()
obj1.multi(2)
obj1.out()
obj2 = Two('abc')
obj2.inlist()
obj2.out()
Результат виконання: 45 90 a b c
```

У цьому випадку об'єкт **obj1** використовує метод **out** з суперкласу **Base**, а **obj2** – зі свого класу **Two**. Атрибути шукають «знизу нагору»: спочатку в класах, потім суперкласах. Оскільки для **obj2** атрибут **out** уже був знайдений у класі **Two**, то з класу **Base** він не використовується. Інакше кажучи, клас **Two** перевизначає атрибут суперкласу **Base**.

Домішки і їх використання

Множинне спадкування, підтримуване Python, дозволяє реалізувати цікавий спосіб розширення функціональності класів за допомогою так званих ***домішок*** (`mixins`).

Домішка – це клас, що включає які-небудь атрибути й методи, які необхідно додати до інших класів.

Оголошуються вони так само, як і звичайні класи.

Приклад 5. Клас-домішка

```
class Mixin:      # Визначаємо сам клас-домішку
    attr = 0              # Визначаємо атрибут домішки
    # Визначаємо метод домішки
    def mixin_method(self):
        print("Метод домішка")

m = Mixin
m.mixin_method(m)
```

```
Результат виконання:  Метод домішка
```

Тепер оголосимо два класи, додамо до їхньої функціональності ту, що визначена в класі домішці **`Mixin`**, і перевіримо її в дії.

Приклад 6. Розширення функціональності класів за допомогою домішки

```
class Mixin:   # Визначаємо сам класс-домішку
    attr = 0   # Визначаємо атрибут домішки
    def mixin_method(self):
        # Визначаємо метод домішки
        print("Метод домішки")

class MixClass(Mixin):
    def MC_method(self):
        print("Метод класу MixClass")
class SubClass (MixClass):
    def Sub_method(self):
        print("Метод класу SubClass")
c1=MixClass()
```

```
c1.MC_method()
c1.mixin_method ()# метод домішки
c2=SubClass()
c2.MC_method()
c2.Sub_method()
c2.mixin_method()# метод домішки
Результат виконання коду:
  Метод класу MixClass
  Метод домішки
  Метод класу MixClass
  Метод класу SubClass
  Метод домішки
```

Домішки активно застосовуються в різних додаткових бібліотеках – зокрема, у популярній бібліотеці Web-програмування `Django`.

У прикладі 6 в методі класу `Subclass` викликають метод іншого класу (у цьому випадку його суперкласу; однак може викликатися метод, що не належить власному суперкласу):

Приклад 7.

```
class Base:
      def __init__(self,N):
            self.numb = N
      def out(self):
            self.numb /= 2
            print (self.numb, end=" ")
class Subclass(Base):
      def out(self):
            print ("\n----")
            Base.out(self)
            print ("\n----")
```

```
i = 0
while i < 10:
      if 4 < i < 7:
            obj = Subclass(i)
      else:
            obj = Base(i)
      i += 1
      obj.out()
```

Для доступу до атрибутів і методів можна використовувати й наступні функції:

Функції доступа до атрибутів та методів

Функція `getattr()` – повертає значення атрибута по його назві, заданій у **вигляді рядка**.

За допомогою цієї функції можна сформувати ім'я атрибута динамічно під час виконання програми. Формат функції:

getattr(<Об'єкт>, <Атрибут>[, <Знач. за замовч>])

Якщо зазначений атрибут не знайдений, виконується виключення `Attributeerror`.

Щоб уникнути виводу повідомлення про помилку, можна в третьому параметрі вказати значення, яке буде повертатися, якщо атрибут не існує.

Застосування функції `getattr()`

Приклад 8.

