



ООП в Python

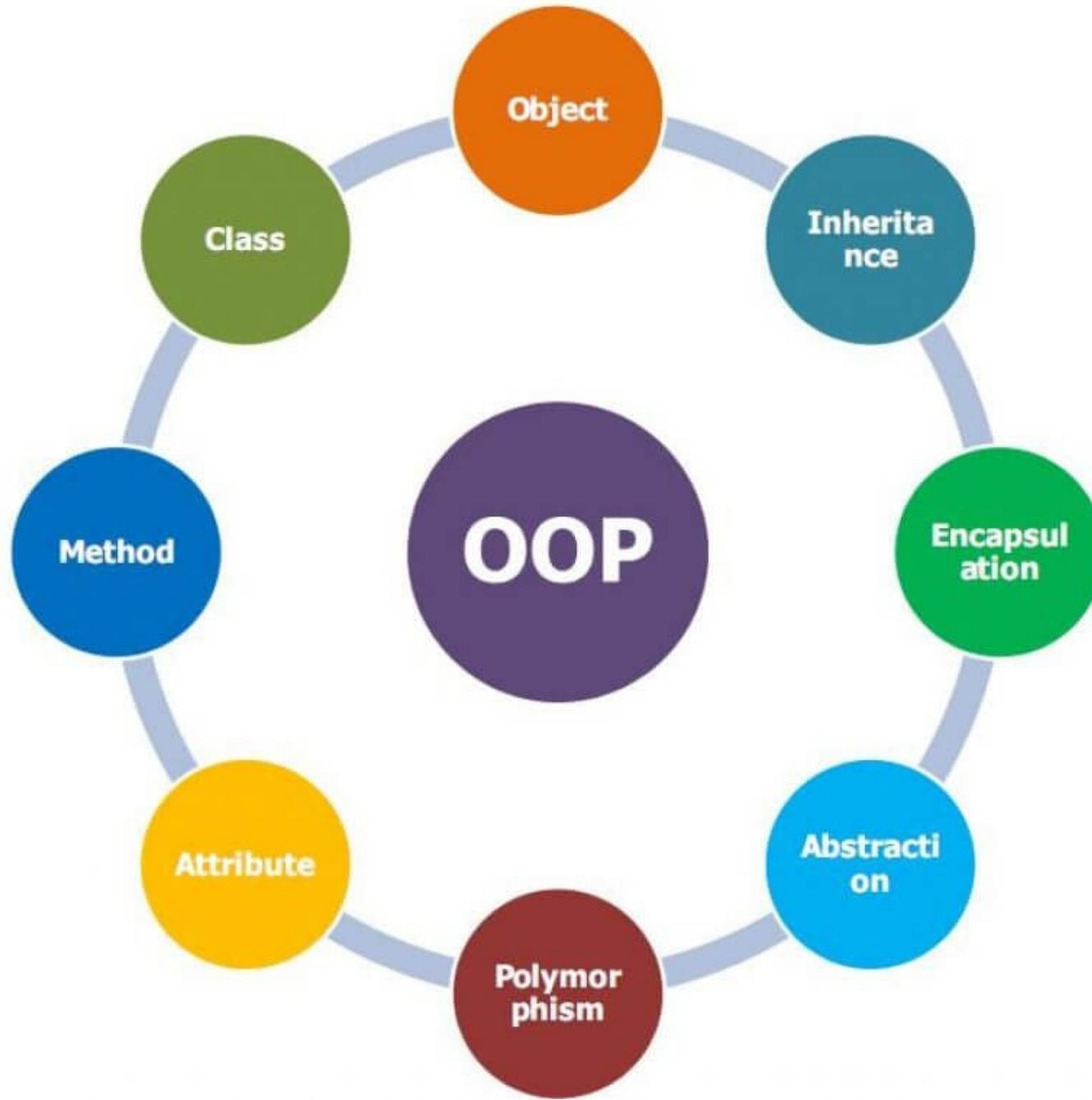
Часть I



O Object
O Oriented
P Programming

Определение ООП

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





Основные принципы ООП

- Инкапсуляция - сокрытие реализации.
- Наследование - создание новой сущности на базе уже существующей.
- Полиморфизм - возможность иметь разные формы для одной и той же сущности.
- Абстракция - набор общих характеристик.
- Посылка сообщений - форма связи, взаимодействия между сущностями.
- Переиспользование - повторное использование кода.

Инкапсуляция

- Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя, открыв только то, что необходимо при последующем использовании.
- Цель инкапсуляции — уйти от зависимости внешнего интерфейса класса (то, что могут использовать другие классы) от реализации. Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса.

Наследование

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется предком, базовым или родительским. Новый класс – потомком, наследником или производным классом.

Полиморфизм

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного и того же интерфейса для задания единого набора действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор языка программирования. Отсюда следует ключевая особенность полиморфизма - использование объекта производного класса, вместо объекта базового (потомки могут изменять родительское поведение, даже если обращение к ним будет производиться по ссылке родительского типа).

Абстракция данных

Абстрагирование означает выделение значимых характеристик и исключение из рассмотрения частных и незначимых. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор наиболее общих характеристик объекта, доступных остальной программе.

Пример:

Представьте, что водитель едет в автомобиле по оживлённому участку движения. Понятно, что в этот момент он не будет задумываться о химическом составе краски автомобиля, особенностях взаимодействия шестерён в коробке передач или влияния формы кузова на скорость . Однако, руль, педали, указатель поворота он будет использовать регулярно.

Посылка сообщений (вызов метода)

Объекты взаимодействуют, посылая и получая сообщения. Сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия. В ООП посылка сообщения (вызов метода) — это единственный путь передать управление объекту. Если объект должен «отвечать» на это сообщение, то у него должна иметься соответствующий данному сообщению метод. Так же объекты, используя свои методы, могут и сами посыпать сообщения другим объектам.

Объект 1

Состояние

Переменная 1

Переменная 2

Интерфейс

Метод 1

Метод 2

Метод 3

Объект 2

Интерфейс

Метод 1

Метод 2

Состояние

Переменная 1

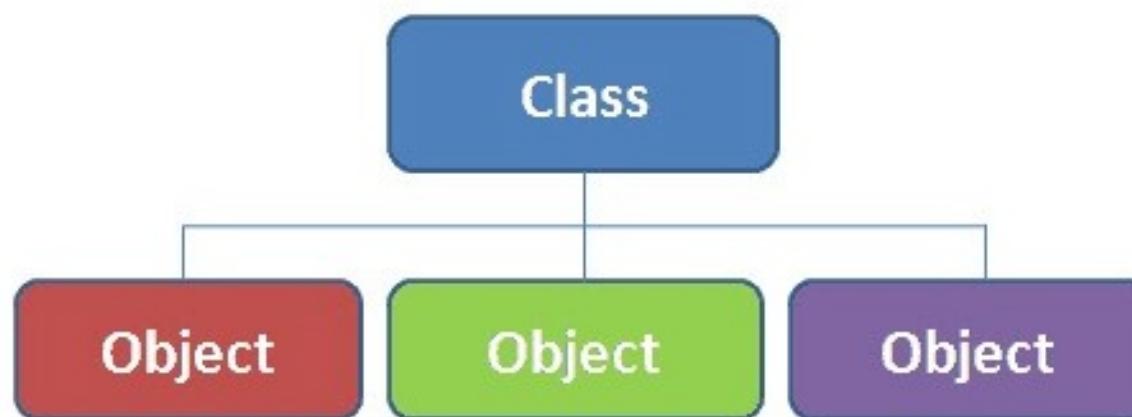
Сообщения





Понятия ООП

- Класс
- Объект
- Интерфейс



Класс

Класс — универсальный, комплексный **тип данных**, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей). В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определённым полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю.

Объект (экземпляр)

Объект - это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом. Каждый объект имеет конкретные значения атрибутов и методы, работающие с этими значениями на основе правил, заданных в классе.

Интерфейс

Интерфейс - это набор методов класса, доступных для использования. Интерфейсом класса будет являться набор всех его публичных методов в совокупности с набором публичных атрибутов. По сути, интерфейс специфицирует класс, чётко определяя все возможные действия над ним.

Класс, объект

Когда речь идёт об объектно-ориентированном программировании в Python, первое, что нужно сказать – это то что любые переменные любых типов данных являются объектами, а типы являются классами.

Каждый объект имеет набор атрибутов, к которым можно получить доступ с помощью оператора “.”. Мы уже имеем опыта работы с атрибутами, пример списков:

```
lst = [1, 2, 3]
```

```
lst.append(4)
```

Атрибуты, которые являются функциями, называются методами.

