



# Классы, объекты

описываем фабрику для своих объектов



# Pass - заглушка

**pass** – оператор-заглушка, равноценный отсутствию операции.

- В ходе исполнения **pass** ничего не происходит, поэтому он может использоваться в качестве заглушки, где это необходимо, например: в инструкциях, где тело является обязательным, таких как **def**, **except**, **class** и т.д.
- Используется там, где пока нет кода, но планируется

```
1  try:
2      some_func()
3  except AttributeError:
4      pass # Описать запись в лог файл.
5
6
7  class MyException(Exception):
8      pass # Добавить DocString.
9
10
11 class MySubclass(MyClass):
12     def do_something(self):
13         pass
14         # Подобное «перекрытие» родительского
15         # метода – возможный индикатор
16         # проблем проектирования интерфейса.
17
18
19 with my_context() as my:
20     pass
21     # При таком подходе теряется сам
22     # смысл менеджера контекста.
```

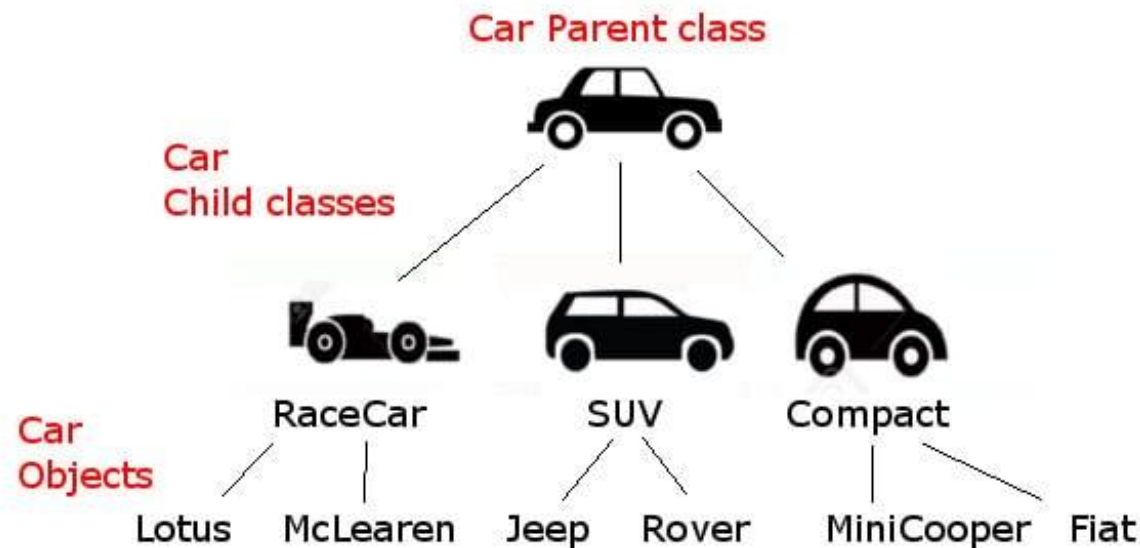


# Что такое объект

**Объект** (экземпляр класса) – некоторая сущность, обладающая определенным состоянием и поведением.

- **Состояние** – определенные свойства (атрибуты)
- **Поведение** – операции над свойствами (методы)

*Объект – некоторый уникальный экземпляр какого-то конкретного предмета.*



# Что такое класс

**Класс** – шаблон для создания объектов. В классе описываются свойства объектов (атрибуты класса) и их поведение (методы класса).

- Для того, чтобы создавать собственные объекты – сперва надо описать класс

```
1 class MyClass:  
2     pass
```



# Как создать свой собственный класс

1. Написать ключевое слово `class`
2. Написать название класса (Классы называем в стиле CamelCase)
3. Не забываем поставить «:»
4. В классе описать его атрибуты
5. В классе описать его методы

```
1 class MyClass:
2     cls_attr = 11
3     obj_attr: str
4
5     def __init__(self, obj_attr):
6         self.obj_attr = obj_attr
7
8     def say_something(self):
9         return f"I say: {self.obj_attr}"
```



# Как создать объект класса

**Инициализация объекта** – создание экземпляра класса.

Для того, чтобы создать экземпляр класса, нужно присвоить переменной имя класса с круглыми скобками и передаваемыми начальными значениями

```
1  class MyCustomClass:
2      val: str
3
4      def __init__(self, some_val):
5          self.val = some_val
6
7
8  class_obj = MyCustomClass('value')
```



# Что происходит «под капотом»

Когда создается объект класса, то интерпретатор Python:

1. Выполняет поиск определения класса `MyCustomClass`
2. Создает в памяти новый объект
3. Вызывает метод `__init__`, передавая только что созданный объект под именем `self` и другие аргументы
4. Сохраняет значение в атрибуте `val`
5. Возвращает новый объект
6. Прикрепляет к объекту имя переменной `class_obj`

```
1  class MyCustomClass:
2      val: str
3
4      def __init__(self, some_val):
5          self.val = some_val
6
7
8  class_obj = MyCustomClass('value')
```



# Атрибуты

Атрибуты (свойства классов и объектов) делятся на:

- Атрибуты класса
- Атрибуты объекта

Доступ к атрибутам осуществляется через точку. Если у класса или объекта нет заданного атрибута, то будет выброшено исключение `AttributeError`

```
1 class MyClass:
2     cls_attr = 11
3     obj_attr: str
4
5     def __init__(self, obj_attr):
6         self.obj_attr = obj_attr
7
8     def say_something(self):
9         return f"I say: {self.obj_attr}"
```



# Атрибуты класса

Атрибуты класса – атрибуты, которые относятся не к конкретному экземпляру, а ко всему классу целиком.

