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

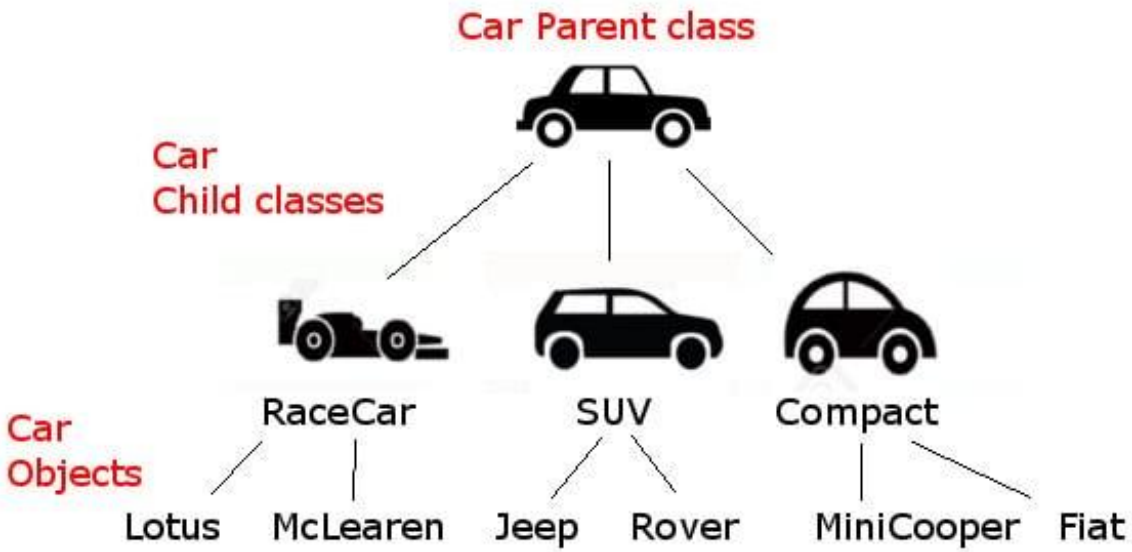
```python
try:
    some_func()
except AttributeError:
    pass  # Описать запись в лог файл.


class MyException(Exception):
    pass  # Добавить DocString.


class MySubclass(MyClass):
    def do_something(self):
        pass
        # Подобное «перекрытие» родительского
        # метода – возможный индикатор
        # проблем проектирования интерфейса.


with my_context() as my:
    pass
    # При таком подходе теряется сам
    # 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 уникальных 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

```python
class MyClass:
    cls_attr = 11
    obj_attr: str

    def __init__(self, obj_attr):
        self.obj_attr = obj_attr

    def say_something(self):
        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

```python
class MyClass:
    cls_attr = 11
    obj_attr: str

    def __init__(self, obj_attr):
        self.obj_attr = obj_attr

    def say_something(self):
        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__

```python
class MyClass:
    obj_attr: str

    def __init__(self, some_str):
        self.obj_attr = some_str


cls_obj = MyClass('value')
print(cls_obj.obj_attr)  # 'value'
```

Методы

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

- Методы объекта
- Методы класса
- Статические методы

```python
class MyClass:
    state = None
    summa: int

    def save_sum(self, *args):
        self.summa = sum(args)

    @classmethod
    def save_state(cls, state):
        cls.state = state

    @staticmethod
    def staticmethod():
        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

```python
class MyClass:
    state = None
    summa: int

    def save_sum(self, *args):
        self.summa = sum(args)


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

```python
class MyClass:
    state = None
    summa: int

    @classmethod
    def save_state(cls, state):
        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

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

Конструкторы и деструкторы

Метод	Использование	Описание
__new__(cls, […)	class_obj = SomeClass(num=1)	Конструктор класса. Создает объект класса и, как правило, но не всегда, передает его в __init__ как self
__init__(self, […)	class_obj = SomeClass(num=1)	Инициализатор класса. Получает аргументы, присваивает значения атрибутам объекта

__init_subclass__(cls)	MyClass:
	  pass
	
	NewClass(MyClass):
	  pass
		Позволяет произвести дополнительную инициализацию класса-наследника. Вызывается всякий раз, когда происходит наследование от класса, в котором данный метод определён. Использование метода упрощает выполнение некоторых типов задач, которые ранее решались при помощи метаклассов.