Создание класса CLASS

Создать класс можно с помощью конструкции **class**

```
# Создание пустого класса Animal  
class Animal:  
    pass
```

Создание объекта класса

Для того чтобы создать объект на основе класса Animal
нужно вызвать его как функцию.

```
animal = Animal()
```

Определение отношения объекта

Для определения, к какому классу относится объект, можно вызвать внутренний атрибут объекта `__class__`. Также можно воспользоваться функцией `isinstance` (рекомендуется этот вариант).

```
# Определить класс объекта
print(isinstance(animal, Animal))

print(animal.__class__)
```

```
# Поля объекта
print(dir(animal))
```

Класс с атрибутами

```
class Animal:  
    color = "red"  
    weight = 200  
  
animal = Animal()  
  
# Поля объекта  
print(dir(animal))  
  
print(animal • color)  
print(animal • weight)
```

Изменение атрибутов

```
class Animal:  
    color = "red"  
    weight = 200  
  
a1 = Animal()  
a2 = Animal()  
  
a1.weight = 400  
print(a1.color, a1.weight)  
a2.color = "blue"  
print(a2.color, a2.weight)
```

Создание новых атрибутов внутри объектов

```
# Добавим новый атрибут speed к объектам a1, a2

class Animal:
    color = "red"
    weight = 200

a1 = Animal()
a2 = Animal()

a1.speed = 300
a2.speed = 400

print(a1.color, a1.weight, a1.speed)
print(a2.color, a2.weight, a2.speed)
```

Доступ к атрибутам класса

Класс Animal так же является объектом и внутри него так же есть атрибуты, как и внутри объектов, созданных с помощью класса Animal.

```
print(Animal.color, Animal.weight)  
Animal.color = "green"  
Animal.weight = 300  
print(a1.color, a1.weight, a1.speed)  
print(a2.color, a2.weight, a2.speed)
```

Как вы могли заметить, в объектах изменились значения только тех атрибутов, которые мы не меняли до этого в этих объектах. Это поведение будет объяснено позже.

Методы класса

Для объявления методов внутри класса нужно просто создать функцию внутри класса. Эта функция должна принимать как минимум один аргумент. По общепринятому соглашению этот аргумент называется **self** и в него передаётся сам объект класса, из которого этот метод и вызывается.

```
class Animal:  
    def set(self, value):  
        self.value = value  
  
    def print(self):  
        print("VALUE:", self.value)  
  
animal = Animal()  
animal.set(20)  
animal.print()  
animal.set("May...")  
animal.print()
```

Конструктор `__init__`

В классах можно описывать так называемые "волшебные" методы, то есть методы, которые имеют заранее прописанный функционал в языке Python. Одним из таких методов является конструктор.

Конструктор – специальный метод, который вызывается сразу же после создания объекта и производит первичные действия. Чтобы создать конструктор в Python – нужно создать функции в классе с именем `__init__`.

```
class Animal:  
    def __init__(self, color, weight):  
        self.color = color  
        self.weight = weight  
  
a1 = Animal("red", 300)      # при создании объекта нужно  
a2 = Animal("green", 200)    # передавать все аргументы, кроме  
                            # self, которые принимаются в  
                            # конструкторе.  
  
a1.color = "blue"  
a2.weight = 500  
print(a1.color, a1.weight)  
print(a2.color, a2.weight)
```

Деструктор `__del__`

Также существует такой "волшебный" метод который называется деструктор. Он вызывается в момент удаления объекта из памяти.

```
class Animal:

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

    def __del__(self):
        print(self.name, "will deleted.")

a1 = Animal("Kent")
a2 = Animal("Afon")
a3 = Animal("Zuk")
del a2 # Удаление переменной
```

Сначала на экран вывелаась строка об удалении Afon, а потом об удалении остальных объектов. Несмотря на то, что мы явно их не удаляем, они всё равно удаляются при достижении программой конца.

```
class Lection:

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

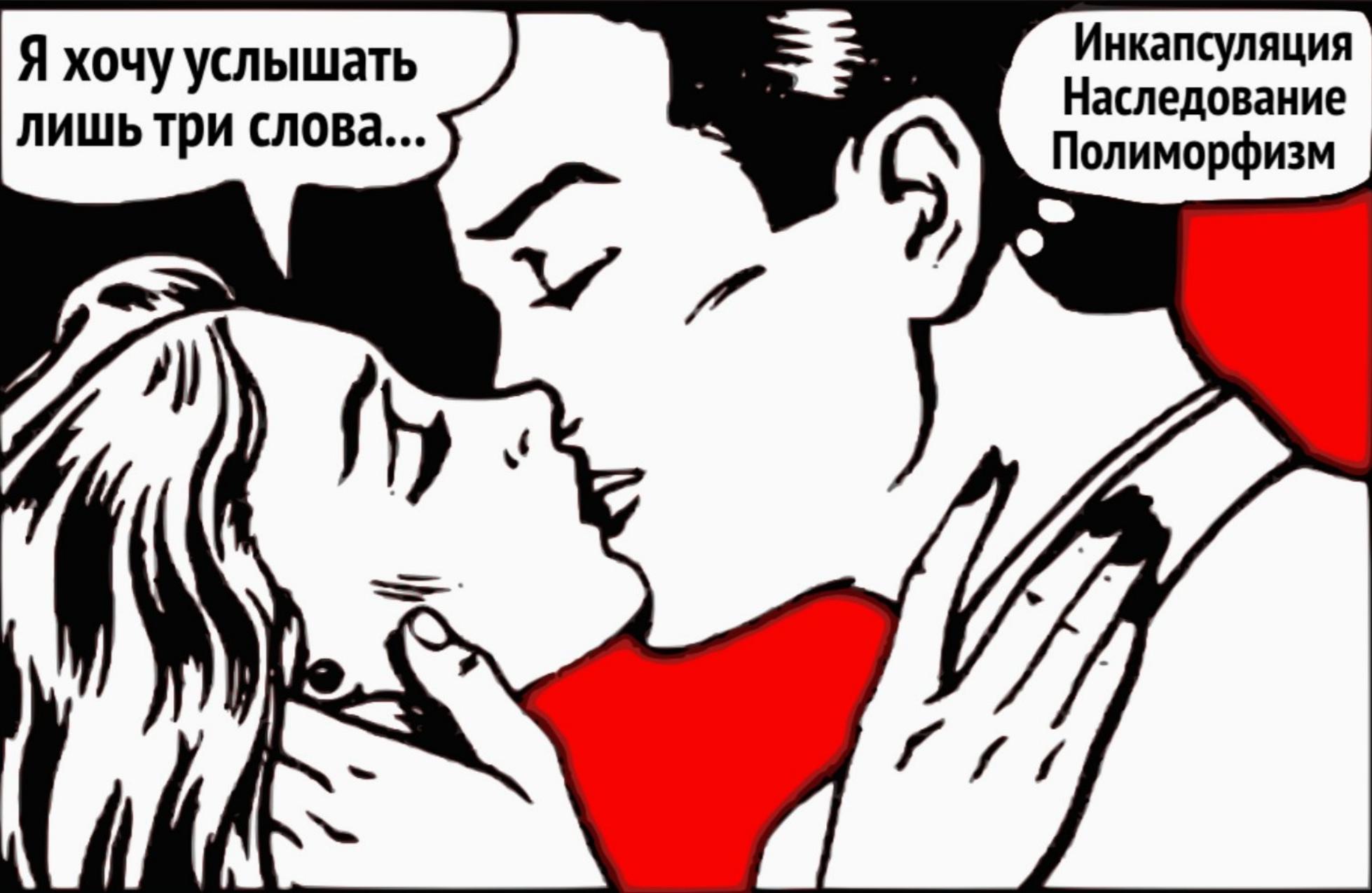
    def __del__(self):
        print(self.name, "Продолжение следует.")

obj = lection("lection_17")
del obj
```

A decorative graphic in the top-left corner consists of a series of semi-transparent blue squares of varying sizes and shades, arranged in a staggered, overlapping fashion that tapers towards the top right.

ООП

Часть II



Я хочу услышать
лишь три слова...

Инкапсуляция
Наследование
Полиморфизм

Инакапсуляция

По умолчанию атрибуты в классах являются общедоступными (public), а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его.

```
class Person:  
    def __init__(self, name):  
        self.name = name      # устанавливаем имя  
        self.age = 1          # устанавливаем возраст  
  
    def display_info(self):  
        print(f"Имя: {self.name}\tВозраст:  
{self.age}")  
  
obj = Person("Pupkin")  
tom.age = 50                  # изменяем атрибут age  
tom.display_info()            # Имя:Pupkin  
                            # Возраст: 50
```

Инкапсуляция

Все объекты в Python инкапсулируют внутри себя данные и методы работы с ними, предоставляя публичные интерфейсы для взаимодействия.