- Создается с помощью описания переменной в классе и присвоения ей значения
- Общий для всех объектов
- При создании в объекте атрибута с таким же именем – будет создан объект атрибута, который не затрагивает атрибут класса

```
1  class MyClass:
2      class_attr = 123
3
4
5  a = MyClass()
6  print(a.class_attr)  # 123
7
8  MyClass.class_attr = 555
9  print(a.class_attr)  # 555
10
11 a.class_attr = 321
12 print(a.class_attr)  # 321
13
14 b = MyClass()
15 print(b.class_attr)  # 555
```

# Атрибут объекта

Атрибуты объекта – атрибуты, которые относятся к конкретному экземпляру класса.

- Можно описать название атрибута и тип, но не обязательно. Не надо присваивать значение, а то получится атрибут класса
- Как правило, начальное значение присваивается в конструкторе класса `__init__`

```
1 class MyClass:
2     obj_attr: str
3
4     def __init__(self, some_str):
5         self.obj_attr = some_str
6
7
8 cls_obj = MyClass('value')
9 print(cls_obj.obj_attr) # 'value'
```



# Методы

Метод – это функция, которая принадлежит классу или объекту. Методы бывают:

- Методы объекта
- Методы класса
- Статические методы

```
1  class MyClass:
2      state = None
3      summa: int
4
5      def save_sum(self, *args):
6          self.summa = sum(args)
7
8      @classmethod
9      def save_state(cls, state):
10         cls.state = state
11
12     @staticmethod
13     def staticmethod():
14         print('static method called')
```



# self

**self** — это стандартное имя первого аргумента, представляющего текущий объект класса.

- Ничто не мешает изменить это имя и использовать любое другое, но так принято.
- Python использует **self**, чтобы найти аргументы и методы правильного объекта

```
1  class MyClass:
2      state = None
3      summa: int
4
5      def save_sum(self, *args):
6          self.summa = sum(args)
7
8
9  my_obj = MyClass()
```



# Методы объекта класса

**Методы объекта** – влияют непосредственно на объект класса.

- Определяются как обычные функции внутри класса
- Первым аргументом принимают self

```
1  class MyClass:
2      state = None
3      summa: int
4
5      def save_sum(self, *args):
6          self.summa = sum(args)
7
8
9  my_obj = MyClass()
```



# cls

`cls` — это стандартное имя первого аргумента, представляющего текущий класс.

- Ничто не мешает изменить это имя и использовать любое другое, но так принято.
- Python использует `cls`, чтобы найти аргументы и методы правильного класса

```
1  class MyClass:
2      state = None
3      summa: int
4
5      @classmethod
6      def save_state(cls, state):
7          cls.state = state
```

# Методы класса

---

**Методы класса** – влияют на весь класс целиком.

- Для определения метода класса, используется декоратор `@classmethod`
- Первым аргументом принимают `cls`

```
1  class MyClass:
2      state = None
3      summa: int
4
5      @classmethod
6      def save_state(cls, state):
7          cls.state = state
```



# Статические методы класса

**Статические методы** – методы, которые не влияют ни на классы, ни на объекты, но они находятся в классе чисто для удобства, вместо того, чтобы располагаться где-то отдельно. Ключевая особенность: не нужно создавать объект для выполнения метода.

- Для определения статического метода, используется декоратор `@staticmethod`
- Не принимает первым параметром ни `self`, ни `cls`

```
1  class MyClass:  
2      @staticmethod  
3      def staticmethod():  
4          print('static method called')
```





# Магические атрибуты класса

|                             |   |           |
|-----------------------------|---|-----------|
| <code>__doc__</code>        | Строка документации, не наследуется дочерними классами или None                                     | Writable  |
| <code>__name__</code>       | Имя функции или класса  | Writable  |
| <code>__qualname__</code>   | Имя функции или класса, включая вложенность   | Writable  |
| <code>__module__</code>     | Имя модуля, где определена функция или None   | Writable  |
| <code>__defaults__</code>   | Кортеж, который содержит аргументы по умолчанию и их значения или None                              | Writable  |
| <code>__code__</code>       | Содержит скомпилированное тело функции  | Writable  |
| <code>__globals__</code>    | Словарь, который содержит имена из глобальной области видимости                                     | Read-only |
| <code>__dict__</code>       | Словарь с атрибутами  | Writable  |
| <code>__annotation__</code> | Словарь с аннотациями типов для атрибутов, вида атрибут-тип и return-тип для возвращаемого значения | Writable  |
| <code>__kwdefaults__</code> | Словарь, который содержит значения по умолчанию для атрибутов                                       | Writable  |
| <code>__slots__</code>      | Атрибуты, для которых нужно зарезервировать место. В качестве значения строка/кортеж/список         | Writable  |



# Конструкторы и деструкторы

| Метод                               | Использование   | Описание  |
|-------------------------------------|---|---|
| <code>__new__(cls, [...])</code>    | <code>class_obj = SomeClass(num=1)</code>                 | Конструктор класса. Создает объект класса и, как правило, но не всегда, передает его в <code>__init__</code> как <code>self</code>  |
| <code>__init__(self, [...])</code>  | <code>class_obj = SomeClass(num=1)</code>                 | Инициализатор класса. Получает аргументы, присваивает значения атрибутам объекта  |
| <code>__init_subclass__(cls)</code> | <pre>MyClass:     pass  NewClass(MyClass):     pass</pre> | Позволяет произвести дополнительную инициализацию класса-наследника. Вызывается всякий раз, когда происходит наследование от класса, в котором данный метод определен. Использование метода упрощает выполнение некоторых типов задач, которые ранее решались при помощи метаклассов. |
| <code>__del__(self)</code>          |   | Деструктор объекта. Если <code>del</code> уменьшает счетчик ссылок на 1, то <code>__del__</code> вызывается, когда счетчик ссылок достигнет 0, либо по завершении работы интерпретатора   |



