# Тема 19. Метакласи та метапрограмування

Ми вже розглядали застосування класів та об’єктів, одинарне та множинне наслідування, використання класів у рекурсивних структурах даних та 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 розповсюджовалось 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 плутаніни, 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.

Усі метакласи походять від єдиного стандартного батьківського класу type. Для метакласів існує наслідування, як і для звичайних класів. Але це наслідування тільки одинарне. Тобто, метаклас не може наслідувати від двох або більше метакласів. Таким чином, будь-який метаклас наслідує безпосередньо від type або його нащадків.

Отже, якщо маємо опис

**class** **MyMeta(**type**):**

Q

то MyMeta – це метаклас.

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.

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. Абстрактний метод позначається за допомогою декоратора

*@abstractmethod*

**def** method**():**

**pass**

де method – ім’я методу. Тіло абстрактного методу – це тотожня команда pass.

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 зображеня 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).

**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**

*@property*

*@abstractmethod*

**def** bgcolor**(**self**):**

"""Властивість, що повертає/встановлює колір фону."""

**pass**

*@bgcolor.setter*

*@abstractmethod*

**def** bgcolor**(**self**,** cl**):**

**pass**

*@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**

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**)**

*@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**)**

**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

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

*@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

**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**)**

**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

Для застосування декоратора до класу A треба вказати

@cls\_decorator

**class** **A():**

*Q*

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

Приклад: перевірка, чи є граф деревом, та відслідковування дій над графом

Перевірити, чи є граф деревом. Здійснити відслідковування виконання дій над графом.

Цей приклад був розглянутий у темі «Множинне наслідування». Реалізація тоді базувалась на використанні класів-домішків.

Як і у темі «Множинне наслідування», використаємо клас GraphIt – граф з ітератором – описаний у темі «Ітератори та генератори». У цьому класі також є методи, що повертають напівстепінь входу та виходу вершини графу: hdegin, hdegout.

Зараз для реалізації застосуємо декоратор класу.

Реалізація перевірки, чи є граф деревом, 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\_\_

**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**)**

#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 GraphIt, опишемо його клас-нащадок LoggedGraph з порожньою реалізацією. До класу LoggedGraph застосуємо наш декоратор logged\_mapping.

Модуль, який містить описи класу LoggedGraph та декоратора logged\_mapping, також використовує функції з побудови графу із файлу fileinputgraph та перевірки, чи є граф деревом istree. Ці функції були описані у темах «Рекурсивні структури даних» та «Ітератори та генератори».

@logged\_mapping

**class** **LoggedGraph(**GraphIt**):**

'''Клас, що успадковує від GraphIt.

Застосовується декоратор logged\_mapping

'''

**pass**

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.

Серед вбудованих класових методів частіше використовують метод \_\_new\_\_. Цей метод 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**):**

*P*

Щоб описати власний класовий метод, треба застосовувати декоратор @classmethod

@classmethod

**def** meth**(**cls**,** **...):**

*Q*

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.

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**)**

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. 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\_\_

'''

# 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 стандартного класу tuple

**return** tuple**.**\_\_new\_\_**(**cls**,** args**)**

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

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\_\_. Якщо з \_\_new\_\_ та \_\_init\_\_ ми вже зустрічались, то метод \_\_call\_\_ потребує додаткового пояснення. Цей метод викликається при створенні об’єкта деякого класу, але після \_\_new\_\_ та \_\_init\_\_.

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).

**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**)))**

Клас StructTuple також є нащадком стандартного класу tuple.

**class** **StructTuple(**tuple**,** metaclass**=**StructTupleMeta**):**

'''Клас, подібний структурі C.

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

Тепер ми можемо породжувати класи-нащадки від StructTuple, вказуючі 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. Марк Лутц, Изучаем Python, 4-е издание, 2010, Символ-Плюс
2. Python 3.4.3 documentation
3. Марк Саммерфилд, Программирование на Python 3. Подробное руководство. - Символ-Плюс, 2009.
4. Tarek Ziadé. Expert Python Programming. - Packt Publishing, 2008.
5. David Beazley and Brian K. Jones, Python Cookbook. - O’Reilly Media, 2013.
6. <http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Metaprogramming.html#basic-metaprogramming>
7. <http://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/>
8. <http://habrahabr.ru/post/65625/>
9. <http://www.python-course.eu/python3_metaclasses.php>
10. <http://lgiordani.com/blog/2014/10/14/decorators-and-metaclasses/#.VpFyGfmLTDc>
11. <http://eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example>