Атрибут может быть объявлен **приватным** (private) с помощью **нижнего подчеркивания** перед именем, но настоящего скрытия на самом деле не происходит – все на уровне соглашений.

```
class SomeClass:  
    def __private(self):  
        print("Это внутренний метод объекта")  
  
obj = SomeClass()  
obj.__private() # Это внутренний метод объекта
```

Два нижних подчеркивани (double underscore) - дандер

```
# Если поставить перед именем атрибута два
# подчеркивания, к нему нельзя будет обратиться
# напрямую.

class SomeClass():

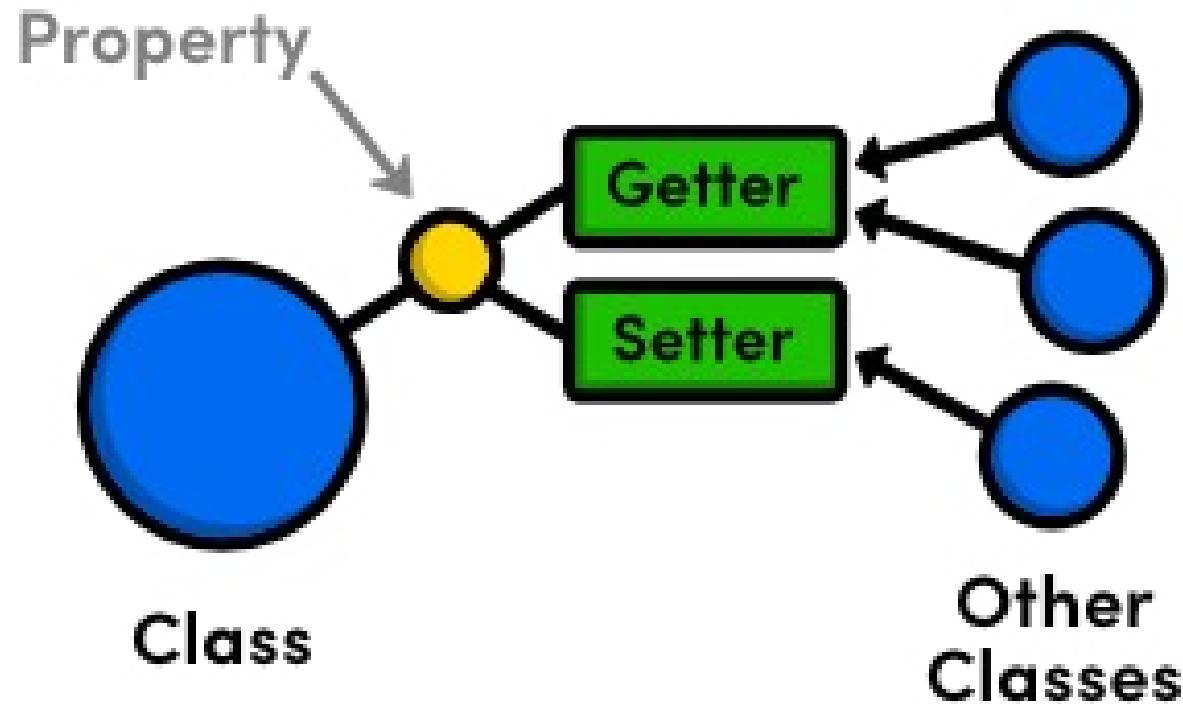
    def __init__(self):
        self.__param = 42 # приватный атрибут

obj = SomeClass()

obj.__param # AttributeError: 'SomeClass' object has
no attribute '__param'

obj._SomeClass__param # – обходной способ
```

Методы доступа к свойствам



```
class Person:
    def __init__(self, name):
        self.__name = name # устанавливаем имя
        self.__age = 1 # устанавливаем возраст

    def set_age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

    def get_age(self):
        return self.__age

    def get_name(self):
        return self.__name

    def display_info(self):
        print(f"Имя: {self.__name}\tВозраст: {self.__age}")

tom = Person("Tom")
tom.display_info() # Имя: Том Возраст: 1
tom.set_age(25)
tom.display_info() # Имя: Том Возраст: 25
```

Встроенные методы

Вместо того чтобы вручную создавать геттеры и сеттеры для каждого атрибута, можно перегрузить встроенные методы

- `__getattr__`
- `__setattr__`
- `__delattr__`

Например, так можно перехватить обращение к свойствам и методам, которых в объекте не существует:

Встроенные методы `__getattr__`

```
#    автоматически вызывается при получении
#    несуществующего свойства класса

class SomeClass():
    attr1 = 42

    def __getattr__(self, attr):
        return attr.upper()

obj = SomeClass()
obj.attr1 # 42
obj.attr2 # ATTR2
```

__getattribute__

Перехватывает все обращения (в том числе и к существующим атрибутам) :

```
class SomeClass():

    attr1 = 42

    def __getattribute__(self, attr):
        return attr.upper()

obj = SomeClass()

obj.attr1 # ATTR1
obj.attr2 # ATTR2
```

Встроенные методы `__setattr__`

#автоматически вызывается при изменении свойства класса;

```
class SomeClass():
```

```
    age = 42
```

```
    def __setattr__(self, attr, value):
```

```
        if attr == 'age':
```

```
            print( 'Age, {} !'.format( value ) )
```

```
            self.age = value
```

```
obj = SomeClass()
```

```
obj.age # 45
```

```
obj.age = 100 # Вызовет метод __setattr__
```

```
obj.name = 'Pupkin'      ← Что произойдет при вызове ?
```

__delattr__ удаление атрибута

```
class Car:

    def __init__(self):
        self.speed = 100

    def __delattr__(self, attr):
        self.speed = 42

# Создаем объект
porsche = Car()
print(porsche.speed)
# 100

delattr(porsche, 'speed') ← Удаление атрибута у объекта
print(porsche.speed) → 42
```

Перехват обращение к свойствам

Например, так можно перехватить обращение к свойствам и методам, которых в объекте не существует:

```
class AccessControl:  
    def __setattr__(self, attr, value):  
        if attr == 'age':  
            self.__dict__[attr] = value  
        else:  
            raise AttributeError, attr + ' not allowed'
```

```
X = AccessControl()  
X.age = 40  
X.name = 'pythonlearn'  
AttributeError: name not allowed
```

Если мы используем метод `__setattr__`, все присваивания в нем придется выполнять посредством словаря атрибутов. Используйте `self.__dict__['age'] = x`, а не `self.name = x`:

Декораторы свойств

Для создания свойства-геттера над свойством ставится аннотация **@property**.

Для создания свойства-сеттера над свойством устанавливается аннотация **имя_свойства_геттера.setter**.

```
class Person:  
    def __init__(self, name):  
        self.__age = 0 # устанавливаем возраст  
  
    @property  
    def age(self):  
        return self.__age  
    #Свойство-сеттер определяется после свойства-геттера.  
    @age.setter  
    def age(self, age):  
        if 1 < age < 110:  
            self.__age = age  
        else:  
            print("Недопустимый возраст")  
  
tom = Person("Tom")  
tom.age = -100 # Недопустимый возраст
```



Наследование

Одиночное наследование

#Родительский класс помещается в скобки после имени класса. Объект производного класса наследует все свойства родительского.

```
class Tree(object):
    def __init__(self, kind, height):
        self.kind = kind
        self.age = 0
        self.height = height
    def grow(self):
        """ Метод роста """
        self.age += 1

class FruitTree(Tree):
    def __init__(self, kind, height):
        # Необходимо вызвать метод инициализации родителя.
        super().__init__(kind, height)

    def give_fruits(self):
        print ("Collected 20kg of {}".format(self.kind))

f_tree = FruitTree("apple", 0.7)
f_tree.give_fruits()
f_tree.grow()
```

Множественное наследование

При множественном наследовании дочерний класс наследует все свойства родительских классов. Синтаксис множественного наследования очень похож на синтаксис обычного наследования.

```
class Horse():
    isHorse = True
class Donkey():
    isDonkey = True
class Mule(Horse, Donkey):
    pass
mule = Mule()
mule.isHorse # True
mule.isDonkey # True
```

Многоуровневое наследование

Мы также можем наследовать класс от уже наследуемого. Это называется многоуровневым наследованием. Оно может иметь сколько угодно уровней.

В многоуровневом наследовании свойства родительского класса и наследуемого от него класса передаются новому наследуемому классу.

```
class Horse():
    isHorse = True
class Donkey(Horse):
    isDonkey = True
class Mule(Donkey):
    pass
mule = Mule()
mule.isHorse # True
mule.isDonkey # True
```

Композиция

Где класс является один класс является полем другого.

```
class Salary:  
    def __init__(self, pay):  
        self.pay = pay  
  
    def getTotal(self):  
        return (self.pay*12)  
  
class Employee:  
    def __init__(self, pay, bonus):  
        self.pay = pay  
        self.bonus = bonus  
        self.salary = Salary(self.pay)  
  
    def annualSalary(self):  
        return "Total: " + str(self.salary.getTotal()) +  
self.bonus)  
  
employee = Employee(100, 10)  
print(employee.annualSalary())
```