__del__(self)		Деструктор объекта. Если del уменьшает счетчик ссылок на 1, то __del__ вызывается, когда счетчик ссылок достигнет 0, либо по завершении работы интерпретатора

Проверка типа

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

Объект как функция

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

Магические методы сравнения

Метод	Использование	Описание
__eq__(self, other)	x == y	Определяет поведение оператора равенства
__ne__(self, other)	x != y	Определяет поведение оператора неравенства
__lt__(self, other)	x < y	Определяет поведение оператора меньше
__le__(self, other)	x <= y	Определяет поведение оператора меньше или равно
__gt__(self, other)	x > y	Определяет поведение оператора больше
__ge__(self, other)	x >= y	Определяет поведение оператора больше или равно

Унарные операторы и функции

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

Магические методы арифметических операций

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

Магические методы логических операторов

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

Отраженные методы арифметических операций

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

Составное присваивание

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

Контейнеры

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

Доступ к атрибутам

Метод	Использование	Описание
__getattr__(self, name)	object.name	Может быть полезным для перехвата и перенаправления частых опечаток, предупреждения об использовании устаревших атрибутов, или возвращать AttributeError, когда это вам нужно. Вызывается только когда пытаются получить доступ к несуществующему атрибуту, поэтому это не очень хорошее решение для инкапсуляции.
__setattr__(self, name, value)	object.name = value	__setattr__ 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 __setattr__, чтобы избежать рекурсии
__delattr__	del object.name	Для удаления атрибутов. Здесь требуются те же меры предосторожности, что и в __setattr__ чтобы избежать бесконечной рекурсии (вызов del self.name в определении __delattr__ вызовет бесконечную рекурсию).
__getattribute__(self, name)	object.name	__getattribute__ 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 object). Этот метод позволяет вам определить поведение для каждого случая доступа к атрибутам (а не только к несуществующим, как __getattr__(self, name)). Он страдает от таких же проблем с бесконечной рекурсией, как и его коллеги (на этот раз вы можете вызывать __getattribute__ у базового класса, чтобы их предотвратить). Он, так же, главным образом устраняет необходимость в __getattr__, который в случае реализации __getattribute__ может быть вызван только явным образом или в случае генерации исключения AttributeError

Магические методы

Метод	Использование	Описание
__dir__(self)	dir(object)	Определяет поведение функции dir(). Должен возвращать пользователю список атрибутов. Обычно, определение __dir__ не требуется, но может быть жизненно важно для интерактивного использования вашего класса, если вы переопределили __getattr__ или __getattribute__, или каким-либо другим образом динамически создаёте атрибуты.
__sizeof__(self)	sys.getsizeof(object)	Определяет поведение функции sys.getsizeof(), вызываемой 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
__str__(self)	str(object)	Определяет поведение функции str()
__repr__(self)	repr(object)	Определяет поведение функции repr(). Главное отличие от str() в целевой аудитории. repr() больше предназначен для машинно-ориентированного вывода (как правило, валидный код на Python), а str() предназначен для чтения людьми.
__format__(self, formatstr)	“{}”.format(object) f”{object}”	Определяет поведение, когда экземпляр вашего класса используется в форматировании строк нового стиля

Магические методы преобразования типов

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

Протокол дескриптора

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

Управляем созданием копий

Метод	Использование	Описание
__copy__(self)	copy.copy(object)	Определяет поведение copy.copy() для объекта. copy.copy() 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.
__deepcopy__(self, memodict={})	copy.deepcopy(object)	Определяет поведение copy.deepcopy() для объекта. copy.deepcopy() возвращает глубокую копию вашего объекта — копируются и объект и его данные. memodict 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 copy.deepcopy() с первым параметром memodict.

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 кл

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).

