Програмування

ТЕМА 19. МЕТАКЛАСИ ТА МЕТАПРОГРАМУВАННЯ

Метакласи

Ми вже розглядали застосування класів та об’єктів, одинарне та множинне наслідування, наслідування та агрегування між класами, використання класів для обробки виключень.

Ці області застосування об’єктно-орієнтованого підходу є стандартними.

Але існують і більш екзотичні сценарії використання класів, у тому числі, пов’язані зі специфікою їх реалізації у Python.

Це, зокрема, метакласи.

Ми вже знаємо, що у Python «все є об’єкт».

Було б дивно, якби це правило не розповсюджувалось на класи. І дійсно, класи у Python також є об’єктами.

Але виникає питання: об’єктами чого є класи? Відповідь: класи є об’єктами інших класів.

Для того, щоб уникнути плутанини, ці «інші» класи мають спеціальну назву – метакласи.

Отже, **метакласом** називається клас, об’єктами якого є класи.

Так само, як клас створює об’єкти, метаклас створює класи.

Метакласи.2

Усі метакласи походять від єдиного стандартного батьківського класу type.

Для метакласів існує наслідування, як і для звичайних класів.

Таким чином, будь-який метаклас наслідує безпосередньо від type або його нащадків.

Отже, якщо маємо опис

```
class MyMeta(type):
    Q
```

- то MyMeta – це метаклас.

Метакласи.3

Кожний «звичайний» клас обов’язково має метаклас.

Якщо метаклас не вказано, то цим метакласом вважається type.

Якщо ж треба вказати інший метаклас, відмінний від type, це робиться за допомогою ключового параметра metaclass у заголовку класу разом з вказанням батьківських класів

```
class MyClass(metaclass = MyMeta):
    P
```

У цьому випадку метакласом класу MyClass буде MyMeta.

Метакласи.4

За висловом Тіма Пітерса, «метакласи є більш глибокою магією, про яку 99% користувачів не повинні турбуватись. Якщо, ви розмірковуєте, чи потрібні вони вам, - вони вам непотрібні».

Але ми розглянемо декілька реальних сценаріїв, у яких метакласи дійсно потрібні.

Абстрактні класи

Клас називається **абстрактним**, якщо його об’єкти не можуть бути створені самі по собі, але він є батьківським класом у деякій ієрархії класів.

Наприклад, клас Shape – геометрична фігура.

Створення об’єктів цього класу не має сенсу, але його класи-нащадки – Circle (коло) Rectangle (прямокутник) вже є конкретними класами, для яких можуть існувати об’єкти.

Можна сказати, що абстрактний клас задає певну сигнатуру, яку потім реалізують його класи-нащадки.

У Python клас вважається абстрактним, якщо він містить хоча б один абстрактний метод.

Абстрактні класи.2

Абстрактний метод позначається за допомогою декоратора

```
@abstractmethod
def method():
    pass
```

- де method – ім’я методу.

Тіло абстрактного методу – це тотожна команда pass.

Абстрактні класи.3

Абстрактними можуть бути також властивості.

У такому випадку треба послідовно вказати декоратори:

```
@property
@abstractmethod
def prop():
    pass
```

- де prop – ім’я властивості.

Абстрактний клас обов’язково повинен мати в якості метакласу стандартний метаклас ABCMeta або один з його нащадків.

Інакше декоратор @abstractmethod просто не буде працювати.

Приклад: зображення точок та кіл

Скласти модуль для зображення та переміщення точок та кіл по екрану.

Цю задачу ми вже розглядали у темі «Класи та об’єкти».

Ми описували 2 класи: Point та Circle, які використовували графічну бібліотеку turtle.

Пізніше у темі «декоратори» ми модифікували ці класи, задавши властивості.

Уявімо, що нам може знадобитися інша графічна бібліотека замість turtle для зображення точок та кіл.

Як зробити, щоб у цьому випадку нам не довелося переписувати повністю наші класи Point та Circle?