Полиморфизм

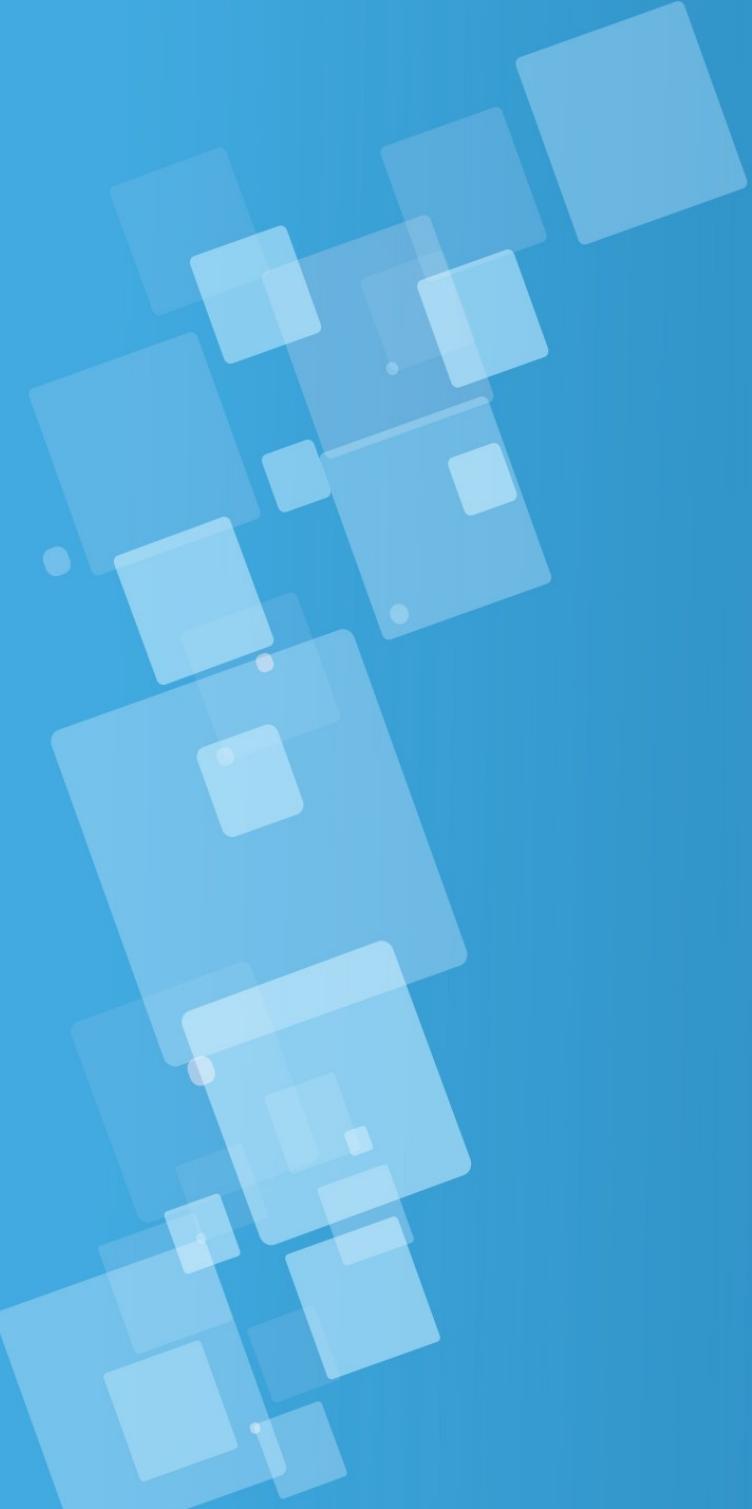
Все методы в языке изначально **виртуальные**. Это значит, что дочерние классы могут их переопределять и решать одну и ту же задачу разными путями, а конкретная реализация будет выбрана только во время исполнения программы. Такие классы называют полиморфными.

```
class Mammal:  
    def move(self):  
        print('Двигается')  
  
class Hare(Mammal):  
    def move(self):  
        print('Прыгает')  
  
animal = Mammal()  
animal.move() # Двигается  
hare = Hare()  
hare.move() # Прыгает
```

```
class Animal:  
    def __init__(self, name):  
        self.name = name  
  
class Horse(Animal):  
    def run(self, distance):  
        print(self.name, "run", distance, "meters")  
  
    def sound(self):  
        print(self.name, "says: igogo")  
  
class Bird(Animal):  
    def fly(self, distance):  
        print(self.name, "fly", distance, "meters")  
  
    def sound(self):  
        print(self.name, "says: chirik")  
  
class Unicorn( Bird, Horse):  
    pass  
  
pegas = Unicorn("Пегас") ← Вызов констуктора баз. класса  
pegas.run(20)  
pegas.fly(800)  
pegas.sound()           ← Что будет издавать Пегас ?
```



Продолжение следует ...



ООП в Python

Часть III



Типы методов

В Python есть еще два типа функций, которые могут быть созданы в классе:

- Статические методы.
- Методы класса.



@classmethod
and
@staticmethod

Статический метод

- Может быть определен только внутри класса, но не для объектов класса.
- Его можно вызвать непосредственно из класса по ссылке на имя класса.
- Он не может получить доступ к атрибутам класса
- Статический метод связан с классом. Таким образом, он не может изменить состояние объекта.
- Он также используется для разделения служебных методов для класса.
- Все объекты класса используют только одну копию статического метода.

Есть два способа определить статический метод в Python:

Использование метода **staticmethod()**

Использование декоратора **@staticmethod**.

Определение Static Method in Python

#Перед реализацией метода нужно добавить декоратор
@staticmethod

```
class Calc:  
    @staticmethod  
    def add(arg1, arg2):  
        return arg1 + arg2  
Calc.add(2,3)
```

После объявления класса сделать обычный метод
статическим

```
class Employee:  
    def sample(x):  
        print('Inside static method', x)
```

```
Employee.sample = staticmethod(Employee.sample)  
# call static method  
Employee.sample(10)
```

Вызов через класс

```
class DB:  
    # Определяем статический метод используя декоратор  
    @staticmethod  
    def get_conn():  
        print("Получим дискриптор соединения с БД!")  
  
    # Вызов метода  
conn = DB.get_conn()
```

Вызов через объект

```
class DB:  
    # Определяем статический метод используя декоратор  
    @staticmethod  
    def get_conn():  
        print("Получим дискриптор соединения с БД!")  
  
test_system_db = DB()  
test_system_db.get_conn()
```

При таком вызове не происходит подкапотной передачи `self`. А значит нет доступа к атрибутам объекта.

А можно как то это исправить ?

Учим статический метод работать с экземпляром класса

```
class DB:  
  
    def __init__(self):  
        self.name = "TestSystem"  
  
    @staticmethod  
    # Определяем переменную которая будет принимать объект  
    def get_conn(self):  
  
        print(f"Получим дискриптор соединения с БД  
              {self.name}!")  
  
test_sysytem_db = DB()  
test_sysytem_db.get_conn(test_sysytem_db)
```

Вызов статического метода из обычного

```
class DB:  
    # Определяем статический метод используя декоратор  
    @staticmethod  
    def get_conn():  
        print("Получим дискриптор соединения с БД!")  
    def get_session(self):  
        # вызов статического метода  
        conn = DB.get_conn()  
  
test_system_db = DB()  
test_system_db.get_session()
```

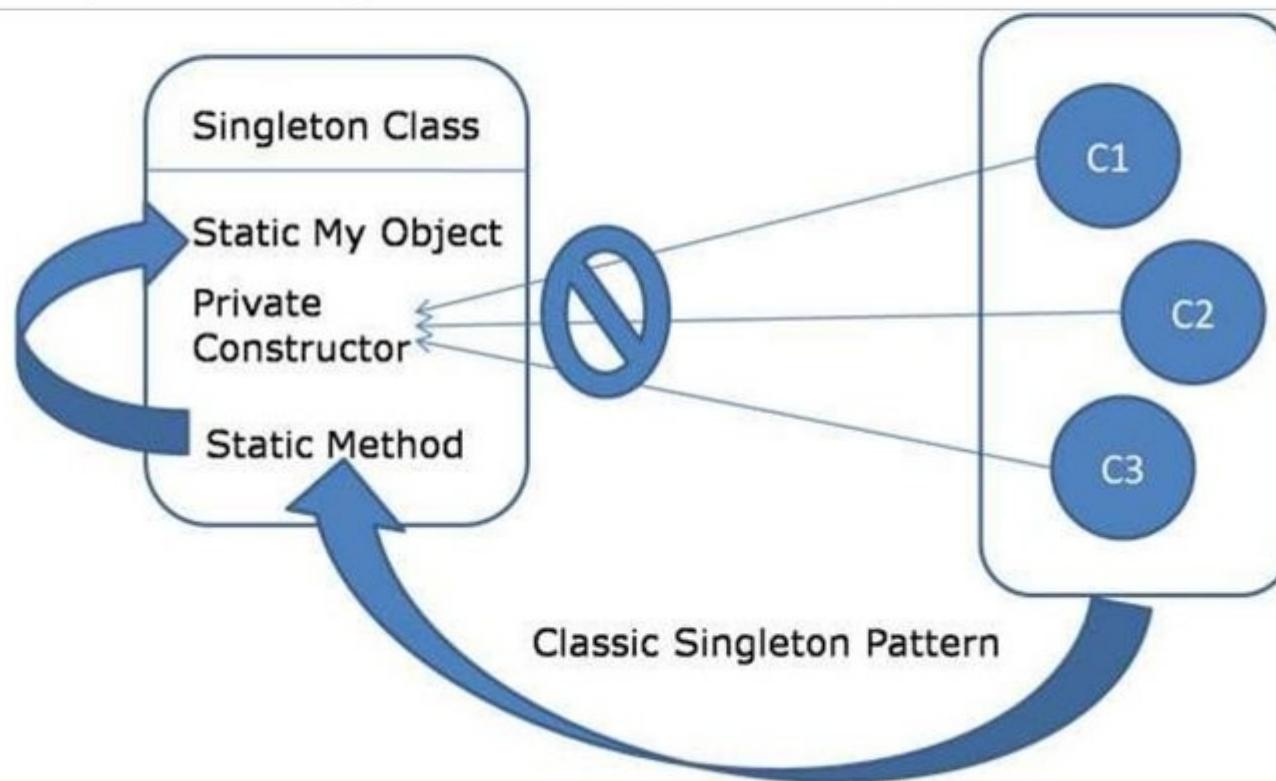


Для чего мы используем static methods ?