# Проверка типа

| Метод  | Использование                            | Описание  |
|--|--|---|
| <code>__instancecheck__(self, instance)</code> | <code>isinstance(object, class)</code>   | Проверяет, является ли экземпляр членом вашего класса |
| <code>__subclasscheck__(self, subclass)</code> | <code>issubclass(subclass, class)</code> | Проверяет, наследуется ли класс от вашего класса      |



# Объект как функция

| Метод                                  | Использование  | Описание   |
|--|--|--|
| <code>__call__(self, [args...])</code> | <code>class_obj = SomeClass(num=1)</code><br><code>class_obj(val=1)</code> | Позволяет объекту быть вызванным как функцию. Принимает произвольное число аргументов; то есть, вы можете определить <code>__call__</code> так же как любую другую функцию |



# Магические методы сравнения

| Метод                            | Использование          | Описание  |
|----------------------------------|------------------------|---|
| <code>__eq__(self, other)</code> | <code>x == y</code>    | Определяет поведение оператора равенства        |
| <code>__ne__(self, other)</code> | <code>x != y</code>    | Определяет поведение оператора неравенства      |
| <code>__lt__(self, other)</code> | <code>x &lt; y</code>  | Определяет поведение оператора меньше           |
| <code>__le__(self, other)</code> | <code>x &lt;= y</code> | Определяет поведение оператора меньше или равно |
| <code>__gt__(self, other)</code> | <code>x &gt; y</code>  | Определяет поведение оператора больше           |
| <code>__ge__(self, other)</code> | <code>x &gt;= y</code> | Определяет поведение оператора больше или равно |



# Унарные операторы и функции

| Метод                                   | Использование                   | Описание  |
|---|---------------------------------|---|
| <code>__pos__(self)</code>              | <code>+object</code>            | Определяет поведение унарного плюса   |
| <code>__neg__(self)</code>              | <code>-object</code>            | Определяет поведение унарного минуса  |
| <code>__abs__(self)</code>              | <code>abs(object)</code>        | Определяет поведение функции <code>abs()</code>                                 |
| <code>__invert__(self)</code>           | <code>~object</code>            | Определяет поведение бинарного комплементарного оператора                       |
| <code>__round__(self[, ndigits])</code> | <code>round(object)</code>      | Определяет поведение встроенной функции <code>round()</code>                    |
| <code>__trunc__(self)</code>            | <code>math.trunc(object)</code> | Определяет поведение для метода <code>math.trunc()</code> для обрезания целого  |
| <code>__floor__(self)</code>            | <code>math.floor(object)</code> | Определяет поведение для метода <code>math.floor()</code> для округления «вниз» |
| <code>__ceil__(self)</code>             | <code>math.ceil(object)</code>  | Определяет поведение для метода <code>math.ceil()</code> для округления «вверх» |



# Магические методы арифметических операций

| Метод                                       | Использование                      | Описание  |
|---|------------------------------------|---|
| <code>__add__(self, other)</code>           | <code>object + other</code>        | Оператор сложения   |
| <code>__sub__(self, other)</code>           | <code>object - other</code>        | Оператор вычитания  |
| <code>__mul__(self, other)</code>           | <code>object * other</code>        | Оператор умножения  |
| <code>__truediv__(self, other)</code>       |                                    | Правильное деление. Работает только когда используется <code>from __future__ import division</code> |
| <code>__floordiv__(self, other)</code>      | <code>object // other</code>       | Определяет поведение оператора целочисленного деления   |
| <code>__mod__(self, other)</code>           | <code>object % other</code>        | Определяет поведение оператора остатка от деления   |
| <code>__divmod__(self, other)</code>        | <code>divmod(object, other)</code> | Определяет поведение для встроенной функции <code>divmod()</code>                                   |
| <code>__pow__(self, other[, modulo])</code> | <code>object ** other</code>       | Оператор возведения в степень   |
| <code>__lshift__(self, other)</code>        | <code>object &lt;&lt; other</code> | Оператор двоичного сдвига влево   |
| <code>__rshift__(self, other)</code>        | <code>object &gt;&gt; other</code> | Оператор двоичного сдвига вправо  |



# Магические методы логических операторов

| Метод                              | Использование                   | Описание                             |
|------------------------------------|---------------------------------|--------------------------------------|
| <code>__and__(self, other)</code>  | <code>object &amp; other</code> | Оператор логического И               |
| <code>__or__(self, other)</code>   | <code>object   other</code>     | Оператор логического ИЛИ             |
| <code>__xor__(self, other)</code>  | <code>object ^ other</code>     | Бинарный оператор исключающее ИЛИ    |
| <code>__rand__(self, other)</code> | <code>other &amp; object</code> | Отраженное логическое И              |
| <code>__ror__(self, other)</code>  | <code>other   object</code>     | Отраженное логическое ИЛИ            |
| <code>__rxor__(self, other)</code> | <code>other ^ object</code>     | Отраженного бинарное исключающее ИЛИ |