Реалізація зображення точок та кіл

Розв’язання полягає у використанні абстрактного класу для зображення точок та кіл, включення об’єкту цього класу як поля у класи Point та Circle та опису нащадка цього класу, який буде використовувати бібліотеку turtle.

Цей абстрактний клас ми назвемо Drawable, а клас-нащадок, що використовує turtle, - TurtleDraw.

Тепер, якщо виникне необхідність застосувати іншу графічну бібліотеку, нам достатньо буде описати клас-нащадок Drawable з відповідними методами, не змінюючи описи класів Point та Circle.

Клас Drawable має властивості читання/встановлення кольорів переднього плану та фону а також методи зображення точки та кола з заданими координатами та кольором (для кола також радіусом).

Реалізація зображення точок та кіл. Drawable

```
class Drawable(metaclass = ABCMeta):
    """Абстрактний клас для зображення точок та кіл заданих розмірів та кольору"""

    @property
    @abstractmethod
    def color(self):
        """Властивість, що повертає/встановлює колір переднього плану."""
        pass

    @color.setter
    @abstractmethod
    def color(self, cl):
        pass
```

Реалізація зображення точок та кіл. Drawable.2

```
    @property
    @abstractmethod
    def bgcolor(self):
    """Властивість, що повертає/встановлює колір фону."""
        pass

    @bgcolor.setter
    @abstractmethod
    def bgcolor(self, cl):
        pass
```

Реалізація зображення точок та кіл. Drawable.3

```
    @abstractmethod
    def draw_point(self, x, y, cl):
    """Зобразити точку з координатами x, y кольором cl."""
        pass

    @abstractmethod
    def draw_circle(self, x, y, r, cl):
     """Зобразити коло з координатами центру x, y радіусом r кольором cl."""
        pass
```

Реалізація зображення точок та кіл. TurtleDraw

Клас TurtleDraw реалізує властивості та методи, описані у абстрактному класі Drawable, а також має конструктор для ініціалізації бібліотеки turtle.

```
class TurtleDraw(Drawable):
    """Клас для зображення точок та кіл заданих розмірів
та кольору.
     TurtleDraw є нащадком абстрактного класу Drawable
та використовує засоби
    роботи з графікою з модуля turtle.    """

    def __init__(self):
        pause = 50
        turtle.up()
        turtle.home()
        turtle.delay(pause)
```

Реалізація зображення точок та кіл. TurtleDraw.2

```
@property
  def color(self):
      """Властивість, що повертає/встановлює колір переднього плану."""
      return turtle.pencolor()

  @color.setter
  def color(self, cl):
      turtle.pencolor(cl)

  @property
  def bgcolor(self):
      """Властивість, що повертає/встановлює колір фону."""
      return turtle.bgcolor()

  @bgcolor.setter
  def bgcolor(self, cl):
      turtle.bgcolor(cl)
```

Реалізація зображення точок та кіл. TurtleDraw.3

```
def draw_point(self, x, y, cl):
    """Зобразити точку з координатами x, y кольором cl."""
    turtle.up()
    turtle.setpos(x, y)
    turtle.down()
    turtle.dot(cl)

  def draw_circle(self, x, y, r, cl):
    """Зобразити коло з координатами центру x, y радіусом r кольором cl."""
    c = self.color
    self.color = cl
    turtle.up()
    turtle.setpos(x, y-r) #малює починаючи знизу кола
    turtle.down()
    turtle.circle(r)
    self.color = c
```

Реалізація зображення точок та кіл. Point та Circle

Класи Point та Circle містять поле _d, яке під час створення повинно набувати значення об’єкта класу, що є нащадком абстрактного класу Drawable.