Паттерн Singleton

Синглтон (одиночка) – это паттерн проектирования, цель которого ограничить возможность создания объектов данного класса одним экземпляром. Он обеспечивает глобальность до одного экземпляра и глобальный доступ к созданному объекту.

Паттерн Singleton для подключения бд



```
class DB:  
    __instance__ = None  
  
    def __init__(self):  
        # Проверяем конструктор на сущ. экземпляр  
        if DB.__instance__ is None:  
            DB.__instance__ = self  
        else:  
            raise Exception("We can not creat another class")  
  
@staticmethod  
def get_instance():  
    # We define the static method to fetch instance  
    if not DB.__instance__:  
        DB()  
    return DB.__instance__  
  
mongo = DB()  
print(mongo)  
  
my_db = DB.get_instance()  
print(my_db)  
  
another_db = DB.get_instance()  
print(another_db)  
  
new_gover = DB() ← Что будет при вызове ?
```

Методы класса

@classmethod — это метод, который получает класс в качестве неявного первого аргумента, точно так же, как обычный метод экземпляра получает экземпляр. Это означает, что вы можете использовать класс и его свойства внутри этого метода, а не конкретного экземпляра.

Метод класса

- Может быть определен только внутри класса
- Получает класс в качестве неявного первого аргумента
- Его можно вызвать непосредственно из класса по ссылке на имя класса.
- Он не может получить доступ к атрибутам класса
- Не может изменить состояние объекта.
- Все объекты класса используют только одну копию метода класса.

Есть два способа определить статический метод в Python:

Использование метода **classmethod()**

Использование декоратора **@classmethod**.

Объявление метода `@classmethod`

```
class MyClass():
    TOTAL_OBJECTS = 0
    def __init__(self):
        MyClass.TOTAL_OBJECTS = MyClass.TOTAL_OBJECTS + 1

    @classmethod
    def total_objects(cls):
        print("Total objects: ", cls.TOTAL_OBJECTS)

# Вызов через объект
my_obj1 = MyClass()
my_obj1.total_objects()

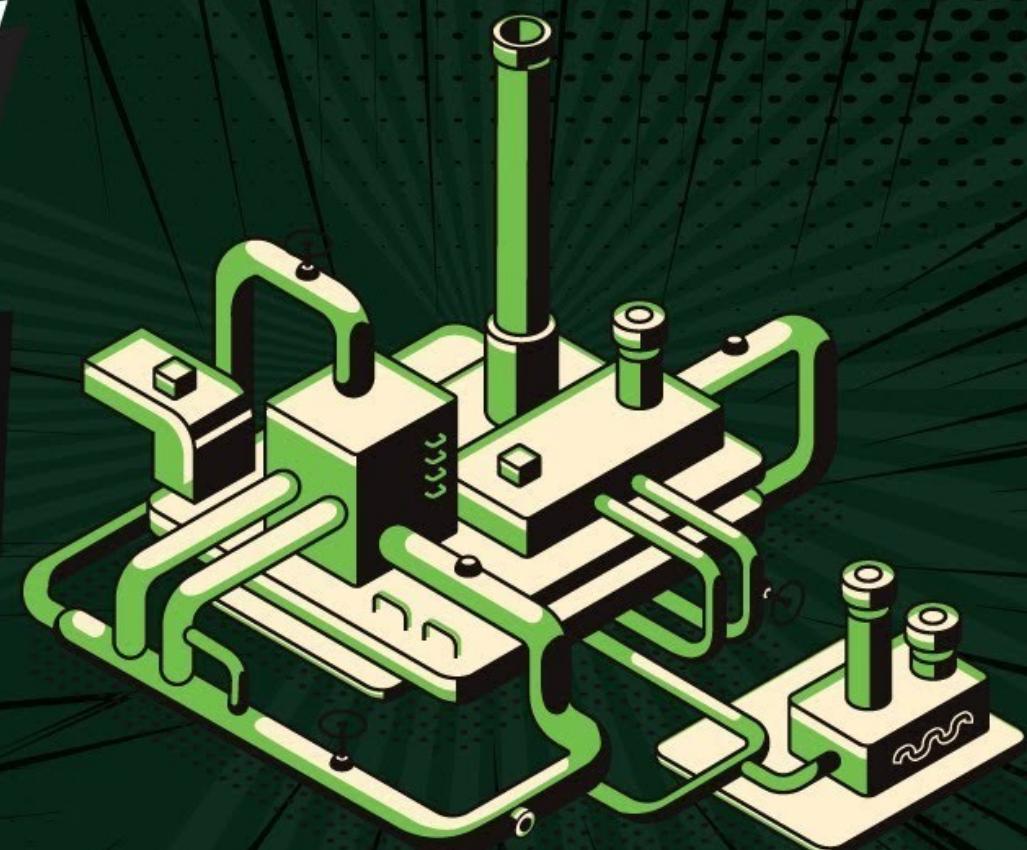
# Вызов через класс
MyClass.total_objects() ← что вернет вызов ?
```

Объявление метода `classmethod()`

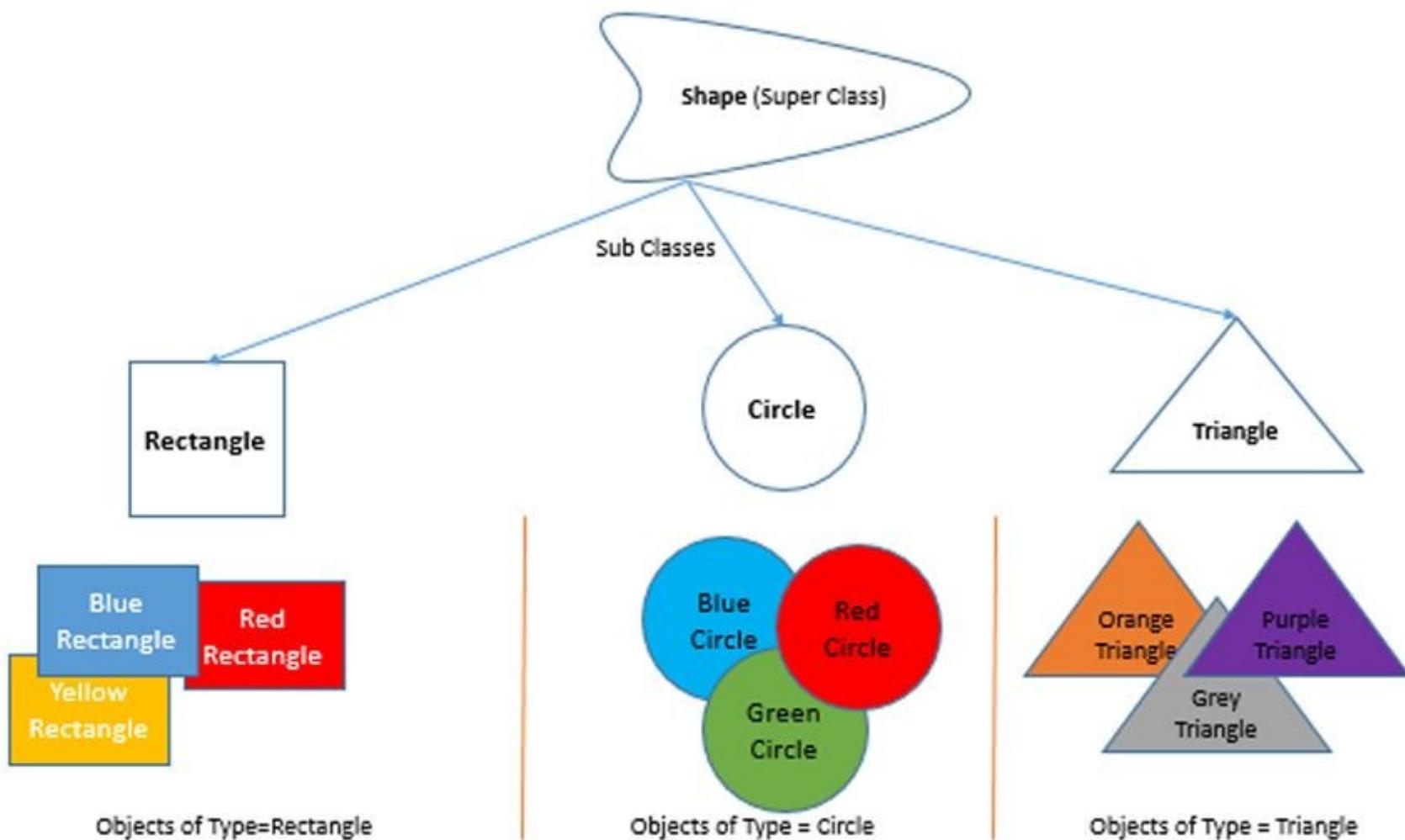
```
class Coffee:  
    def __init__(self, milk, beans):  
        self.milk = milk # percentage  
        self.coffee = 100 - milk  
        self.beans = beans  
  
    def __repr__(self):  
        return f'Milk={self.milk}% Coffee={self.coffee}%  
                Beans={self.beans}'  
    def cappuccino(cls):  
        return cls(80, 'Arrabica')  
  
Coffee.cappuccino = classmethod(Coffee.cappuccino)  
print(Coffee.cappuccino())
```