# Отраженные методы арифметических операций

| Метод  | Использование                      | Описание   |
|--|------------------------------------|--|
| <code>__radd__(self, other)</code>           | <code>object + other</code>        | Отраженное сложение  |
| <code>__rsub__(self, other)</code>           | <code>object - other</code>        | Отраженное вычитание   |
| <code>__rmul__(self, other)</code>           | <code>object * other</code>        | Отраженное умножение   |
| <code>__rtruediv__(self, other)</code>       |                                    | Отражённое правильное деление. Работает только когда используется <code>from __future__ import division</code> . |
| <code>__rfloordiv__(self, other)</code>      | <code>object // other</code>       | Отражённое целочисленное деление   |
| <code>__rmod__(self, other)</code>           | <code>object % other</code>        | Отражённый остаток от деления  |
| <code>__rdivmod__(self, other)</code>        | <code>divmod(other, self)</code>   | Определяет поведение для встроенной функции <code>divmod()</code>  |
| <code>__rpow__(self, other[, modulo])</code> | <code>object ** other</code>       | Отражённое возведение в степерь  |
| <code>__rlshift__(self, other)</code>        | <code>object &lt;&lt; other</code> | Отражённый двоичный сдвиг влево  |
| <code>__rrshift__(self, other)</code>        | <code>object &gt;&gt; other</code> | Отражённый двоичный сдвиг вправо   |



# Составное присваивание

| Метод  | Использование                       | Описание   |
|--|-------------------------------------|--|
| <code>__iadd__(self, other)</code>           | <code>object += other</code>        | Сложение с присваиванием   |
| <code>__isub__(self, other)</code>           | <code>object -= other</code>        | Вычитание с присваиванием  |
| <code>__imul__(self, other)</code>           | <code>object *= other</code>        | Умножение с присваиванием  |
| <code>__imatmul__(self, other)</code>        | <code>object @= other</code>        |  |
| <code>__itruediv__(self, other)</code>       |                                     | Правильное деление с присваиванием. Работает только если используется <code>from __future__ import division</code> . |
| <code>__ifloordiv__(self, other)</code>      | <code>object //= other</code>       | Целочисленное деление с присваиванием  |
| <code>__idiv__(self, other)</code>           | <code>object /= other</code>        | Деление с присваиванием  |
| <code>__imod__(self, other)</code>           | <code>object %= other</code>        | Остаток от деления с присваиванием   |
| <code>__ipow__(self, other[, modulo])</code> | <code>object **= other</code>       | Возведение в степень с присваиванием   |
| <code>__ilshift__(self, other)</code>        | <code>object &lt;=&lt; other</code> | Двоичный сдвиг влево с присваиванием   |
| <code>__irshift__(self, other)</code>        | <code>object &gt;=&gt; other</code> | Двоичный сдвиг вправо с присваиванием  |
| <code>__iand__(self, other)</code>           | <code>object &amp;= other</code>    | Двоичное И с присваиванием   |
| <code>__ixor__(self, other)</code>           | <code>object ^= other</code>        | Двоичный xor с присваиванием   |
| <code>__ior__(self, other)</code>            | <code>object  = other</code>        | Двоичное ИЛИ с присваиванием   |



# Контейнеры

| Метод                                      | Использование   | Описание   |
|--|---|--|
| <code>__len__(self)</code>                 | <code>len(object)</code>                                  | Возвращает количество элементов в контейнере. Часть протоколов для изменяемого и неизменяемого контейнеров.  |
| <code>__getitem__(self, key)</code>        | <code>object[key]</code>                                  | Определяет поведение при доступе к элементу. Тоже относится и к протоколу изменяемых и к протоколу неизменяемых контейнеров. Должен выбрасывать соответствующие исключения: <code>TypeError</code> если неправильный тип ключа и <code>KeyError</code> если ключу не соответствует никакого значения   |
| <code>__setitem__(self, key, value)</code> | <code>object[key] = value</code>                          | Определяет поведение при присваивании значения элементу. Часть протокола изменяемого контейнера. Должен выбрасывать соответствующие исключения: <code>TypeError</code> если неправильный тип ключа и <code>KeyError</code> если ключу не соответствует никакого значения   |
| <code>__delitem__(self, key)</code>        | <code>del object[key]</code>                              | Определяет поведение при удалении элемента. Это часть только протокола для изменяемого контейнера. Должен выбрасывать соответствующие исключения: <code>TypeError</code> если неправильный тип ключа и <code>KeyError</code> если ключу не соответствует никакого значения   |
| <code>__iter__(self)</code>                | <code>iter(object)</code><br><code>for i in object</code> | Должен вернуть итератор для контейнера. Итераторы возвращаются в множестве ситуаций, главным образом для встроенной функции <code>iter()</code> и в случае перебора элементов контейнера выражением <code>for x in container:</code> . Итераторы сами по себе объекты и они тоже должны определять метод <code>__iter__</code> , который возвращает <code>self</code>                  |
| <code>__contains__(self, item)</code>      | <code>x in object</code><br><code>x not in object</code>  | Для проверки принадлежности элемента с помощью <code>in</code> и <code>not in</code> . Когда <code>__contains__</code> не определён, Python просто перебирает всю последовательность элемент за элементом и возвращает <code>True</code> если находит нужный   |
| <code>__missing__(self, key)</code>        | <code>object[key]</code>                                  | <code>__missing__</code> используется при наследовании от <code>dict</code> . Определяет поведение для каждого случая, когда пытаются получить элемент по несуществующему ключу (так, например, если у меня есть словарь <code>d</code> и я пишу <code>d["george"]</code> когда <code>"george"</code> не является ключом в словаре, вызывается <code>d.__missing__("george")</code> ). |
| <code>__reversed__(self)</code>            | <code>reversed(object)</code>                             | Вызывается чтобы определить поведения для встроенной функции <code>reversed()</code> . Должен вернуть обратную версию последовательности. Реализуйте метод только если класс упорядоченный, как список или кортеж.   |