Для класу Point реалізація виглядає так:

```
class Point:
    '''Точка екрану'''
    count = 0

    def __init__(self, x, y, drawable):
        self._x = x             # _x - координата x точки
        self._y = y             # _y - координата y точки
        self._visible = False   # _visible - чи є точка видимою на екрані
        self._d = drawable()    # _d - об'єкт класу-нащадка
                                # абстрактного класу Drawable
        Point.count += 1
```

Реалізація зображення точок та кіл. Point та Circle.2

```
@property
def x(self):
    '''Повертає координату x точки'''
    return self._x

@property
def y(self):
    '''Повертає координату y точки '''
    return self._y

@property
def onscreen(self):
    '''Перевіряє, чи є точка видимою на екрані'''
    return self._visible
```

Реалізація зображення точок та кіл. Point та Circle.3

```
def switchon(self):
    '''Робить точку видимою на екрані '''
    if not self._visible:
        self._visible = True
        self._d.draw_point(self._x,self._y,self._d.color)

def switchoff(self):
    '''Робить точку невидимою на екрані'''
    if self._visible:
        self._visible = False
        self._d.draw_point(self._x,self._y,self._d.bgcolor)
```

Реалізація зображення точок та кіл. Point та Circle.4

```
def move(self, dx, dy):
'''Пересуває точку на екрані на dx, dy позицій'''
    vis = self._visible
    if vis:
        self.switchoff()
    self._x += dx
    self._y += dy
    if vis:
        self.switchon()
@staticmethod
 def printcount():
    print('Кількість точок:', Point.count)
```

У основній частині модуля створення об’єкту класу Point виконується командою

```
p = Point(50,50,TurtleDraw)
```

Основна частина модуля створює точку та коло, а потім переміщує їх по екрану.

Метапрограмування

Метапрограмування – це побудова програм, які сприймають інші програми як дані.

Тобто, таких програм, які модифікують програми «на льоту».

Розглянуті раніше декоратори функцій можуть вважатись прикладом метапрограмування.

У Python термін метапрограмування має більш вузьку трактовку – це модифікація класів під час виконання програм.

Оскільки клас – це об’єкт, клас може бути модифікований під час виконання програми.

Така модифікація розповсюджується як на зміну властивостей/поведінки вже існуючих класів, так і на створення цілком нових класів.

Декоратори класів

Одним з прийомів метапрограмування є застосування декораторів класів.

Раніше ми розглядали декоратори функцій.

Побудова декораторів класів не складніша за побудову декораторів функцій.

Декоратор класу – це функція, що отримує клас як параметр, модифікує його та повертає модифікований клас в якості результату.

Стандартний шаблон декоратора класів виглядає так

```
def cls_decorator(cls):
    #модифікувати клас
    return cls
```

Декоратори класів.2

Для застосування декоратора до класу A треба вказати

```
@cls_decorator
class A():
    Q
```

- де *Q* – опис полів та методів класу A.

Приклад: копіювання поліному та відслідковування дій над поліномом

Скопіювати заданий поліном.

Здійснити відслідковування виконання дій над поліномом.

Для реалізації цього завдання використаємо клас PolinomeEx, описаний у темі «Обробка помилок та виключних ситуацій».

Подібний приклад ми вже розглядали у темі «Множинне наслідування»

Зараз для реалізації застосуємо декоратор класу.

Реалізація копіювання поліному з використанням декоратора класу

Опишемо декоратор logged_mapping(), що модифікує заданий клас так, щоб здійснювати виведення факту використання операцій читання, зміни та видалення елементу деякого типу даних.

Цей декоратор містить перевизначення спеціальних методів __getitem__, __setitem__ та __delitem__, які відповідають за виконання вищевказаних функцій.

```
def logged_mapping(cls):
    '''Додати виведення операцій get/set/delete для налагодження.
    '''
    #Отримати наявні у класі методи __getitem__, __setitem__,
__delitem__
    orig_getitem = cls.__getitem__
    orig_setitem = cls.__setitem__
    orig_delitem = cls.__delitem__
```

Реалізація копіювання поліному з використанням декоратора класу.2

```
def log_getitem(self, key):
    print('Getting ' + str(key))
    return orig_getitem(self, key)

def log_setitem(self, key, value):
    print('Setting {} = {!r}'.format(key, value))
    return orig_setitem(self, key, value)

def log_delitem(self, key):
    print('Deleting ' + str(key))
    return orig_delitem(self, key)
```

