

Python

Занятие #6. Классы. Часть 1.

О чем это занятие:

- Зачем нужны классы?
- Синтаксис
 - определение
 - атрибуты
 - методы
 - свойства
- Наследование
 - super
 - mro
- Возможные ошибки

Теги:

классы, ООП, атрибуты, методы класса, абстрагирование, инкапсуляция, наследование, полиморфизм, __init__, self, __dict__, __slots__, super, mro, иерархия наследования

Краткий обзор темы

- ❑ зачем в программировании используются классы
- ❑ базовые принципы ООП: абстрагирование, инкапсуляция, наследование, полиморфизм
- ❑ синтаксис определения класса: конструктор `__init__`, идентификатор `self` – первый в списке параметров
- ❑ атрибуты: публичные, защищенные и приватные
- ❑ атрибуты экземпляра и атрибуты класса
- ❑ использование `__dict__` и `__slots__`
- ❑ методы класса: также бывают связанными, статическими
- ❑ механизм наследования классов
- ❑ функция `super` для определения родителя и алгоритм `mro` для преобразования иерархии наследования в линейный список

Зачем нужны классы?

+ использования

☐ Уменьшение сложности

С помощью [инкапсуляции](#) можно скрыть детали реализации классов и предоставить простой и ясный интерфейс пользователям. Они смогут интегрировать эти классы в свои функции, не разбираясь в их внутреннем устройстве

☐ Переиспользование кода

☐ Определение общего поведения


- использования

☐ Требуется определенной подготовки

[ООП](#) сложнее для понимания, чем процедурное программирование

☐ При неграмотном использовании можно снизить понятность/ясность кода


Принципы объектно-ориентированного программирования (ООП)

 Объектно-ориентированное программирование ([ООП](#)) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Базовые принципы ООП:

1. **Абстрагирование** – набор наиболее значимых характеристик объекта

Это означает отделение того, что будет описано непосредственно в классе, от того, что будет представлено миру.

 Пример: пусть существует объект “зоомагазин”, в котором есть три вида животных: собака, кошка и птица. При этом собака лает, кошка мяукает, а птица чирикает. Но по отношению к внешнему миру все эти действия похожи, поэтому можно их обобщить, введя название свойства “издаёт звук”. То есть мы внесли некоторую неясность для конкретных объектов, но это упростило их взаимодействие с объектами, обладающими существенно отличными свойствами.

2. **Инкапсуляция** – объединение данных и [методов](#)

Во внешнем коде в классе “животное” у объектов есть свойство “издаёт звук”, а уже внутри класса это свойство конкретизируется. Какой объект какой именно звук издаёт, определяется при передаче конкретного объекта класса.

3. **Наследование** – новый класс на основе уже существующего

Если у классов есть некоторая общая функциональность, ее выносят в базовый класс, а исходные классы становятся его наследниками. Внутри наследников реализуется только то, что отличает их от базового класса.

4. **Полиморфизм** – одинаковый код, разные типы

Во внешнем коде следует ориентироваться на общий базовый класс, которые имеет абстрактные методы. При подстановке в этот код конкретной реализации в коде виден базовый класс.

Синтаксис

Синтаксис определения класса

```
class <имя_класса>:  
    def <метод1>  
    def <метод2>  
    ...
```

👉 Первый метод в классе – это
`__init__`

Конструктор `__init__` (инициатор)
вызывается при создании объектов
класса.

💬 Такие методы иногда называются
“дандер”- методы, от англ. double under
– двойное подчеркивание (они же —
магические)

```
1 class TwoNumberCalc:  
2     metric_name = 'two_number_calc'  
3  
4     """Some cool stuff that operates on numbers."""  
5     def __init__(self, first_num: int, second_num: int):  
6         self.first_num = first_num  
7         self.second_num = second_num  
8  
9     def add(self) -> int:  
10         return self.first_num + self.second_num  
11  
12 calc_object = TwoNumberCalc(1, 5)  
13 print('First num: {}'.format(calc_object.first_num))  
14 print('Second num: {}'.format(calc_object.second_num))
```