# Доступ к атрибутам

| Метод                                       | Использование                    | Описание  |
|---|----------------------------------|---|
| <code>__getattr__(self, name)</code>        | <code>object.name</code>         | Может быть полезным для перехвата и перенаправления частых опечаток, предупреждения об использовании устаревших атрибутов, или возвращать <code>AttributeError</code> , когда это вам нужно. Вызывается только когда пытаются получить доступ к несуществующему атрибуту, поэтому это не очень хорошее решение для инкапсуляции.  |
| <code>__setattr__(self, name, value)</code> | <code>object.name = value</code> | <code>__setattr__</code> решение для инкапсуляции. Позволяет определить поведение для присвоения значения атрибуту, независимо от того существует атрибут или нет. Можно определить любые правила для любых изменений значения атрибутов. Нужно быть осторожным с тем, как использовать <code>__setattr__</code> , чтобы избежать рекурсии  |
| <code>__delattr__</code>                    | <code>del object.name</code>     | Для удаления атрибутов. Здесь требуются те же меры предосторожности, что и в <code>__setattr__</code> чтобы избежать бесконечной рекурсии (вызов <code>del self.name</code> в определении <code>__delattr__</code> вызовет бесконечную рекурсию).   |
| <code>__getattribute__(self, name)</code>   | <code>object.name</code>         | <code>__getattribute__</code> может использоваться только с классами нового типа (в новых версиях Питона все классы нового типа, а в старых версиях вы можете получить такой класс унаследовавшись от <code>object</code> ). Этот метод позволяет вам определить поведение для каждого случая доступа к атрибутам (а не только к несуществующим, как <code>__getattr__(self, name)</code> ). Он страдает от таких же проблем с бесконечной рекурсией, как и его коллеги (на этот раз вы можете вызывать <code>__getattribute__</code> у базового класса, чтобы их предотвратить). Он, так же, главным образом устраняет необходимость в <code>__getattr__</code> , который в случае реализации <code>__getattribute__</code> может быть вызван только явным образом или в случае генерации исключения <code>AttributeError</code> |



# Магические методы

| Метод                         | Использование                      | Описание  |
|-------------------------------|------------------------------------|---|
| <code>__dir__(self)</code>    | <code>dir(object)</code>           | Определяет поведение функции <code>dir()</code> . Должен возвращать пользователю список атрибутов. Обычно, определение <code>__dir__</code> не требуется, но может быть жизненно важно для интерактивного использования вашего класса, если вы переопределили <code>__getattr__</code> или <code>__getattribute__</code> , или каким-либо другим образом динамически создаёте атрибуты. |
| <code>__sizeof__(self)</code> | <code>sys.getsizeof(object)</code> | Определяет поведение функции <code>sys.getsizeof()</code> , вызванной на экземпляре вашего класса. Метод должен вернуть размер вашего объекта в байтах. Он главным образом полезен для классов, определённых в расширениях на C, но всё-равно полезно о нём знать.  |



# Магические методы представления объектов

| Метод                                    | Использование  | Описание  |
|--|--|---|
| <code>__str__(self)</code>               | <code>str(object)</code>                                     | Определяет поведение функции <code>str()</code>   |
| <code>__repr__(self)</code>              | <code>repr(object)</code>                                    | Определяет поведение функции <code>repr()</code> . Главное отличие от <code>str()</code> в целевой аудитории. <code>repr()</code> больше предназначен для машинно-ориентированного вывода (как правило, валидный код на Python), а <code>str()</code> предназначен для чтения людьми. |
| <code>__format__(self, formatstr)</code> | <code>"{}".format(object)</code><br><code>f"{object}"</code> | Определяет поведение, когда экземпляр вашего класса используется в форматировании строк нового стиля  |



# Магические методы преобразования

## ТИПОВ

| Метод                          | Использование                | Описание  |
|--------------------------------|------------------------------|---|
| <code>__int__(self)</code>     | <code>int(object)</code>     | Преобразование типа в int.  |
| <code>__float__(self)</code>   | <code>float(object)</code>   | Преобразование типа в float.  |
| <code>__complex__(self)</code> | <code>complex(object)</code> | Преобразование типа в комплексное число.  |
| <code>__index__(self)</code>   | <code>some[object]</code>    | Преобразование типа к int, когда объект используется в срезах (выражения вида <code>[start:stop:step]</code> ). Если вы определяете свой числовой тип, который может использоваться как индекс списка, вы должны определить <code>__index__</code> .  |
| <code>__hash__(self)</code>    | <code>hash(object)</code>    | Определяет поведение функции <code>hash()</code> . Метод должен возвращать целочисленное значение, которое будет использоваться для быстрого сравнения ключей в словарях. Заметьте, что в таком случае обычно нужно определять и <code>__eq__</code> тоже. Руководствуйтесь следующим правилом: <code>a == b</code> подразумевает <code>hash(a) == hash(b)</code> . |
| <code>__bytes__(self)</code>   | <code>bytes(object)</code>   | Определяет поведение функции <code>bytes()</code>   |
| <code>__bool__(self)</code>    | <code>bool(object)</code>    | Определяет поведение функции <code>bool()</code>  |



# Протокол дескриптора

| Метод  | Использование   | Описание   |
|--|---|--|
| <code>__get__(self, instance, instance_class)</code> | <code>some_val = object</code>  | Определяет поведение при возвращении значения из дескриптора. <code>instance</code> – объект класса. <code>owner</code> – это тип (класс) объекта.   |
| <code>__set__(self, instance, value)</code>          | <code>object = some_val</code>  | Определяет поведение при изменении значения из дескриптора. <code>instance</code> это объект класса. <code>value</code> – это значение для установки в дескриптор.                                   |
| <code>__delete__(self, instance)</code>              |   | Определяет поведение для удаления значения из дескриптора. <code>instance</code> это объект, владеющий дескриптором.   |
| <code>__set_name__(self, owner, name)</code>         | <pre>class MyClass:     field1 = MyDescriptor()     field2 = MyDescriptor()</pre> | Позволяет получить имя атрибута, связанного с данным дескриптором. <code>owner</code> : Класс владельца дескриптора. <code>name</code> : имя атрибута из класса владельца, связанное с дескриптором. |