Шаблон проектирования “Фабрика”

FACTORY
PATTERN



Class Object in Python



```
# Рассмотрим фабрику
class Shape:
    def draw(self):
        raise NotImplementedError('This method should
have implemented.')

class Triangle(Shape):
    def draw(self):
        print("треугольник")

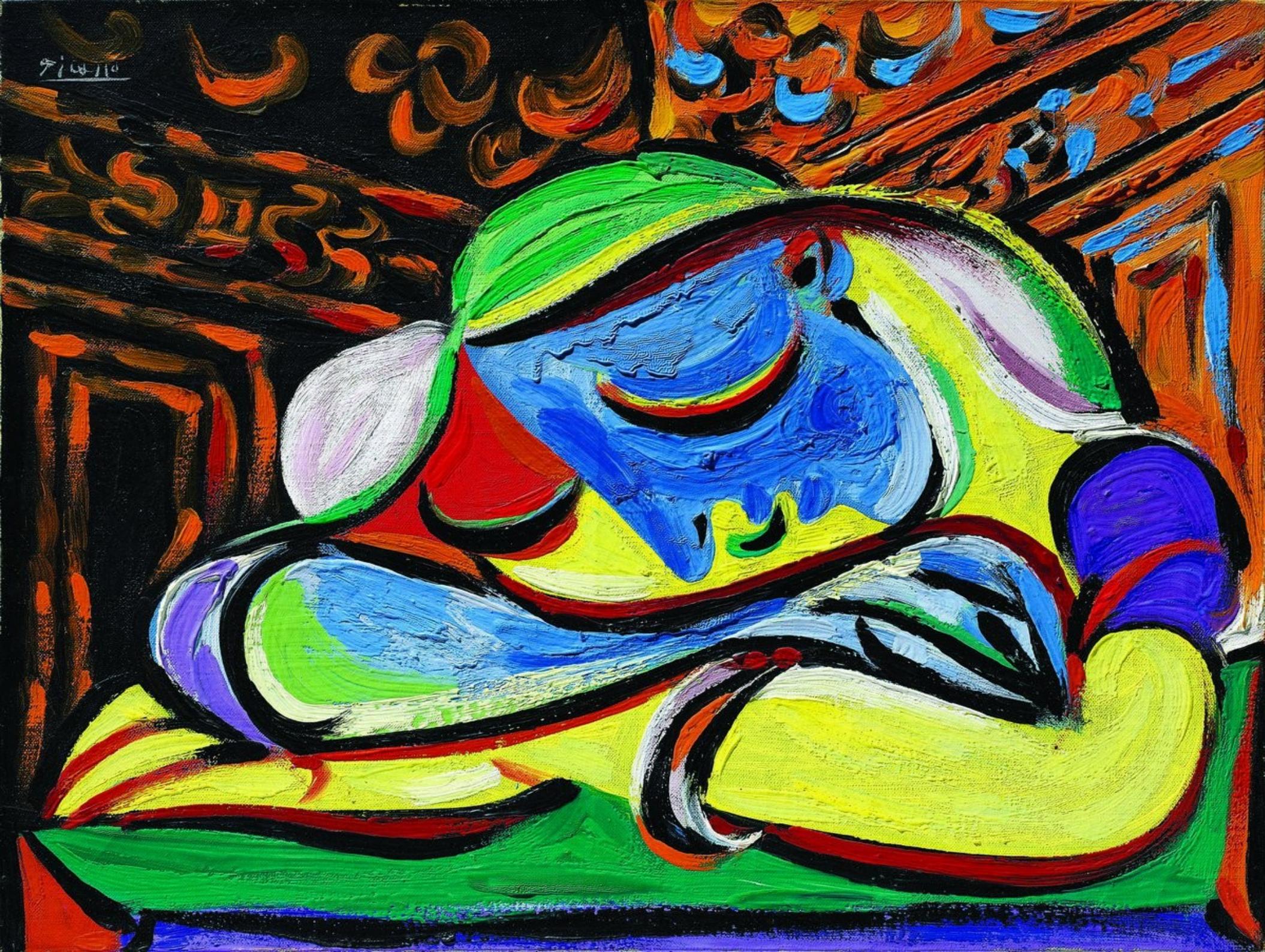
class Rectangle(Shape):
    def draw(self):
        print("прямоугольник")

class ShapeFactory:
    def getShape(self, shapeType):
        if shapeType == 'Triangle':
            return Triangle()
        elif shapeType == 'Rectangle':
            return Rectangle()
        else:
            pass
obj = ShapeFactory()
trgl = obj.getShape("Triangle")
trgl.draw()
```

```
class Coffee:  
    def __init__(self, milk, beans):  
        self.milk = milk # percentage  
        self.coffee = 100-milk # percentage  
        self.beans = beans  
    def __repr__(self):  
        return f'Milk={self.milk}% Coffee={self.coffee}%  
                Beans={self.beans}'  
@classmethod  
def cappuccino(cls):  
    return cls(80, 'Arrabica')  
  
@classmethod  
def espresso_macchiato(cls):  
    return cls(30, 'Robusta')  
  
@classmethod  
def latte(cls):  
    return cls(95, 'Arrabica')  
  
print(Coffee.cappuccino())  
print(Coffee.espresso_macchiato())  
print(Coffee.latte())
```

Заключение

Декоратор аннотации `@classmethod` используется для создания фабричных методов, поскольку они могут принимать любой ввод и предоставлять объект класса на основе параметров и обработки.



Абстрактные классы и методы в Python

- **Абстрактные классы** – реализуют механизм организации объектов в иерархии, позволяющий утверждать о наличии требуемых методов.
- **Абстрактный метод** – это метод для которого отсутствует реализация. Объявляется с помощью декоратора `@abstractmethod` из модуля `abc`

Абстрактные классы и методы в Python

Чтобы объявить абстрактный класс, нам сначала нужно импортировать модуль abc . Давайте посмотрим на пример.

```
from abc import ABC
class abs_class(ABC):
    @abstractmethod
    def render(self):
        pass
```

Абстрактный базовый класс – класс, на основе которого нельзя создать экземпляр объекта.

Абстрактный метод – это метод, определенный в базовом классе, но он может не обеспечивать какую-либо реализацию

Абстрактный базовый класс – класс, на основе которого нельзя создать экземпляр объекта.

```
from abc import ABC
class AbsClass(ABC):
    @abstractmethod
    def render(self):
        pass

obj = AbsClass() # вызовет ошибку
```

Чтобы объявить абстрактный класс, нам сначала нужно импортировать модуль abc .

```
from abc import ABC, abstractmethod

class Absclass(ABC):
    def print(self, x):
        print("Passed value: ", x)
    @abstractmethod
    def task(self):
        print("We are inside Absclass task")

class test_class(Absclass):
    def task(self):
        print("We are inside test_class task")

test_obj = test_class()
test_obj.task()
test_obj.print("10")
```



Продолжение следует....



Обработка исключения

try.. except..



Обработка исключений в Python

Программа, написанная на языке Python, останавливается сразу как обнаружит ошибку. Ошибки могут быть:

Синтаксические ошибки — возникают, когда написанное выражение не соответствует правилам языка (например, написана лишняя скобка);

Логические ошибки — это ошибки, когда синтаксис действительно правильный, но логика не та, какую вы предполагали. Программа работает успешно, но даёт неверные результаты.

Исключения — возникают во время выполнения программы (например, при делении на ноль).



Перехват ошибок во время выполнения

Не всегда при написании программы можно сказать возникнет или нет в данном месте исключение. Чтобы приложение продолжило работу при возникновении проблем, такие ошибки нужно перехватывать и обрабатывать с помощью ловушки **try/except**.

В каких случаях нужно предусматривать обработку ошибок ?

- **Работа с файлами** (нет прав доступа , диск переполнен и т.д)
- **Работа с СУБД** (сервер не доступен, ошибка выполнения запроса и т.д)
- **Работа с сетью** (сеть недоступна, ошибка соединения, обработка кода возврата и т.д)
- **Работа с различными библиотеками** которые бросают exception в случае обнаружения ошибки.

Как устроен механизм исключений ?

В Python есть встроенные исключения, которые появляются после того как приложение находит ошибку. В этом случае текущий процесс временно приостанавливается и передает ошибку на уровень вверх до тех пор, пока она не будет обработана. В случае когда ошибка не обрабатывается, программа прекратит свою работу и в консоли мы увидим Traceback с подробным описанием ошибки.