```
class Base:
      tb=10
class One(Base):
      to=20
class Two(Base):
      tt=30
x=Base()
y=One()
z=Two()
L=[(x,"tb"),(y,"to"),(z,"tt")]
for i,j in L:
   print(getattr(i,j), end=" ")
```

Результат роботи :

```
10 20 30
```

Функція `setattr ()` – задає значення атрибута. Назва атрибута вказується **у вигляді рядка**.

Формат функції:

```
setattr(<Об'єкт>, <Атрибут>, <Значення>)
```

Другим параметром методу `setattr()` можна передати ім'я неіснуючого атрибута

Якщо атрибут не існує, то він буде створений із зазначеним іменем.

Приклад 9. Застосування функції `setattr()`

```
class One:
      to=10
class Two:
      tw=20
x=One()
y=Two()
L=[(x,"to",1),(y,"tw",2)]
for i,j,k in L:
     setattr(i,j,k)
print(x.__dict__)
print(y.__dict__)
```

Результат роботи:

```
{'to': 1}
{'tw': 2}
```

Функція delattr (<Об'єкт>, <Атрибут>) – видаляє зазначений атрибут. Назва атрибуга вказується **у вигляді рядка**.

Приклад 10

```
class One():
   def __init__(self,a,b):
          self.t3 = a
          self.t4 = b
x=One(20,50)
print("Атрибути до видалення",x.__dict__)
delattr(x,"t3")
print("Атрибути після видалення",x.__dict__)
```

Результат роботи:

```
Атрибути до видалення:{'t4': 50, 't3': 20}
Атрибути після видалення: {'t4': 50}
```

Функція `hasattr (<Об'єкт>, <Атрибут>)` – перевіряє наявність зазначеного атрибута. Якщо атрибут існує, функція повертає значення `True`.

Назва атрибуга вказується **у вигляді рядка**.

Продемонструємо роботу функцій на прикладі.

Приклад 11.

```
class MyClass:
    def __init__(self):
        self.x = 10
    def get_x(self):
        return self.x
c = MyClass ()
print(hasattr(c, "x"))    # Виведе: True
print(hasattr(c, "y"))    # Виведе: False
```

Принципи доступу до атрибутів

1.Усі атрибути класу в мові Python є відкритими (`public`)

Отже атрибути є доступними для безпосередньої зміни:

- із самого класу,
- з інших класів,
- з основного коду програми.

Атрибути можна створювати динамічно.

Можна створити як атрибут об'єкта класу, так і атрибут екземпляра класу.

Розглянемо це на прикладі:

Приклад 12. Атрибути класу й екземпляра класу

```
class MyClass: # Визначаємо порожній клас
        pass
MyClass.x = 50 # Створюємо атрибут об'єкта класу
c1, c2 = MyClass (), MyClass ()
# Створюємо два екземпляри класу
c1.y = 10    # Створюємо атрибут екземпляра класу
c2.y = 20   # Створюємо атрибут екземпляра класу
print(c1.x, c1.y) # Виведе: 50 10
print(c2.x, c2.y) # Виведе: 50 20
```

У цьому прикладі ми визначаємо порожній клас, розмістивши в ньому оператор **`pass`**. Далі створюємо атрибут класу **`x`**. Цей атрибут буде доступний усім створюваним екземплярам класу. Потім створюємо два екземпляри класу і додаємо однойменні атрибути **`y`**. Значення цих атрибутів будуть різними в кожному екземплярі класу. Але якщо створити новий екземпляр (наприклад, `c3`), то атрибут `y` в ньому визначений не буде. Таким чином, за допомогою класів можна імітувати типи даних, підтримувані іншими мовами програмування (наприклад, тип `struct`, доступний у мові C).

Дуже важливо розуміти різницю між атрибутами класу й атрибутами екземпляра класу.

Визначення атрибуів класу
та атребутів екземпляру класу

Атрибут класу доступний усім екземплярам класу, але після зміни атрибута значення зміниться у всіх екземплярах класу.

Атрибут екземпляра класу може зберігати унікальне значення для кожного екземпляра, і зміна його в одному екземплярі класу не торкнеться значень однойменного атрибута в інших екземплярах того ж класу. Розглянемо це на прикладі, створивши клас з атрибутом класу (`x`) і атрибутом екземпляра класу (`y`):

Приклад 13. Зміна атрибуту класу