Пример:

В примере в конструкторе класса принимается два числовых параметра. Описан конструктор `__init__` и функция `add`. После описания класса идентификатору `calc_object` присваивается инстанс класса `TwoNumberCalc` с переданными числами для инициации.


```
15 print('Class name: {}'.format(calc_object.__doc__))
16 print('Adding result: {}'.format(calc_object.add()))
17 print(calc_object.metric_name)
18
19 Out:
    First num: 1
    Second num: 5
    Class name: None
    Adding result: 6
    two_number_calc
```

 Объект класса часто называют **инстанс класса**

Пример: класс Valuer

```
1 class Valuer:
2     """Оценивает качество объявлений"""
3     def __init__(self, verbose=logging.NOTSET):
4         self._logger = logging.getLogger(self.__class__.__name__)
5         self._logger.setLevel(verbose)
6
7     def predict(self, features):
8         features_cnt = len(features)
9         self._logger.debug(f'features count: {features_cnt}')
10
11         str_features = [f for f in features if isinstance(f, str)]
12         self._logger.debug(f'str_features count: {len(str_features)}')
13         return len(str_features) / (features_cnt or 1)
```

В методе `__init__` можно добавить новые атрибуты в объект класса (строка 4?)

 Python позволяет добавлять новые атрибуты объекту класса в любое время.

```
1 Valuer.__doc__
2 Out: 'Оценивает качество объявлений'
3
4 Valuer.__name__
5 Out: 'Valuer'
6
7 Valuer().__class__
8 Out: __main__.Valuer
```

Можно получить docstring, название класса или класс объекта.

 К docstring можно обратиться как через имя класса, так и через имя объекта класса.

Синтаксис: `self`

Все методы класса должны начинаться с указания на объект класса – со слова `self`.

✗ Не смотря на то, что синтаксис языка не запрещает использовать другое название вместо `self`, этого делать не стоит, т.к. существует негласное правило среди разработчиков использовать именно его

`self`:

- обозначает экземпляр класса
- принято называть именно `self`
- явно указывается первым аргументом в методах
- неявно передается при вызове методов

Пример

Сравним идентификаторы объектов

```
1 class Lemmatizer:
2     def _my_id(self):
3         return id(self)
4
5
6 en_lematizer = Lemmatizer('en')
7 assert en_lematizer._my_id() == id(en_lematizer)
8 print(en_lematizer._my_id())
9
```

Out: 4419943280

 <https://docs.python.org/3/library/functions.html#id> – id(object)

Синтаксис: атрибуты

- ❑ атрибуты, не начинающиеся с подчеркивания, называются публичными или открытыми.
- ❑ атрибуты, начинающиеся с одинарного подчеркивания, называются защищенными и используются только внутри класса и его наследников.
- ❑ атрибуты, начинающиеся с двойного подчеркивания, называются [приватными](#).

Пример:

рассматривается класс Adder, объекты которого имеют два публичных атрибута: first_num и second_num

```
1 class Adder:
2     metric_name = 'adder'
3
4     def __init__(self, first_num: int, second_num: int):
5         self.first_num = first_num
6         self.second_num = second_num
7
8 adder = Adder(4, 10) # adder -> | 4 | 10 | 'adder' |
9 another_adder = adder # another_adder -> id=100 | 4 | 10 |
10 'adder' |, adder -> id=100 | 4 | 10 | 'adder' |
11 print('Adder first num: {}'.format(adder.first_num))
```

Здесь два указателя ссылаются на один и тот же объект. Поэтому, когда объект изменяется при обращении через одну ссылку (через another_adder), при обращении по другой (по adder) изменения все равно видны.

```
12 another_adder.first_num = -1000
13 print('Adder first num: {}'.format(adder.first_num))
14
15 Out:
Adder first num: 4
Adder first num: -1000
```

Если применить конструкцию `del another_adder`, то удалится ссылка, но сам объект останется в памяти и через другую ссылку к нему можно будет обратиться.

👉 Можно переопределить атрибут класса на уровне объекта, у класса его значение не изменится, у объекта – будет новым.