# Управляем созданием копий

| Метод  | Использование                      | Описание  |
|--|------------------------------------|---|
| <code>__copy__(self)</code>                  | <code>copy.copy(object)</code>     | Определяет поведение <code>copy.copy()</code> для объекта. <code>copy.copy()</code> возвращает поверхностную копию вашего объекта — это означает, что хоть сам объект и создан заново, все его данные ссылаются на данные оригинального объекта. И при изменении данных нового объекта, изменения будут происходить и в оригинальном.   |
| <code>__deepcopy__(self, memodict={})</code> | <code>copy.deepcopy(object)</code> | Определяет поведение <code>copy.deepcopy()</code> для объекта. <code>copy.deepcopy()</code> возвращает глубокую копию вашего объекта — копируются и объект и его данные. <code>memodict</code> это кэш предыдущих скопированных объектов, он предназначен для оптимизации копирования и предотвращения бесконечной рекурсии, когда копируются рекурсивные структуры данных. Когда вы хотите полностью скопировать какой-нибудь конкретный атрибут, вызовите на нём <code>copy.deepcopy()</code> с первым параметром <code>memodict</code> . |



# Когда классы, а когда функции

## Классы

- Нужно иметь некоторое количество отдельных экземпляров с одинаковым поведением (методами), но разным состоянием (атрибутами)
- Нужна поддержка механизма наследования
- Если есть несколько переменных, которые содержат разные значения и передаются как аргументы функций, то ВОЗМОЖНО ИХ СТОИТ ВЫНЕСТИ В

## Функции

- Нужен только один объект
- Функции в модулях.  
Независимо от того, сколько обращений будет к модулю – он будет загружен только один раз
- Используйте простое решение задачи. Словарь, список или кортеж проще, компактнее и быстрее, чем модуль, который в свою очередь проще, чем класс

# Совет от Гвидо ван Россума

«Избегайте усложнения структур данных. Кортежи лучше объектов (можно воспользоваться именованными кортежами (namedtuple из collections)). Предпочитайте простые поля функциям, геттерам и сеттерам. Используйте больше чисел, строк, кортежей, списков, множеств, словарей. Взгляните также на библиотеку collections, особенно на класс deque»





# Ваши вопросы

что необходимо прояснить в рамках  
данного раздела



# Итераторы, менеджер контекста

как работают последовательности «под капотом», работа с менеджером контекста



# Итератор

Итератор – это объект, который способен перебирать элементы контейнерного класса без необходимости пользователю знать реализацию определенного контейнерного класса

Для создания итераторов используется функция:

- `iter(object)`

```
1 some_list = [1, 2, 3]
2
3 iterator = iter(some_list)
4
5 print(next(iterator)) # 1
6 print(next(iterator)) # 2
7 print(next(iterator)) # 3
8 print(next(iterator)) # StopIteration
```



# Протокол итератора

Для того, чтобы создать свой итератор из класса, класс должен реализовывать протокол дескриптора: 2 метода `iter` (`aiter`) и `next` (`anext`).

|                              |  |   |
|------------------------------|--|---|
| <code>__iter__(self)</code>  | <code>iter(object)</code>                              | Должен возвращать объект итератора (либо <code>self</code> )  |
| <code>__next__(self)</code>  | <code>next(object)</code>                              | Должен возвращать результат следующего значения итератора. В случае исчерпания значений должно вернуть <code>StopIteration</code>           |
| <code>__aiter__(self)</code> | <code>async for item in object(*args, **kwargs)</code> | Должен возвращать объект асинхронного итератора   |
| <code>__anext__(self)</code> | <code>async for item in object(*args, **kwargs)</code> | Должен возвращать awaitable результат следующего значения итератора. В случае исчерпания значений должно вернуть <code>StopIteration</code> |