Иерархия классов исключений Python

<https://docs.python.org/3/library/exceptions.html>

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        +-- FloatingPointError
        +-- OverflowError
        +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        +-- ModuleNotFoundError
    +-- LookupError
        +-- IndexError
        +-- KeyError
    +-- MemoryError
    +-- NameError
        +-- UnboundLocalError
    +-- OSError
```

BaseException – базовое исключение, от которого берут начало все остальные.

SystemExit – исключение, порождаемое функцией `sys.exit` при выходе из программы.

KeyboardInterrupt – порождается при прерывании программы пользователем

GeneratorExit – порождается при вызове метода `close` объекта `generator`.

Exception – а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обычновенные, с которыми можно работать.

StopIteration – порождается встроенной функцией `next`, если в итераторе больше нет элементов.

ArithmeticalError – арифметическая ошибка.

FloatingPointError – порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.

OverflowError – возникает, когда результат арифметической операции слишком велик для представления.

ZeroDivisionError – деление на ноль.

Генерация исключения

Напишем код, который будет создавать исключительную ситуацию. К примеру, попробуем поделить число на 0:

```
>>> print(1 / 0)
```

В командной оболочке получим следующее:

Traceback (most recent call last) :

```
  File "", line 1, in 
ZeroDivisionError: division by zero
```

Разберём это сообщение подробнее:

Интерпретатор нам сообщает о том, что он поймал исключение и напечатал информацию: Traceback (most recent call last).

Далее имя файла File "". Имя пустое, потому что этот код был запущен в интерактивном интерпретаторе, строка в файле line 1;

Название исключения **ZeroDivisionError** и краткое описание исключения division by zero.

При выбросе исключения программа закрывается и не выполняет код, который следует за строкой, в которой произошло исключение!

Windows

A fatal exception 0E has occurred at 0028:C0011E36 in UXD UMM(01) + 00010E36. The current application will be terminated.

- * Press any key to terminate the current application.
- * Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all your applications.

Press any key to continue _

Синтаксис конструкций

try:

{

Run this code

except:

{

Execute this code when
there is an exception

else:

{

No exceptions? Run this
code.

finally:

{

Always run this code.

Пример

```
try:  
    a = 1/0  
  
except ZeroDivisionError:  
    print("Возникло исключение: ошибка деления на ноль! ")  
    a = 0  
  
else:  
    print("Ветка else вызывается если не возникло  
исключения")  
  
finally:  
    print("Аккуратно обрабатываем ошибку и идем дальше...")
```

Ловушка для двух исключений

```
from calc import div

try:
    result = div(100, 0)
    print("Расчёт проведён успешно")
except (ZeroDivisionError, KeyError) as e:
    print("Ошибка деления или ошибка обращения по
          ключу. Вот она:", e)
print(result)
```

Несколько ловушек

```
from calc import div
try:
    result = div(100, 0)
    print("Расчёт проведён успешно")
except ZeroDivisionError as e:
    print("Ошибка деления произошла", e)
except KeyError as e:
    print("Ошибка обращения по ключу произошла:", e)
print(result)
```

Перехват ошибки чтения

```
try:  
    file = open('ok123.txt', 'r')  
except FileNotFoundException as e:  
    print(e)  
  
> [Errno 2] No such file or directory: 'ok123.txt'
```

Порядок следования обработки

```
try:  
    file = open('ok123.txt', 'r')  
except Exception as e:  
    Print(Exception, e)  
except FileNotFoundException as e:  
    Print(FileNotFoundException, e)
```

Какой блок поймает исключение если файл не обнаружен ?

Блок finally выполняется всегда

```
try:  
    file = open('ok.txt', 'r')  
    lines = file.readlines()  
    print(lines[5]) ← Обращение к несуществ. строке  
except IndexError as e:  
    print(e)  
finally:  
    file.close()  
    if file.closed:  
        print("файл закрыт!")  
> файл закрыт!
```

Каскадное включение перехватчиков

```
try:
```

```
.....
```

```
try:
```

```
.....
```

```
except E:
```

```
.....
```

```
except E:
```

```
.....
```

```
import numpy as np

def divide(x, y):
    try:
        out = x/y
    except:
        try:
            out = np.inf * x / abs(x)
        except:
            out = np.nan
    finally:
        return out

divide(15, 3)    # 5.0
divide(15, 0)    # inf
divide(-15, 0)   # -inf
divide(0, 0)      # nan
```

Генерация исключений в Python

Для принудительной генерации исключения используется инструкция **raise**.

```
try:
```

```
    raise Exception("Что то пошло не так")
```

```
except Exception as e:
```

```
    print("Message:" + str(e))
```

Валидатор входного строкового значения на имя человека.

```
def validate(name):  
    if len(name) < 10:  
        raise ValueError  
  
try:  
    name = input("Введите имя:")  
    validate(name)  
  
except ValueError:  
    print("Имя слишком короткое:")
```

Пользовательские исключения

В Python можно создавать собственные исключения. Такая практика позволяет увеличить гибкость процесса обработки ошибок в рамках той предметной области, для которой написана ваша программа.

```
class NameTooShortError(ValueError):
    pass

def validate(name):
    if len(name) < 10:
        raise NameTooShortError
```

Пользовательские исключения в Python

Для реализации собственного типа исключения необходимо создать класс, являющийся наследником от одного из классов исключений.

```
class NegValException(Exception):
    pass

try:
    val = int(input("input positive number: "))
    if val < 0:
        raise NegValException("Neg val: " + str(val))
    print(val + 10)
except NegValException as e:
    print(e)
```

Вызов конструктора базового класса

```
class NegValException(Exception):  
    def __init__(self, number):  
        super().__init__(f"Neg val: {number}")  
        self.number = number  
  
try:  
    val = int(input("input positive number: "))  
    if val < 0:  
        raise NegValException(val)  
  
except NegValException as e:  
    print(e)
```



Продолжение следует...

Handle psycopg2 exceptions that occur while connecting to PostgreSQL

```
# declare a new PostgreSQL connection object
try:
    conn = connect(
        dbname = "python_test",
        user = "WRONG_USER",
        host = "localhost",
        password = "mypass"
    )
except OperationalError as err:
    # pass exception to function
    print_psycopg2_exception(err)

    # set the connection to 'None' in case of error
conn = None
```

```
def print_psycopg2_exception(err):
    # get details about the exception
    err_type, err_obj, traceback = sys.exc_info()

    # get the line number when exception occurred
    line_num = traceback.tb_lineno

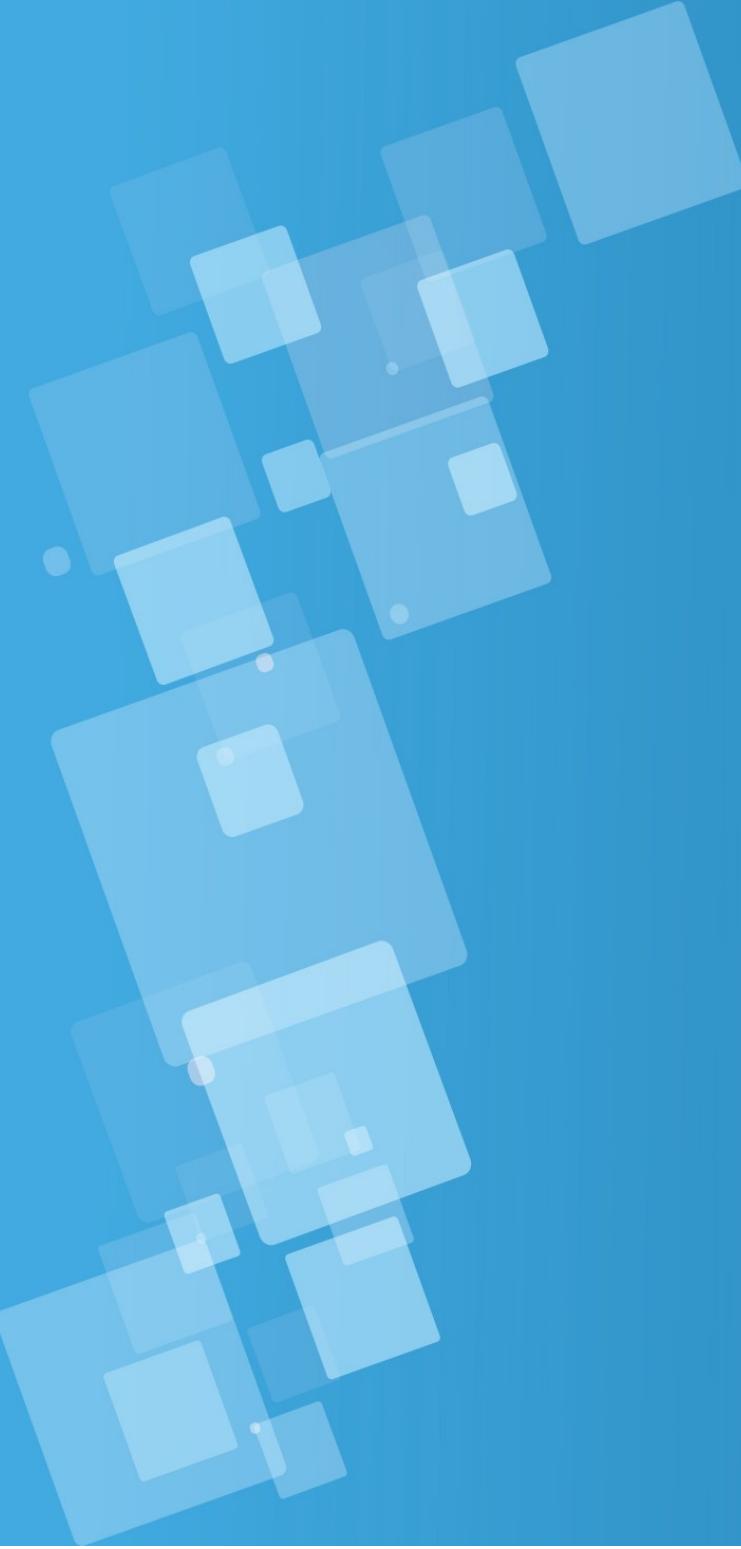
# print the connect() error
    print ("\npsycopg2 ERROR:", err, "on line
number:", line_num)
    print ("psycopg2 traceback:", traceback, "--
type:", err_type)