Реалізація копіювання поліному з використанням декоратора класу.3

```
#Зберегти у класі модифіковані методи
__getitem__, __setitem__, __delitem__
    cls.__getitem__ = log_getitem
    cls.__setitem__ = log_setitem
    cls.__delitem__ = log_delitem
    return cls
```

Щоб не наводити повторно опис класу PolynomeEx, опишемо його клас-нащадок LoggedPolynome з порожньою реалізацією.

До класу LoggedPolynome застосуємо наш декоратор logged_mapping.

Класові методи

Для модифікації класів часто використовують так звані «класові методи».

Класовий метод – це метод, який застосовується до класу, а не до екземпляру класу.

Класовий метод містить клас (зазвичай позначений cls) як перший параметр.

Нагадаємо, що звичайний метод містить першим параметром об’єкт (self), до якого застосовується цей метод, а статичний метод не містить спеціального першого параметру.

У Python класові методи можуть бути як вбудованими, так і створеними власноруч.

Вбудовані класові методи, як і інші спеціальні методи беруться у подвійне підкреслення ‘__’ з обох боків.

Класові методи.2

Щоб описати власний класовий метод, треба застосовувати декоратор @classmethod

```
@classmethod
def meth(cls, ...):
    Q
```

Метод __new__

Розглянемо ще один особливий метод Python: __new__. Цей метод є статичним.

Метод __new__ викликається перед створенням будь-якого об’єкту будь-якого класу. Його виклик передує виклику методу __init__.

Тому __new__ застосовують для модифікації поведінки об’єкта класу (а отже – і самого класу).

Метод __new__ в якості першого параметру містить ім’я класу, а інші параметри – ті ж самі, які передаються у метод __init__. Якщо __new__ описують для модифікації довільного класу, то застосовують найбільш загальне позначення аргументів:

```
def __new__(cls, *args, **kwargs):
    pass
```

Словник атрибутів класу

Усі атрибути класу містяться у спеціальному словнику __dict__.

Ключами в ньому є імена атрибутів.

Значеннями атрибутів, що є полями, є їх значення.

Для атрибутів, що є методами, значеннями є відповідні методи.

Модифікуючи словник __dict__, можна додавати нові поля або методи, а також змінювати наявні методи іншими.

Створення класів у динаміці

Класи у Python можна створювати не тільки статично, вказуючи опис класу, але й у динаміці.

Для створення класів у динаміці можна застосувати функції type() або types.newclass().

Функція type може мати 1 або 3 аргументи.

Якщо функція має 1 аргумент, вона просто повертає тип цього аргументу.

Якщо ж type має 3 аргументи, вона створює новий клас.

Щоб створити новий клас за допомогою type, треба у її виклику вказати:

cls = type(classname, bases, cls_dict)

- де classname – ім'я нового класу (рядок), bases – кортеж, що містить класи-предки, cls_dict – словник __dict__ нового класу.

Створення класів у динаміці.2

Наприклад, опис двох функцій та створення класу

```
def __init__(self,x):
    self.x = x

def plus1(self):
    self.x = self.x +1
    print(self.x)

A = type('A', (), {'__init__':__init__, 'plus1': plus1})
```

Рівносильно такому опису класу

```
class A():
    def __init__(self,x):
        self.x = x

    def plus1(self):
        self.x = self.x +1
        print(self.x)
```

Створення класів у динаміці.3

Функція types.newclass() з модуля types відрізняється від функції type() своїми параметрами.

Вона має 4 параметри

cls = types.newclass(classname, bases, kwds, exec_body)

де classname – ім’я нового класу (рядок),

bases – кортеж, що містить класи-предки,

kwds – словник з ключовими параметрами, які вказують у заголовку опису класу (наприклад, metaclass),

exec_body – це функція яка буде викликатися при створенні класу для оновлення словника класу. Ця функція повинна мати 1 параметр – словник класу – та повертати оновлений словник.