# Пример создания своего итератора

Пример показывает, как реализовать итератор-аналог `itertools.count`

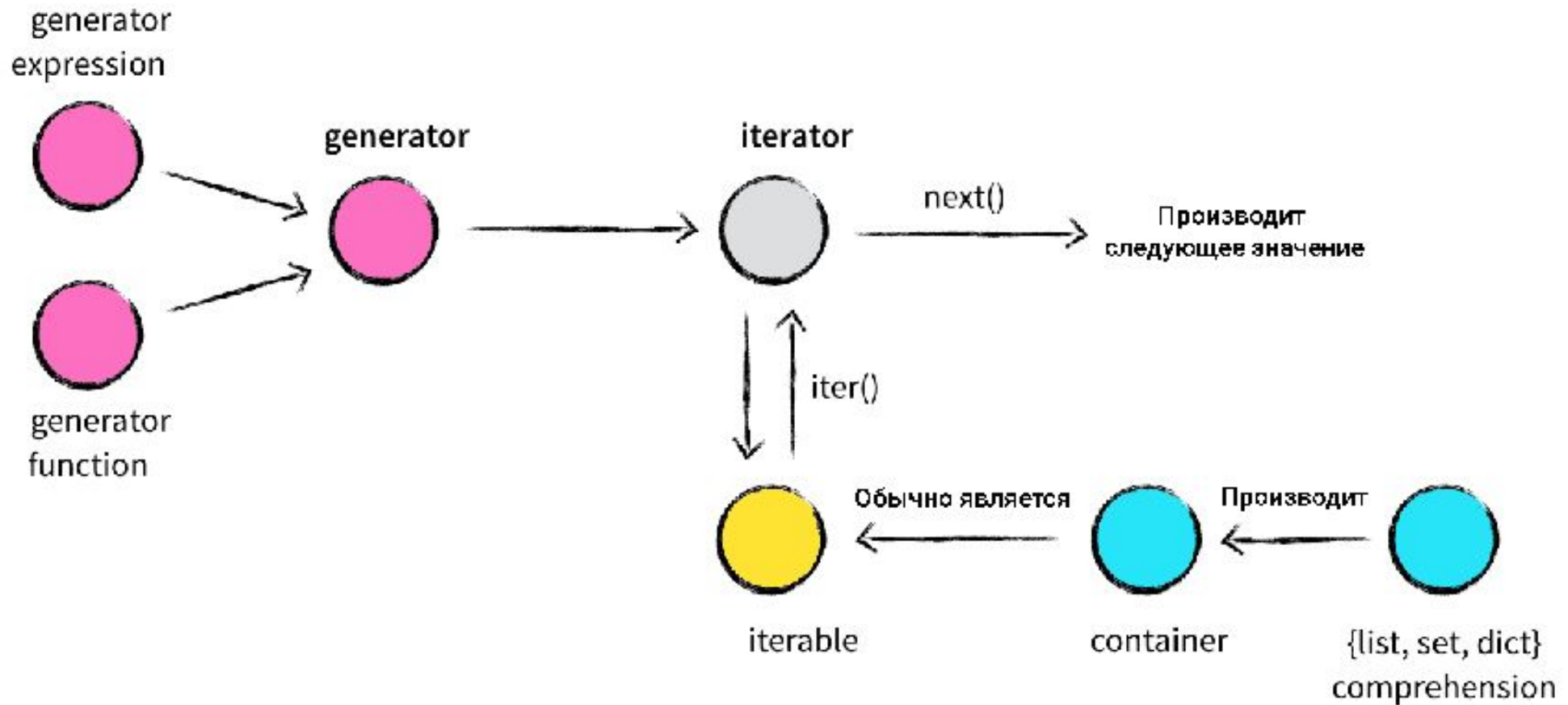
1. Создаем класс
2. Описываем `iter` и `next`
3. Создаем объект класса
4. Итерируемся с помощью `next`

```
1 class Counter:
2     def __init__(self, start=0):
3         self.num = start
4
5     def __iter__(self):
6         return self
7
8     def __next__(self):
9         num = self.num
10        self.num += 1
11        return num
12
13
14 count_iter = Counter()
15
16 print(next(count_iter)) # 0
17 print(next(count_iter)) # 1
18 print(next(count_iter)) # 2
```





# Небольшая шпаргалка



# Менеджер контекста

Конструкция `with ... as` используется для оборачивания выполнения блока инструкций менеджером контекста. Иногда это более удобная конструкция, чем `try...except...finally`.

```
1 class DBConnect:
2     def __init__(self):
3         print('open connection')
4         self.connection = 1
5
6     def __enter__(self):
7         return self.connection
8
9     def __exit__(self, exc_type, exc_val, exc_tb):
10        print('close connection')
11        self.connection = 0
12
13
14 with DBConnect() as connection:
15     print(connection)
16     raise TypeError
```





# Ваши вопросы

что необходимо прояснить в рамках  
данного раздела





# Строки документации

как сделать свой код понятный всем



# Docstring

---

Строки документации - строковые литералы, которые являются первым оператором в модуле, функции, классе или определении метода.

- Становится специальным атрибутом `__doc__` этого объекта
- Все модули должны, как правило, иметь строки документации, и все функции и классы, экспортируемые модулем также должны иметь строки документации. Публичные методы (в том числе `__init__`) также должны иметь строки документации. Пакет модулей может быть документирован в `__init__.py`.



# Создаем Docstring

Для согласованности, всегда используйте для строк документации:

- `"""triple double quotes"""`
- `r"""raw triple double quotes"""`,  
если вы будете использовать обратную косую черту в строке документации

```
1  """
2  Docstring модуля
3  """
4
5
6  class MyClass:
7      """
8      Docstring класса
9      """
10
11     def __init__(self):
12         """Docstring конструктора"""
13         pass
14
15     def set_val(self, val):
16         """Docstring метода"""
17         self.val = val
18
19
20 def sum_numbers(num_1: int, num_2: int) -> int:
21     """Docstring функции
22     """
23     return num_1 + num_2
```



# Docstring функции

Строка документации функции или метода должна:

- Обобщить поведение
- Описать побочные эффекты
- Ограничения на вызов функции
- Описать аргументы
- Описать исключения
- Описать возвращаемые значения

```
1  def sum_numbers(num_1: int, num_2: int) -> int:
2      """Возвращает сумму двух чисел"""
3
4      :param num_1: первое число
5      :type num_1: int
6
7      :param num_2: второе число
8      :type num_2: int
9
10     :return: сумму чисел num_1 и num_2
11     :rtype: int
12
13     :raises TypeError: если num_1 или num_2 не int
14     """
15     return num_1 + num_2
```





# Docstring класса

Строка документации класса должна:

- Обобщить поведение
- Может содержать перечень открытых и закрытых методов
- Может содержать перечень открытых и закрытых атрибутов
- Должно быть указано, если класс предназначен для использования подклассами

```
1 @inspection._self_inspects
2 class InstanceState(interfaces.InspectionAttrInfo):
3     """tracks state information at the instance level.
4
5     The :class:`.InstanceState` is a key object used by the
6     SQLAlchemy ORM in order to track the state of an object;
7     it is created the moment an object is instantiated, typically
8     as a result of :term:`instrumentation` which SQLAlchemy applies
9     to the ``__init__()`` method of the class.
10
11     :class:`.InstanceState` is also a semi-public object,
12     available for runtime inspection as to the state of a
13     mapped instance, including information such as its current
14     status within a particular :class:`.Session` and details
15     about data on individual attributes. The public API
16     in order to acquire a :class:`.InstanceState` object
17     is to use the :func:`_sa.inspect` system::
18
19         >>> from sqlalchemy import inspect
20         >>> insp = inspect(some_mapped_object)
21
22     .. seealso::
23
24         :ref:`core_inspection_toplevel`
25
26     """
```