```
class MyClass:
    x = 10   # Атрибут класу
    def __init__(self):
        self.y = 20  # Атрибут екземпляра класу
```

Тепер створимо два екземпляри цього класу:

```
c1 = MyClass()  # Створюємо екземпляр класу
c2 = MyClass()  # Створюємо екземпляр класу
```

Виведемо значення атрибута `x`, а потім змінимо значення й знову виведемо:

```
print(c1.x, c2.x) # 10 10
MyClass.x = 88 # Змінюємо атрибут класу
print(c1.x, c2.x) # 88 88
```

Як видно з прикладу, зміна атрибута класу торкнулася значення у двох екземплярах класу відразу.

Зміна атрибуту екземпляра класу

Приклад 14.

```
print(c1.y, c2.y)  # 20 20
c1.y = 88 # Змінюємо атрибут екземпляра класу
print(c1.y, c2.y) # 88 20
```

У цьому випадку зміна в екземплярі `c1`.

Атрибут екземпляра може бути заданий безпосередньо в екземплярі класу. **Приклад 15.**

```
class MyClass:
    x = 10  # Атрибут класу
c1 = MyClass() # Створюємо екземпляр класу
c2 = MyClass() # Створюємо екземпляр класу
print(c1.x, c2.x) # 10 10
c1.y = 88 # Змінюємо атрибут класу
print(c1.x, c1.y) # 10 88
c2.z = 100
print(c2.x, c2.z)  # 10 100
```

Одночасне існування атрибута атрибута класу та екземпляра класу з однаковим іменем

При одночасному існуванні атрибутів класу та екземпляра класу с одним іменем доступ до них потрібно виконувати користуючись таким правилом.

1. Доступ до атрибуту класу виконувати з безпосереднім вказуванням імені класу через крапку.

2. Доступ до атрибуту екземпляру класу виконувати з безпосереднім вказуванням імені екземпляра класу через крапку.

Приклад використання атрибутів класу та екземпляру з однаковим іменем

Приклад 16.

```
class MyClass:
    x = 10  # Атрибут класу
c = MyClass() # Створюємо екземпляр класу
MyClass.x = 100    # Змінюємо атрибут об'єкта класу
print(c.x)
c.x = 200  # Створюємо атрибут екземпляра
print(c.x, MyClass.x)  # 200 100.
```

Спеціальні методи
Метод __call__()

Метод `__call__()` викликається при виклику екземпляра класу, як виклик функції.

Формат методу:

__call__(self[, <Парам1>[,...,<Парамn>]])

Приклад 17. Мінімальний код

```
class A:
    def __call__(self):
        print("Виклик у стилі функції")
a=A()
a()
```

Приклад 18. Застосування методу __call__

```
class MyClass:
    def __init__(self, m):
        self.msg = m
    def __call__(self):
        print(self.msg)
        return "Повертаємо "+self.msg
x = MyClass("Повідомлення об'єкта x")
# Створення екземпляра класу
y = MyClass("повідомлення y")
# Створення екземпляра класу
x()
b=y()
print(b)
```

Результат: Повідомлення об'єкта x
повідомлення y
Повертаємо повідомлення y

Метод __getattr__()

Метод `__getattr__()` – викликається при спробі доступу до неіснуючого атрибуту класу.

Формат методу:

```
__getattr__(self, <Атрибут>)
```

Приклад 19. Мінімальний код

```
class A:
    def __getattr__(self,b):
      print("Викликано метод __getattr__()")

a=A()
a.c
```

Результат

```
Викликано метод __getattr__()
```

Приклад 20.

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattr__(self, attr):
        print("Метод __getattr__()")
        return 0
c = MyClass()
# Атрибут i існує
print(c.i)
print(c.s)

Результат.
20
Метод __getattr__()
0
```

Метод `__getattribute__()`

Метод викликається при доступі до будь-якого атрибута класу.