Атрибуты нужны для хранения значений:

```
1 def __init__(self, verbose=logging.NOTSET):
2     self._logger = logging.getLogger(self.__class__.__name__)
3     self._logger.setLevel(verbose)
```

Атрибуты экземпляра

- задаются посредством присвоения `self`

```
1 self.lang = lang
2
3 # присвоение атрибуту экземпляра
4 en_lematizer.lang = 'ru'
```

- хранят значения, принадлежащие только рассматриваемому экземпляру

```
1 l1, l2 = Lemmatizer('en'), Lemmatizer('es')
2
3 l1.lang = 'ru'
4 print(l1.lang, l2.lang)
5
Out: ru es
```

Атрибуты класса

- задаются посредством объявления в теле класса

```
1 class Lemmatizer:
2     LANGS = {'en', 'ru'}
3
4     # присвоение напрямую атрибуту класса
5     Lemmatizer.LANGS = {'en', 'ru', 'es'}
```

- хранят значения, доступные всем экземплярам

```
1 l1, l2 = Lemmatizer('en'), Lemmatizer('es')
2
3 Lemmatizer.LANGS = {'en', 'ru', 'es'}
4 print(l1.LANGS, l2.LANGS)
5
Out: {'es', 'en', 'ru'} {'es', 'en', 'ru'}
```

Приватные атрибуты

Вообще говоря, Python позволяет в любой момент обращаться к любому атрибуту независимо от того, двойное или одинарное у него подчёркивание, однако, существует соглашение, что у приватных атрибутов:

- отсутствуют модификаторы доступа
- в начало названия добавляется нижнее подчеркивание

```
1 class Guide:
2     """Формирует советы для улучшения объявления"""
3     recommendations = []
4     _max_recommendation_cnt = 4
```

 <https://stackoverflow.com/a/52903693/2190638> — `__` and `__`

- Для предотвращения случайного доступа к атрибуту используют **два** нижних подчеркивания
- Для доступа нужно указать название класса

```
1 class FastGuide:
2     def guides(self):
3         return ['Add a photo']
4
5     __guides = guides
6 guide = FastGuide()
7 guide._FastGuide__guides()
8
9 Out: ['Add a photo']
```

✗ Однако, использование приватных методов и атрибутов таким способом является свидетельством плохой реализации класса

👉 Чаще всего приватные методы и атрибуты нужны для операций, которые скрыты внутри класса и не являются частью его публичного интерфейса

obj.__dict__

- атрибуты в Python – это словарь `__dict__` внутри каждого объекта
- служебная функция `vars`, примененная к объекту, также позволяет получить словарь с его атрибутами

```
1 class Guide:
2     def __init__(self, title):
3         self.title = title
4
5 guide = Guide("photo recommendations")
6 guide.top_cnt = 10
7 guide.__dict__
8 Out: {'title': 'photo recommendations', 'top_cnt': 10}
9
10 vars(guide) is guide.__dict__
    Out: True
```

- при изменении словаря `__dict__` меняются атрибуты объекта

```
1 guide.point_thr
2
3 Out:
4 -----
5 AttributeError: 'Guide' object has no attribute 'point_thr'
6
7 guide.__dict__['point_thr'] = 0.2
8 guide.point_thr
9
10 Out: 0.2
```

👉 Атрибуты объекта содержатся в `__dict__`, а атрибуты класса – нет.

`obj.__slots__`

Если необходимо, чтобы множество атрибутов класса было фиксированным, используется специальный атрибут `__slots__`, который и хранит неизменяемый набор атрибутов.