# Docstring модуля

Строка документации модуля должна:

- Обобщить поведение
- Как правило, перечислять функции, классы, исключения (и любые другие объекты), которые экспортируются модулем, с краткими пояснениями (в одну строчку) каждого из них

```
1  """Defines SQLAlchemy's system of class instrumentation.
2
3  This module is usually not directly visible to user applications, but
4  defines a large part of the ORM's interactivity.
5
6  instrumentation.py deals with registration of end-user classes
7  for state tracking.  It interacts closely with state.py
8  and attributes.py which establish per-instance and per-class-attribute
9  instrumentation, respectively.
10
11 The class instrumentation system can be customized on a per-class
12 or global basis using the :mod:`sqlalchemy.ext.instrumentation`
13 module, which provides the means to build and specify
14 alternate instrumentation forms.
15
16 .. versionchanged: 0.8
17    The instrumentation extension system was moved out of the
18    ORM and into the external :mod:`sqlalchemy.ext.instrumentation`
19    package.  When that package is imported, it installs
20    itself within sqlalchemy.orm so that its more comprehensive
21    resolution mechanics take effect.
22  """
```



# Docstring пакета

Строка документации пакета должна:

- Размещаться в файле «\_\_init\_\_.py»
- Как правило, может включать перечисление импортируемого функционала с кратким (в одну строчку) описанием

```
1 """SQL connections, SQL execution and high-level DB-API interface.
2
3 The engine package defines the basic components used to interface
4 DB-API modules with higher-level statement construction,
5 connection-management, execution and result contexts. The primary
6 "entry point" class into this package is the Engine and its public
7 constructor ``create_engine()``.
8
9 This package includes:
10
11 base.py
12     Defines interface classes and some implementation classes which
13     comprise the basic components used to interface between a DB-API,
14     constructed and plain-text statements, connections, transactions,
15     and results.
16
17 default.py
18     Contains default implementations of some of the components defined
19     in base.py. All current database dialects use the classes in
20     default.py as base classes for their own database-specific
21     implementations.
22
23 strategies.py
24     The mechanics of constructing ``Engine`` objects are represented
25     here. Defines the ``EngineStrategy`` class which represents how
26     to go from arguments specified to the ``create_engine()``
27     function, to a fully constructed ``Engine``, including
28     initialization of connection pooling, dialects, and specific
29     subclasses of ``Engine``.
30
31 threadLocal.py
32     The ``TLEngine`` class is defined here, which is a subclass of
33     the generic ``Engine`` and tracks ``Connection`` and
34     ``Transaction`` objects against the identity of the current
35     thread. This allows certain programming patterns based around
36     the concept of a "thread-local connection" to be possible.
37     The ``TLEngine`` is created by using the "threadLocal" engine
38     strategy in conjunction with the ``create_engine()`` function.
39
40 url.py
41     Defines the ``URL`` class which represents the individual
42     components of a string URL passed to ``create_engine()``. Also
43     defines a basic module-loading strategy for the dialect specifier
44     within a URL.
45 """
```



# Docstring скрипта (самостоятельной программы)

Строки документации самостоятельной программы должна:

- Быть доступны в качестве "сообщения по использованию", напечатанной, когда программа вызывается с некорректными или отсутствующими аргументами (или, возможно, с опцией "-h", для помощи)
- Описать синтаксис командной строки, переменные окружения и файлы.
- Содержать полный справочник со всеми вариантами и аргументами для искушенного пользователя

```
1 import argparse
2
3
4 def parse_cli_args():
5     """
6     Парсинг аргументов командной строки
7     :return: Namespace(port=8000, debug=False, skip=False)
8     """
9     parser = argparse.ArgumentParser(
10         description='Run backend',
11         epilog='Author: https://github.com/EdiBoba'
12     )
13     parser.add_argument(
14         "-h",
15         "--host",
16         help="TCP/IP hostname to serve on (default: %(default)r)",
17         default="localhost",
18     )
19     parser.add_argument(
20         "-p",
21         "--port",
22         type=int,
23         help="TCP/IP port to serve on (default: %(default)r)",
24         default=8000
25     )
26     parser.add_argument(
27         "-d",
28         "--debug",
29         help='activate debug mode (default: %(default)r)',
30         action='store_true',
31         default=False
32     )
33     parser.add_argument(
34         "-s",
35         "--skip",
36         help='skip old messages (default: %(default)r)',
37         action='store_true',
38         default=False
39     )
40     parser.add_argument(
41         "-v",
42         "--version",
43         help='show app version and exit',
44         action='version',
45         version='v1'
46     )
47     arguments = parser.parse_args()
48     return arguments
```

# Стили оформления Docstring

| Тип форматирования     | Описание  | Поддержка Sphinx |
|------------------------|---|------------------|
| Google docstring       | Рекомендации Google по оформлению документации кода   | Да               |
| reStructuredText (RST) | Официальный стандарт Python по оформлению документации кода. Не дружелюбный к новичкам синтаксис, но богатый функционал | Да               |
| NumPy/SciPy docstring  | NumPy комбинация рекомендаций из Google Docstring и ReStructuredText  | Да               |
| Epytext                | Адаптированная для Python Epydoc. Дружелюбна к java разработчикам   | Не официально    |