Формат методу:

```
__getattribute__(self, <Атрибут>)
```

Необхідно враховувати, що використання точкової нотації (для доступу до атрибута класу) всередині цього методу призведе до зациклення. Щоб уникнути зациклення, слід викликати метод `__getattribute__()` об'єкта `object`. Всередині методу потрібно повернути значення атрибута або виконати виключення `Attributeerror`.

Приклад 21.

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattribute__(self, attr):
        print("Метод __getattribute__()")
        return object.__getattribute__(self, attr)  # Тільки так!!!
c = MyClass ()
print(c.i)
```

Результат

```
Метод __getattribute__()
20
```

Метод `__setattr__()`

Метод викликається при спробі присвоювання значення атрибуту екземпляра класу.

Формат методу:

```
__setattr__(self,<Атрибут>,<Значення>)
```

Якщо всередині методу необхідно присвоїти значення атрибуту, то слід використовувати словник `__dict__`, інакше при точковій нотації метод `__setattr__()` буде викликаний повторно, що призведе до зациклення.

Приклад 22.

```
class MyClass:
    def __setattr__(self, attr, value):
        print("Метод __setattr__()")
        self.__dict__[attr] = value  # Тільки так!!!

c = MyClass()
c.i = 10
print(c.i)
```

```
Результат:
Метод __setattr__()
10
```

Метод __len__()

Метод викликають при використанні функції `len()`

Формат методу :

```
__len__ (self)
```

Метод повинен повертати додатне ціле число.

Приклад 23.

```
class MyClass:
    x="Line"
    def __len__(self):
        b=len(self.x)
        return b
c = MyClass()
print(len(c))
```

Метод `__bool__(self)` – викликають при використанні функції `bool()`.

Метод `__int__(self)` – викликають при перетворенні об'єкта в ціле число за допомогою функції `int()`.

Метод `__float__(self)` – викликають при перетворенні об'єкта в дійсне число за допомогою функції `float()`.

Метод `__complex__(self)` – викликають при перетворенні об'єкта в комплексне число за допомогою функції `complex()`.

Метод `__round__(self, n)` – викликають при використанні функції `round()`.

Метод `__index__(self)` – викликають при використанні функцій `bin()`, `hex()` і `oct()`.

Методи `__repr__(self)` і `__str__(self)`

Методи `__repr__(self)` і `__str__(self)` – служать для перетворення об'єкта в рядок.

Метод `__repr__()` викликають при виводі в інтерактивній оболонці, а також при використанні функції `repr()`.

Метод `__str__()` викликають при виводі за допомогою функції `print()`, а також при використанні функції `str()`.

Якщо метод `__str__()` відсутній, то буде викликаний метод `__repr__()`.

Як значення методи `__repr__()` і `str()` повинні повертати рядок.

Приклад 24.

```
class MyClass:
    def __init__(self, m):
        self.msg = m
    def __repr__(self):
        return "Метод  __repr__()  {0}".format(self.msg)
    def __str__ (self):
        return "Метод __str__() {0}".format(self.msg)
c = MyClass("Значення")
print(repr(c))
print(str(c))
print(c)
```

Результат роботи:

Викликаний метод __repr__() Значення
Викликаний метод __str__() Значення
Викликаний метод __str__() Значення

Метод __hash__(self)

Метод перевизначити, якщо екземпляр класу заплановано використовувати як ключ словника або всередині множини.

Приклад 25.

```
class MyClass:
    def __init__ (self, y):
        self.x = y
    def hash (self) :
        return hash(self.x)

m = MyClass(10)
d = {}
d[m] = "Значення"
print(d[m])
Результат: Значення
```