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

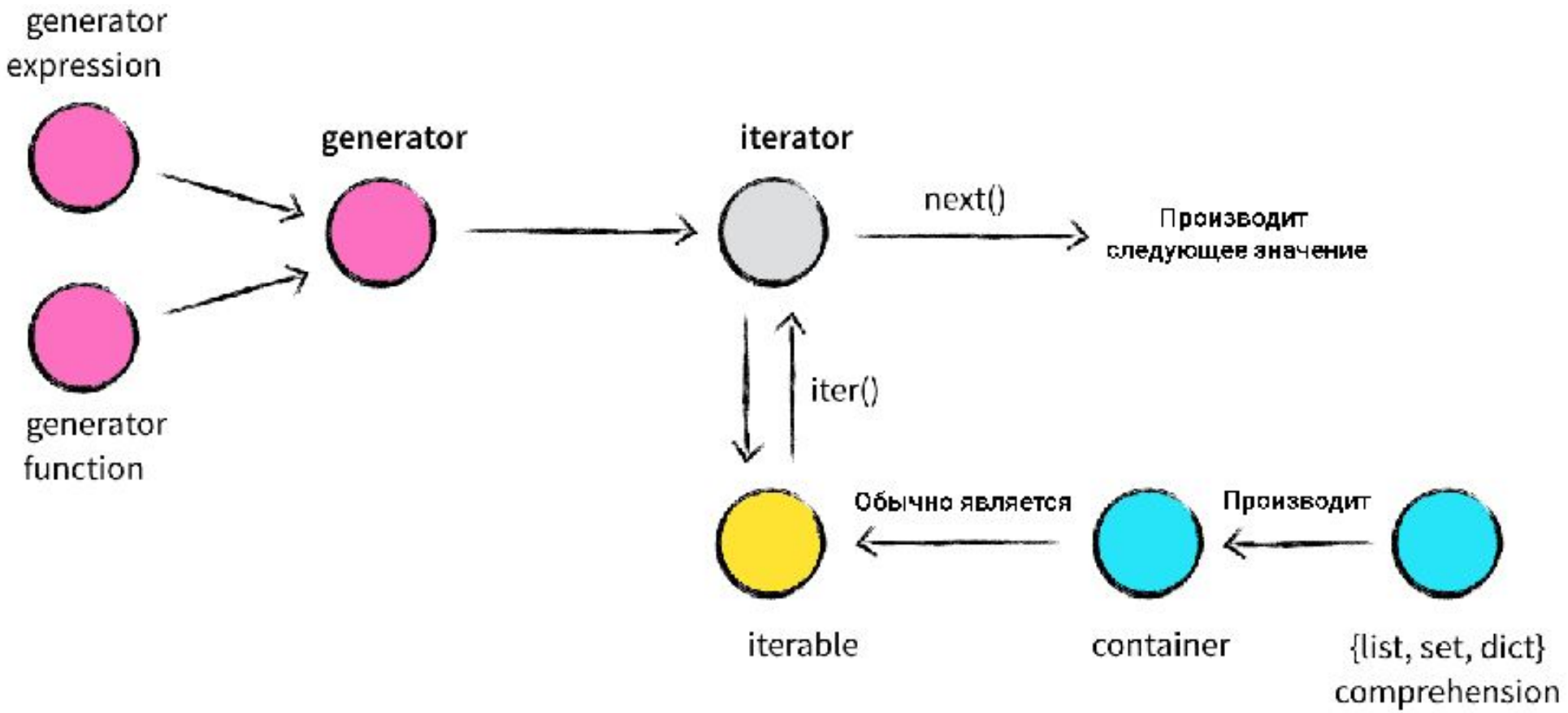
Пример создания своего итератора

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

```python
class Counter:
    def __init__(self, start=0):
        self.num = start

    def __iter__(self):
        return self

    def __next__(self):
        num = self.num
        self.num += 1
        return num


count_iter = Counter()

print(next(count_iter))  # 0
print(next(count_iter))  # 1
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

```python
"""
Docstring модуля
"""


class MyClass:
    """
    Docstring класса
    """

    def __init__(self):
        """Docstring конструктора"""
        pass

    def set_val(self, val):
        """Docstring метода"""
        self.val = val


def sum_numbers(num_1: int, num_2: int) -> int:
    """Docstring функции
    """
    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

```python
def sum_numbers(num_1: int, num_2: int) -> int:
    """Возвращает сумму двух чисел

    :param num_1: первое число
    :type num_1: int

    :param num_2: второе число
    :type num_1: int

    :return: сумму чисел num_1 и num_2
    :rtype: int

    :raises TypeError: если num_1 или num_2 не int
    """
    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

```
@inspection._self_inspects
class InstanceState(interfaces.InspectionAttrInfo):
    """tracks state information at the instance level.