💬 Например, это используется для уменьшения количества памяти, занимаемой словарями атрибутов объектов одного класса, т.е. объекты с `__slots__` занимают меньше памяти, так как словари `__dict__` не создаются для каждого объекта

👉 после определения `__slots__` атрибут `__dict__` становится недоступен

```

1 class Guide:
2     __slots__ = ('title',)
3
4 guide = Guide("photo recommendations")
5 guide.point_thr = 10
6
7 Out: AttributeError: 'Guide' object has no attribute 'point_thr'
8
9
10 guide.__dict__
    Out: AttributeError: 'Guide' object has no attribute '__dict__'

```

При использовании `__slots__` получаем более быстрый доступ к атрибутам:

```

1 $ python -m memory_profiler mem_prof.py
2
3 Line #      Mem usage  Increment  Line Contents
4 =====
5 13    48.1 MiB   48.1 MiB   @profile
6 14                                def mem_usage():
7 15    102.5 MiB    3.7 MiB   s = [Gs('s') for _ in range(int(1e6))]
8 16    267.3 MiB    6.6 MiB   d = [Gd('d') for _ in range(int(1e6))]

```

В примере Gs – объекты с использованием `__slots__`, а Gd – без этого атрибута. Объем занимаемой памяти для массива из миллиона элементов отличается более чем в два раза (первый столбец таблицы).

cls.__dict__

Атрибут класса имеет те же параметры, что и атрибут экземпляра: может быть публичным, защищенным или приватным.

Атрибут класса `__dict__` содержит все атрибуты класса и методы класса:

```
1 class Figure:
2     """A representation of geometric figure."""
3     metric_name = 'figure'
4     name = 'figure'
5     square = ''
6     color = ''
7
8     def paint(self):
9         return 'Paint {} of square {} with color "{}".format(self.name, self.square, self.color)
10
11 print(Figure.__dict__)
12 Out:
13 {'__module__': '__main__', '__doc__': 'A representation of geometric figure.', 'metric_name':
'figure', 'name': 'figure', 'square': '', 'color': '', 'paint': <function Figure.paint at
0x7f68505e61e0>, '__dict__': <attribute '__dict__' of 'Figure' objects>, '__weakref__': <attribute
'__weakref__' of 'Figure' objects>}
```

Метод, вызванный для экземпляра класса, в действительности работает так: вызывается метод класса, а в качестве входного параметра передается объект:

```
1 print(romb.paint()) # -> Figure.paint(romb)
```

`mappingproxy` создает двойника атрибутов класса, поэтому все изменения исходных значений атрибутов видны, но изменяться они не могут.

```
1 class Guide:
2     """Формирует советы для улучшения объявления"""
3     point_thr = 0.2
4
5 Guide.__dict__
6
7 Out:
mappingproxy({'__doc__': 'Формирует советы для улучшения объявления',
              'point_thr': 0.2,
              '__dict__': <attribute '__dict__' of 'Guide' objects>,
              '__weakref__': <attribute '__weakref__' of 'Guide' objects>})
```

👉 `mappingproxy` защищает атрибуты класса от изменений

```
1 Guide.__dict__['other_point_thr'] = 0.4
2 Out:
3 TypeError: 'mappingproxy' object does not support item assignment
4
5 Guide.__dict__ = {}
6 Out:
AttributeError: attribute '__dict__' of 'type' objects is not writable
```

В примере показано, что при попытке изменить атрибут класса `other_point_thr` выдается ошибка

🔍 <http://bit.ly/2TqVWJJ> — `types.MappingProxyType`

Синтаксис: методы

Методы – это функции, принадлежащие определенному классу

```
1 class Guide:
2     def guides(self):
3         return ['Add a photo']
4
5 Guide.guides
6
Out: <function __main__.Guide.guides(self)>
```

Связанные методы

Первый аргумент связанного метода зафиксирован и равен связанному объекту:

```
1 guide = Guide()
2 guide.guides
3
4 Out:<bound method Guide.guides of <__main__.Guide object at 0x1107ae240>>
5
6 guide.guides.__self__ is guide
7
Out: True
```

Для вызова метода не нужно передавать экземпляр:

```
1 guide.guides()  
2  
Out: ['Add a photo']
```

Для вызова функции из класса нужно явно передавать экземпляр:

```
1 Guide.guides(guide)  
2  
Out: ['Add a photo']
```

Статические методы

- в статическом методе не нужно указывать `self` первым аргументом
- такой метод одинаково вызывается как у класса так и у объекта

```
1 class Guide:  
2     @staticmethod  
3     def find_bad_words(sentence: str) -> List[str]:  
4         return [w for w in sentence.split() if w.startswith('a')]  
5  
6 Guide.find_bad_words('aaaa bbbb aaabbb')  
7 Out: ['a', 'ab']  
8  
9 guide = Guide()  
10 guide.find_bad_words('a b ab')  
Out: ['a', 'ab']
```

Это может быть нужно, когда некоторый метод связан с классом, но нет необходимости делать его функцией.