Створення класів у динаміці.4

Як правило, у якості exec_body вказують lambda-функцію, хоча це може бути і звичайна функція. Типове значення параметру exec_body:

```
lambda ns: ns.update(cls_dict)
```

де cls_dict – словник, що містить оновлення словника класу.

Функція types.newclass() є більш гнучкою у порівнянні з type(), оскільки надає можливість вказати метаклас, а також вказати тільки зміни до словника класу.

Приклад створення класу у динаміці: іменовані кортежі

Реалізувати клас для іменованого кортежу та використати його для обчислення центру мас системи точок простору та максимальної відстані між точками.

Цей приклад був розглянутий у темі «Кортежі».

Тоді ми використали стандартну функцію namedtuple з модуля collections.

Ця функція будує новий клас іменованого кортежу за ім’ям та переліком полів, використовуючи команду exec, яка дозволяє виконати побудований рядок програмного коду.

Реалізація створення класу у динаміці: іменовані кортежі

Зараз ми розглянемо інший підхід.

Опишемо функцію named_tuple, яка будує клас іменованого кортежу в динаміці (взято з "Python Cookbook" by David Beazley and Brian K. Jones).

```
def named_tuple(classname, fieldnames):
    '''Функція створює та повертає клас, що реалізує іменований кортеж.
     classname - ім'я створюваного класу,
    fieldnames - послідовність імен полів.
    Функція повертає модифікований клас для реалізації звичайних кортежів (tuple),
    змінюючи в ньому метод __new__
    '''
```

Реалізація створення класу у динаміці: іменовані кортежі.2

```
    # Заповнити словник функціями доступу до полів кортежу за їх номерами
    # Ключі у словнику - імена полів іменованого кортежу
    # cls_dict буде містити імена атрибутів нового класу
    # operator.itemgetter(n) - функція, що повертає n-ий елемент послідовності
    cls_dict = { name: property(operator.itemgetter(n))
                 for n, name in enumerate(fieldnames) }
     # Створити нову функцію __new__ та додати до словника класу
    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            #перевірити, чи рівна кількість полів при ініціалізації об'єкта
            #кількості полів у описі класу
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        # викликати та повернути результат __new__ з tuple
        return tuple.__new__(cls, args)
```

Реалізація створення класу у динаміці: іменовані кортежі.3

```
cls_dict['__new__'] = __new__

    # Створити клас за допомогою types.new_class
    cls = types.new_class(classname, (tuple,), {},
                          lambda ns: ns.update(cls_dict))
#У даному випадку можна було використати й функцію type наступним чином:
#    cls = type(classname, (tuple,), cls_dict)

    # встановити ім'я модуля рівним імені модуля, звідки викликається named_tuple
    cls.__module__ = sys._getframe(1).f_globals['__name__']
#ненадійно!
    return cls
```

Реалізація створення класу у динаміці: іменовані кортежі.4

Ця функція будує новий клас іменованого кортежу як нащадку стандартного класу кортежів tuple.

Для побудови модифікується метод __new__.

Також у новий клас додаються методи отримання значення поля за його ім’ям (використовується функція operator.itemgetter(n) з модуля operator, що повертає відповідний елемент послідовності з номером n).

Новий клас будується з використанням функції types.newclass().

Головний модуль програми практично не відрізняється від розглянутого у темі «Кортежі», за виключенням того, що він імпортує створену нами функцію named_tuple замість стандартної функції namedtuple.

Написання власних метакласів

Власні метакласи також є засобом метапрограмування, оскільки вони здатні вносити зміни до класів або створювати нові класи.

Власний метаклас повинен бути нащадком type або одного з його нащадків.

Для того, щоб метаклас впливав на створення класу, що є його об’єктом, потрібним чином, використовують перевизначення методів __new__, __init__ або __call__ у метакласі.