    The :class:`.InstanceState` is a key object used by the
    SQLAlchemy ORM in order to track the state of an object;
    it is created the moment an object is instantiated, typically
    as a result of :term:`instrumentation` which SQLAlchemy applies
    to the ``__init__()`` method of the class.

    :class:`.InstanceState` is also a semi-public object,
    available for runtime inspection as to the state of a
    mapped instance, including information such as its current
    status within a particular :class:`.Session` and details
    about data on individual attributes.  The public API
    in order to acquire a :class:`.InstanceState` object
    is to use the :func:`_sa.inspect` system::

        >>> from sqlalchemy import inspect
        >>> insp = inspect(some_mapped_object)

    .. seealso::

        :ref:`core_inspection_toplevel`

    """
```

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

```
"""Defines SQLAlchemy's system of class instrumentation.

This module is usually not directly visible to user applications, but
defines a large part of the ORM's interactivity.

instrumentation.py deals with registration of end-user classes
for state tracking.   It interacts closely with state.py
and attributes.py which establish per-instance and per-class-attribute
instrumentation, respectively.

The class instrumentation system can be customized on a per-class
or global basis using the :mod:`sqlalchemy.ext.instrumentation`
module, which provides the means to build and specify
alternate instrumentation forms.

.. versionchanged: 0.8
   The instrumentation extension system was moved out of the
   ORM and into the external :mod:`sqlalchemy.ext.instrumentation`
   package.  When that package is imported, it installs
   itself within sqlalchemy.orm so that its more comprehensive
   resolution mechanics take effect.
"""
```

Docstring пакета

Строка документации пакета должна:

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

```
"""SQL connections, SQL execution and high-level DB-API interface.

The engine package defines the basic components used to interface
DB-API modules with higher-level statement construction,
connection-management, execution and result contexts.  The primary
"entry point" class into this package is the Engine and its public
constructor ``create_engine()``.

This package includes:

base.py
    Defines interface classes and some implementation classes which
    comprise the basic components used to interface between a DB-API,
    constructed and plain-text statements, connections, transactions,
    and results.

default.py
    Contains default implementations of some of the components defined
    in base.py.  All current database dialects use the classes in
    default.py as base classes for their own database-specific
    implementations.

strategies.py
    The mechanics of constructing ``Engine`` objects are represented
    here.  Defines the ``EngineStrategy`` class which represents how
    to go from arguments specified to the ``create_engine()``
    function, to a fully constructed ``Engine``, including
    initialization of connection pooling, dialects, and specific
    subclasses of ``Engine``.

threadlocal.py
    The ``TLEngine`` class is defined here, which is a subclass of
    the generic ``Engine`` and tracks ``Connection`` and
    ``Transaction`` objects against the identity of the current
    thread.  This allows certain programming patterns based around
    the concept of a "thread-local connection" to be possible.
    The ``TLEngine`` is created by using the "threadlocal" engine
    strategy in conjunction with the ``create_engine()`` function.