👉 Если внутри метода используется класс, то этот метод нужно описывать как метод класса, если же класс не используется, метод описывается как статический.

Методы класса

- Первым аргументом указывается ссылка на класс `cls`
- Используется как альтернативный конструктор

```
1 class Guide:
2     @classmethod
3     def from_tips(cls, tips):
4         guide = cls()
5         for tip in tips:
6             guide.tips.append(tip)
7
8         return guide
9
10 Guide.from_tips(['photo', 'text']).tips
Out: ['photo', 'text']
```

Свойства

`@property` позволяет создавать атрибуты с некоторой логикой, например, атрибуты, вычисляющие значение во время обращения.

```

1 class Guide:
2     def __init__(self, title, trh):
3         self.title = title
4         self.trh = trh
5
6     @property
7     def name(self):
8         return f'{self.title} [{self.trh}]'
9
10 guide = Guide("check text", 0.2)
11 guide.name
12
13 Out: 'check text [0.2]'

```

В примере у класса есть два атрибута *title* и *trh* и одно свойство *name*, при этом свойство *name* используется только для чтения.

С помощью свойств можно изменить поведение, не меняя интерфейс:

```

1 class Guide:
2     def __init__(self, title, trh):
3         self.title = title
4         self._trh = trh
5
6     @property
7     def trh(self):
8         return self._trh

```

Также можно добавить проверку при изменении атрибута:

```
1 @trh.setter
2 def trh(self, new_trh):
3     assert new_trh >= 0
4     self._trh = new_trh
5
6 guide = Guide("check text", 0.2)
7 guide.trh = -2
8
9 Out: AssertionError: trh less than 0
```

или добавить логику при удалении:

```
1 @trh.deleter
2 def trh(self):
3     self._trh = 0
4
5 del guide.trh
6 guide.trh
7
8 Out: 0
```

Тело класса

Объявление атрибутов и методов не является специальным синтаксисом:

```
1 class Guide:
2     guide1, guide2 = 1, 2
3     for i in range(3):
4         guide1 += i
5
6 Guide.__dict__
7
8 Out:
mappingproxy({'guide1': 4, 'guide2': 2, 'i': 2})
```

 <http://bit.ly/2Tug5ym> – statements in the body of a class

Наследование

Синтаксис

Новый класс создается на основе уже описанного, при этом класс перенимает все атрибуты и методы у родительского класса, кроме приватных. В созданном классе есть возможность переопределять или изменять методы родительского класса. Родительский класс указывается в скобках после названия класса.

Пример:

Класс TextGuide наследует класс Guide

```
1 class Guide:
2     _trh = 0.2
3
4     def get_trh(self):
5         raise NotImplementedError
6
7 class TextGuide(Guide):
8     def get_trh(self):
9         return self._trh
10
11 TextGuide().get_trh()
Out: 0.2
```

Поиск атрибутов и методов

Атрибуты и методы объекта класса при попытке обратиться к ним ищутся в следующем порядке:

- сначала в самом экземпляре
- затем в текущем классе, к которому принадлежит объект
- затем в родительских классах

```

1 class Guide:
2     def __init__(self, trh):
3         self.trh = trh
4
5 class TextGuide(Guide):
6     pass
7
8 t_guide = TextGuide(0.2) # вызывается Guide.__init__
9 t_guide.__dict__
Out: {'trh': 0.2}