Якщо з __new__ та __init__ ми вже зустрічались, то метод __call__ метакласу потребує додаткового пояснення.

Цей метод (__call__ метакласу) викликається при створенні об’єкта деякого класу перед викликами __new__ та __init__ самого класу.

Приклад використання метакласів: класи подібні структурам

Розглянемо ту ж сам задачу: обчислити центр мас системи точок простору та максимальну відстань між точками.

Якщо у попередньому прикладі ми використали функцію для побудови класу іменованого кортежу, то тепер використаємо метаклас, який будує класи, що подібні структурам у C, тобто, ті ж кортежі, доступ до елементів яких здійснюється за іменами полів.

Опишемо метаклас StructTupleMeta та клас StructTuple, що використовує StructTupleMeta як метаклас (взято з "Python Cookbook" by David Beazley and Brian K. Jones).

Приклад використання метакласів: класи подібні структурам.2

```
class StructTupleMeta(type):
    '''Метаклас, що створює клас, подібний структурі C.

    Метаклас доповнює словник створюваного класу властивостями
    з іменами, що містяться у списку _fields.
    Для читання значення кожної властивості використовується
    operator.itemgetter(n) - отримання n-го елементу послідовності'''
    def __init__(cls, name, bases, nmspc):
        """Конструктор __init__ метакласу.
         Приймає параметри для створення класу (не об'єкту!).
        cls - клас, name - його ім'я,
        bases - кортеж з базових класів, nmspc - словник класу"""
       #Виклик конструктора базового класу
       super().__init__(name, bases, nmspc)
        for n, name in enumerate(cls._fields):
            #додати у клас властивість з ім'ям name
            #та методом читання operator.itemgetter(n)
            setattr(cls, name, property(operator.itemgetter(n)))
```

Приклад використання метакласів: класи подібні структурам.3

Клас StructTuple також є нащадком стандартного класу tuple.

```
class StructTuple(tuple, metaclass=StructTupleMeta):
    '''Клас, подібний структурі С.

    Модифікує __new__ для перевірки рівності кількості полів та параметрів
    '''
    _fields = []    #список імен полів

    def __new__(cls, *args):
        if len(args) != len(cls._fields):
            raise ValueError('{} arguments required'.format(len(cls._fields)))
        return tuple.__new__(cls, args) #виклик tuple.__new__ для створення кортежу
```

Приклад використання метакласів: класи подібні структурам.4

Тепер ми можемо породжувати класи-нащадки від StructTuple, вказуючи у їх полі _field список імен полів нашого кортежу.

Наприклад, для точки простору опис класу буде виглядати так:

```
class Point3(StructTuple):

    _fields = ['x','y','z']
```

Цей опис вказано у головному модулі, залишок якого після опису класу Point3 не відрізняється від розглянутого у темі «Кортежі».

Резюме

Ми розглянули:

1. Метакласи. Оголошення метакласу у класі.
2. Абстрактні класи
3. Метапрограмування
4. Декоратори класів
5. Класові методи
6. Побудова класів у динаміці
7. Написання власних метакласів

Де прочитати

1. Обвінцев О.В. Інформатика та програмування. Курс на основі Python. Матеріали лекцій. – К., Основа, 2017
2. Марк Лутц, Изучаем Python, 4-е издание, 2010, Символ-Плюс
3. Python 3.4.3 documentation
4. Марк Саммерфилд, Программирование на Python 3. Подробное руководство. - Символ-Плюс, 2009.
5. Tarek Ziadé. Expert Python Programming. - Packt Publishing, 2008.
6. David Beazley and Brian K. Jones, Python Cookbook. - O'Reilly Media, 2013.
7. http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Metaprogramming.html#basic-metaprogramming
8. http://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/
9. http://habrahabr.ru/post/65625/
10. http://www.python-course.eu/python3_metaclasses.php
11. http://lgiordani.com/blog/2014/10/14/decorators-and-metaclasses/#.VpFyGfmLTDc
12. http://eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example