Практичне заняття №4а

з навчальної дисципліни

Спеціалізовані мови програмування

на тему:

ОБЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Зміст

Методи класу мають одну відмінність від звичайних функцій: вони повинні мати додатково ім'я, яке додається до початку списку параметрів. Однак, при виклику методу ніякого значення цьому параметру привласнювати не потрібно - його вкаже Python. Ця змінна вказує на сам об'єкт екземпляра класу, і за традицією вона називається **self**

У нас є клас з ім'ям **MyClass** і екземпляр цього класу з ім'ям **myobject**

myobject.method(arg1, arg2)

Python автоматично перетворює це в

MyClass.method(myobject, arg1, arg2)

```
class Person:
    pass # порожній блок

p = Person()
print(p)
```

```
class Person:
    def sayHi(self):
        print('Hello! How are you?')

p = Person()
p.sayHi()

# або

Person().sayHi()
```

Метод **__init__** запускається, як тільки об'єкт класу реалізується. Цей метод корисний для здійснення різного роду ініціалізації, необхідної для даного об'єкта. Зверніть увагу на подвійні підкреслення на початку і в кінці імені.

```
class Person:
    def __init__(self, name):
        self.name = name
    def sayHi(self):
        print('Hi! My name is ', self.name)

p = Person('Tom')
p.sayHi()

# або

Person("Tom").sayHi()
```

Тут ми визначаємо метод **__init__** так, щоб він приймав параметр **name** (Поряд зі звичайним self). Далі ми створюємо нове поле з ім'ям **name**. Зверніть увагу, що це дві різні змінні, навіть незважаючи на те, що вони обидві названі name. Це не проблема, так як точка в вираженні **self.name** позначає, що існує щось з ім'ям «**name**», що є частиною об'єкта «**self**», і інше **name** - локальна змінна.

Дані, тобто поля, є не чим іншим, як звичайними змінними, укладеними в просторі імен класів і об'єктів. Це означає, що їх імена дійсні тільки в контексті цих класів або об'єктів. Звідси і назва «**простір імен**».

Змінні класу роздільні - доступ до них можуть отримувати всі екземпляри цього класу. Змінна класу існує тільки одна, тому коли будь-який з об'єктів змінює змінну класу, ця зміна відіб'ється і у всіх інших екземплярах того ж класу.

Змінні об'єкта належать кожному окремому екземпляру класу. В цьому випадку у кожного об'єкта є своя власна копія поля, тобто не спільна і жодним чином не пов'язана з іншими такими ж полями в інших екземплярах.

```
class Robot:
    '''Представляє робота з ім'ям.'''
    # Змінна класу, що містить кількість роботів
    population = 0

    def __init__(self, name):
        '''ініціалізація даних.'''
        self.name = name
        print('(ініціалізація {0})'.format(self.name))

        # При створенні цієї особистості, додається робот
        # до змінної 'population'
        Robot.population += 1

    def __del__(self):
        """ I'm dying."""
        print('{0} destroyed!'.format(self.name))
        Robot.population -= 1
        if Robot.population == 0:
            print('{0} was the last.'.format(self.name))
        else:
            print('залишилося {0:d} працюючих роботів.'.format(Robot.population))
```

```
def sayHi(self):
    '''Привітання робота. '''
    print('Вітаю! Мої господарі називають мене {0}.'.format(self.name))

  def howMany():
    '' Виводить чисельність роботів.'''
    print('У нас {0:d} роботів.'.format(Robot.population))

  howMany = staticmethod(howMany)
```

```
droid1 = Robot('R2-D2')
droid1.sayHi()
Robot.howMany()

droid2 = Robot('C-3PO')
droid2.sayHi()
Robot.howMany()
print("\nтут роботи можуть виконати якусь роботу.\n")
print(" Роботи закінчили свою роботу. Давайте знищимо їх.")
del droid1
del droid2
Robot.howMany()
```

Тут **population** належить класу **Robot**, і тому є змінною класу. Змінна **name** належить об'єкту (їй присвоюється значення за допомогою **self**), і тому є змінною об'єкта.

Таким чином, ми звертаємося до змінної класу **population** як **Robot.population**, а не **self.population**. До змінної ж об'єкта name у всіх методах цього об'єкта ми звертаємося за допомогою позначення **self.name**.

Одне з головних достоїнств об'єктно-орієнтованого програмування полягає в багаторазовому використанні одного і того ж коду, і один із способів цього досягти - за допомогою механізму **наслідування**. Найлегше уявити собі спадкування у вигляді відносини між класами як **тип** і **підтип**.

```
class SchoolMember:
    '''Представляет любого человека в школе.'''

    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Создан SchoolMember: {0})'.format(self.name))

    def tell(self):
        '''Вывести информацию.'''
        print('Имя:"{0}" Возраст:"{1}"'.format(self.name, self.age), end=" ")
```

```
class Teacher(SchoolMember):
    '''Представляет преподавателя.'''

    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Создан Teacher: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Зарплата: "{0:d}"'.format(self.salary))
```

```
class Student(SchoolMember):
    '''Представляет студента.'''

    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Создан Student: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Оценки: "{0:d}"'.format(self.marks))
```

```
t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)
print()

members = [t, s]
for member in members:
    member.tell()
```

Точно так же, як класи використовуються для створення об'єктів, можна використовувати **метакласи** для створення класів. Метакласи існують для зміни або додавання нової поведінки в класи.

Припустимо, ми хочемо бути впевнені, що ми завжди створюємо виключно екземпляри підкласів класу **SchoolMember**, і не створюємо екземпляри самого класу **SchoolMember**.

Для досягнення цієї мети ми можемо використовувати концепцію під назвою «**абстрактні базові класи**». Це означає, що такий клас **абстрактний**, тобто є лише якоюсь **концепцією**, яка *не призначена для використання в якості реального класу*.

```
from abc import *

class SchoolMember(metaclass=ABCMeta):
    '''Представляє будь-яку людину в школі.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Створено SchoolMember: {0})'.format(self.name))

    @abstractmethod
    def tell(self):
        '''Вывести информацию.'''
        print('Імя:"{0}" Вік:"{1}"'.format(self.name, self.age), end=" ")
```

```
class Teacher(SchoolMember):
    ''' Представляє викладача.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(створено Teacher: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Зарплата: "{0:d}"'.format(self.salary))
```

```
class Student(SchoolMember):
    '''Представляє студента.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(створено Student: {0})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Оцінки: "{0:d}"'.format(self.marks))
```

```
t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)
#m = SchoolMember('abc', 10)
# Це призведе до помилки : "TypeError: Can't instantiate abstract
class
# SchoolMember with abstract methods tell"

print() members = [t, s]
for member in members:
    member.tell()
```

Лекцію закінчено.
Дякую за увагу!