```

Переопределение методов

- Позволяет дополнить родительский метод новым функционалом
- Если не вызывается родительский метод, то полностью меняется реализация

```

1 class Guide:
2     def __init__(self, trh):
3         self.trh = trh
4
5 class TextGuide(Guide):
6     def __init__(self, trh, min_words):
7         Guide.__init__(self, trh)
8         self.min_words = min_words
9
10 t_guid = TextGuide(0.2, 40)

```

В примере в классе TextGuide вызывается метод `__init__` родительского класса и переопределяет, дополняя его.

```
12 t_guid.__dict__  
Out: {'min_words': 40, 'trh': 0.2}
```

super

Функция `super` “знает”, какой у текущего класса родитель. Она:

- возвращает прокси-объект, передающий вызов методов в родительский класс
- не требует явно указывать родителя
- не требует передавать `self`

```
1 def __init__(self, trh, min_words):  
2     self.min_words = min_words  
3     super().__init__(trh)
```

Множественное наследование

<pre>1 class Guide: 2 def __init__(self, trh): 3 self.trh = trh</pre>	<pre>class Lemmatizer: def lemmatize(self): print('lemmatize')</pre>
---	--

```

1 class TextGuide(Lemmatizer, Guide):
2     def __init__(self, trh, min_words):
3         super().__init__(trh)
4         self.min_words = min_words
5
6 t_guide = TextGuide(0.2, 10)
7 t_guide.lemmatize()
8 T_guide.trh
10
Out: lemmatize
0.2

```

mro (method resolution object)

mro — это специальный алгоритм, который из дерева наследования пытается сделать линейный список, при этом родительские классы линейизируются с помощью алгоритма [C3](#).

👉 Посмотреть линейаризованный список классов в порядке наследования можно с помощью `cls.mro()`

💬 Это необходимо делать, если неизвестно, в каком порядке будут вызываться методы

mro сохраняет порядок родителей, указанный в объявлении класса:

1	class A:	class B(A):	class C:	class D(C, B):
2	pass	pass	pass	pass

```

1 D.mro() # D.__mro__
2
Out:
[__main__.D, __main__.C, __main__.B, __main__.A, object]

```

 <http://bit.ly/2Scy7sE> – The Python 2.3 Method Resolution Order

👉 Если `mro` выдает ошибку, это означает, что иерархия наследования не может быть представлена в виде линейного списка.

Ещё раз о super

Если имеет место множественное наследование, то функция `super` делегирует вызов метода следующему классу в `mro`

```

1 class A:
2     def process(self):
3         print('A.process')
4

```

```

class B(A):
    def process(self):
        print('B.process')
        super().process()

```

```

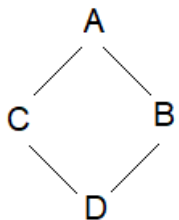
class C(A):
    def process(self):
        print('C.process')
        super().process()

```

```

class D(C, B):
    pass

```



– иллюстрация иерархии наследования

```
1 D().process() # [__main__.D, __main__.C, __main__.B, __main__.A, object]
Out: C.process B.process A.process
```

Предостережения

- ✗ Множественное наследование усложняет понимание кода
- ✗ Результат линейаризации не всегда тривиален, поэтому использовать сложные иерархии множественного наследования не рекомендуется
- ✗ Если неудачно реализовать наследование, то `mro` построить не получится

1	<code>class A:</code>	<code>class B(A):</code>
2	<code>pass</code>	<code>pass</code>

```
1 class C(A, B):
2     pass
3
4 Out:
TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
```

Если родительский класс определяет `__slots__`, а наследник нет, то используется `__dict__`

```
1 class WithSlots:
2     __slots__ = ('a', 'b')
3
4 class WithDict(WithSlots):
5     pass
6
7 class AlsoWithSlots(WithSlots):
8     __slots__ = ()
```

Проверка принадлежности

- `isinstance(obj, clsinfo)` – проверяет, что `obj` является экземпляром класса или кортежа классов
- `issubclass(cls, clsinfo)` – проверяет, что `cls` является потомком класса или кортежа классов

```
1 isinstance('obj', str)
2 Out: True
3
4 issubclass(KeyError, (LookupError, Exception))
5 Out: True
```