url.py
    Defines the ``URL`` class which represents the individual
    components of a string URL passed to ``create_engine()``.  Also
    defines a basic module-loading strategy for the dialect specifier
    within a URL.
"""
```

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

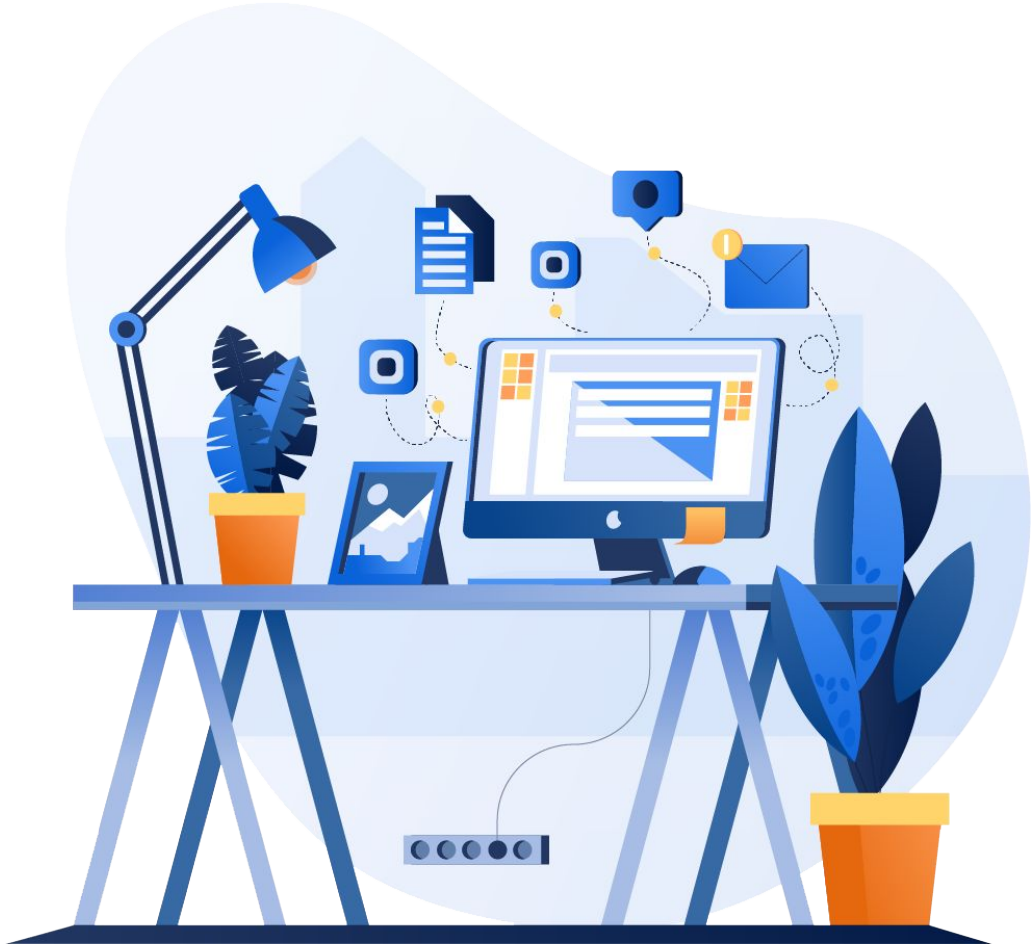
```python
import argparse


def parse_cl_args():
    """
    Парсинг аргументов командной строки
    :return: Namespace(port=8000, debug=False, skip=False)
    """
    parser = argparse.ArgumentParser(
        description='Run backend',
        epilog='Author: https://github.com/EdiBoba'
    )
    parser.add_argument(
        "-H",
        "--host",
        help="TCP/IP hostname to serve on (default: %(default)r)",
        default="localhost",
    )
    parser.add_argument(
        '-p',
        '--port',
        type=int,
        help='TCP/IP port to serve on (default: %(default)r)',
        default=8000
    )
    parser.add_argument(
        '-d',
        '--debug',
        help='activate debug mode (default: %(default)r)',
        action='store_true',
        default=False
    )
    parser.add_argument(
        '-s',
        '--skip',
        help='skip old messages (default: %(default)r)',
        action='store_true',
        default=False
    )
    parser.add_argument(
        '-v',
        '--version',
        help='show app version and exit',
        action='version',
        version='v1'
    )
    arguments = parser.parse_args()
    return arguments
```

Стили оформления Docstring

Тип форматирования	Описание	Поддержка Sphynx
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

Домашнее задание

1. Выполнить задания из репозитория: https://github.com/EdiBoba/belhard_7_tasks
2. Выслать скриншоты успешного прохождения pytest и flake8



Спасибо за внимание!

вопросы во вне учебное время можно задать по контактам с 9:00 до 21:00:

t.me/ediboba

+375(29)339-25-87 (МТС)

rineisky@gmail.com