# psycopg2.extensions.Diagnostics object attribute
print ("\nextensions.Diagnostics:", err.diag)

# print the pgcode and pgerror exceptions
print ("pgerror:", err.pgerror)
print ("pgcode:", err.pgcode, "\n")
```

Pecypc

<https://kb.objectrocket.com/postgresql/python-error-handling-with-the-psycopg2-postgresql-adapter-645>



Магические методы в ООП



Table 1-1. Special method names (operators excluded)

Category	Method names
String/bytes representation	<code>__repr__</code> , <code>__str__</code> , <code>__format__</code> , <code>__bytes__</code>
Conversion to number	<code>__abs__</code> , <code>__bool__</code> , <code>__complex__</code> , <code>__int__</code> , <code>__float__</code> , <code>__hash__</code> , <code>__index__</code>
Emulating collections	<code>__len__</code> , <code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__contains__</code>
Iteration	<code>__iter__</code> , <code>__reversed__</code> , <code>__next__</code>
Emulating callables	<code>__call__</code>
Context management	<code>__enter__</code> , <code>__exit__</code>
Instance creation and destruction	<code>__new__</code> , <code>__init__</code> , <code>__del__</code>
Attribute management	<code>__getattr__</code> , <code>__getattribute__</code> , <code>__setattr__</code> , <code>__delattr__</code> , <code>__dir__</code>
Attribute descriptors	<code>__get__</code> , <code>__set__</code> , <code>__delete__</code>
Class services	<code>__prepare__</code> , <code>__instancecheck__</code> , <code>__subclasscheck__</code>

Методы конструирования и инициализации `__new__` `__init__`

```
class Person(object):
    instance = None
    def __new__(cls, *args, **kwargs):
        if not cls.instance:
            cls.instance = object.__new__(cls)
        return cls.instance
    def __init__(self):
        self.__name = 'Peter I'

obj_one = Person()
obj_two = Person()
print( obj_one is obj_two )
```

Метод `__new__` вызывается первым. Он открывает пространство памяти затем вызывается метод `__init__`. Используется для переопределения immutable классов (`int`, `str`, `tuple`)

Метод документирования doc

```
# дандер для документирования класса и методов
# Комментарии нужно помещать сразу после объявления
class Test(object) :
    ''' Класс Test для демонстрации '''

    def show(self):
        ''' This is Show Function DocString '''
        pass

t = Test()
print(t.doc)                      # Описание класса
print(Test.doc)                    # Описание класса
print(t.show.doc)                  # Описание описание метода
```

Стрековые методы `__str__` `__repr__`

```
class Car:  
    def __init__(self, model, color, vin):  
        self.model = model  
        self.color = color  
        self.VIN = vin  
  
car = Car("Mercedes-benz", "silver", "WDB1240221J081498")
```

```
print(car)  
<__main__.Car object at 0x7fe009b78a60>
```

```
print(str(car))  
<__main__.Car object at 0x7fe009b78a60>
```

1. Для преобразования строк в классах можно использовать дандер методы `__str__` и `__repr__`
2. В свои классы всегда следует добавлять метод `__repr__`.

Строковые методы `__str__` `__repr__`

Дандеры для отображения объекта в виде строки вызываются при работе с функциями `print()` `str()`

```
class Car:  
    def __init__(self, model, color, vin):  
        self.model = model  
        self.color = color  
        self.VIN = vin  
  
    def __str__(self):  
        return f"Модель: {self.model} с VIN номером  
               {self.VIN}"  
  
    def __repr__(self):  
        return f"Модель: {self.model} с VIN номером  
               {self.VIN}"  
  
car = Car("Mercedes-benz", "silver", "WDB1240221J081498")  
print(car)  
print(str(car))
```

А что будет если нет строковых дандеров ?

Магические методы сравнения

`__eq__(self, other)`

Определяет поведение оператора равенства `==`

`__ne__(self, other)`

Определяет поведение оператора неравенства, `!=`

`__lt__(self, other)`

Определяет поведение оператора меньше, `<`

`__gt__(self, other)`

Определяет поведение оператора больше, `>`

`__le__(self, other)`

Определяет поведение оператора меньше или равно, `<=`

`__ge__(self, other)`

Определяет поведение оператора больше или равно, `>=`

Откуда такие обозначения ?

```
#!/bin/bash
#This Script accepts a positive integer from the user
#and calculate its factorial
#Date: Sep 2015
if [ $# -ne 1 ]; then
    echo "Error: One Argument Expected"
    exit 3
else
    if [ "$1" -eq "$1" 2> /dev/null ]; then
        if [ $1 -ge 0 ]; then
            result=1
            for i in `seq $1`
            do
                let "result*=$i"
            done
            echo "Factorial of $1 equals $result"
        else
            echo "You Should Enter a Positive Integer"
            exit 2
        fi
    else
        echo "You Should Enter an Integer"
        exit 1
    fi
fi
```

Равенство значений объектов класса __eq__

```
class Car:  
    def __init__(self, model, color, vin):  
        self.model = model  
        self.color = color  
        self.VIN = vin  
  
    def __eq__(self, obj):  
        if not isinstance(obj, Car):  
            raise ValueError("Передан другой тип  
объекта")  
        return (self.VIN == obj.VIN)
```

```
car_one = Car("Mercedes-benz", "silver", "WDB1240221J081498")  
car_two = Car("Mercedes-benz", "red", "WDB1240221J081498")  
print(car_one == car_two) ← Что вернет сравнение ?
```

Что нужно для полного сравнения ?

Оператор больше `__gt__`

```
class Car:  
    def __init__(self, model, price):  
        self.model = model  
        self.price = price  
  
    def __gt__(self, obj):  
        if not isinstance(obj, Car):  
            raise ValueError("Передан другой тип объекта")  
        return (self.price > obj.price)  
  
    def __str__(self):  
        return f"Модель: {self.model} с ценой {self.price}"  
  
car_one = Car("Mercedes-benz", "5000000")  
car_two = Car("Aurus Senat", "5000000")  
print(car_one > car_two) ← ?  
print(car_one < car_two) ← Что если изменить знак ?
```

Оператор меньше `__lt__`

```
class Car:  
    def __init__(self, model, price):  
        self.model = model  
        self.price = price  
  
    def __lt__(self, obj):  
        if not isinstance(obj, Car):  
            raise ValueError("Передан другой тип объекта")  
        return (self.price < obj.price)  
  
    def __str__(self):  
        return f"Модель: {self.model} с ценой {self.price}"  
  
car_one = Car("Mercedes-benz", "5000000")  
car_two = Car("Aurus Senat", "50000000")  
print(car_one > car_two) ← ?
```

Название класса **__name__**

```
class Car():
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __str__(self):
        return f"Модель: {self.model} с ценой
{self.price} "
```



```
print(Car.__name__)
obj = car("Lada", "800000")
print(obj.__name__) ?
```

dict словарь для хранения атрибутов

```
class Car():
    color = "Red"
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __str__(self):
        return f"Модель: {self.model} с ценой
{self.price} "

obj = Car("Mercedes-benz", 5_000_000)
print(obj.__dict__)
{'model': 'Mercedes-benz', 'price': 5000000}
```

Если это словарь то ?

dict

```
# Добавим атрибут

class Car:

    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price}"
    "


obj = Car("Mercedes-benz", 5_000_000)
obj.__dict__['color'] = "red"

print(obj.__dict__)
```

dict словарь для хранения атрибутов

Посмотрим атрибуты класса

```
class Car():
    "Класс Car"
    color= "Red"
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price} "

print(Car.__dict__)

{ '__module__': '__main__',
  '__doc__': 'Класс Car',
  'color': 'Red',
  '__init__': <function Car.__init__ at 0x7f74c5c9d790>,
  '__str__': <function Car.__str__ at 0x7f74c5c9d8b0>,
  '__dict__': <attribute '__dict__' of 'Car' objects>,
  '__weakref__': <attribute '__weakref__' of 'Car' objects>}
```

__slots__ ограничение атрибутов

Когда мы создаем объект класса, атрибуты этого объекта сохраняются в словарь под названием __dict__.

```
class Article:  
    def __init__(self, date, writer):  
        self.date = date  
        self.writer = writer  
  
article = Article("2020-06-01", "xiaoxu")  
article.reviewer = "jojo"  
print(article.__dict__)  
{'date': '2020-06-01', 'writer': 'xiaoxu', 'reviewer':  
'jojo'}
```

slots ограничение атрибутов

Определив волшебный метод `__slots__` мы ограничим кол-во атрибутов для экземпляра класса.

```
class Article:  
    __slots__ = ["date", "writer"]  
  
    def __init__(self, date, writer):  
        self.date = date  
        self.writer = writer  
  
article = Article("2020-06-01", "xiaozi")  
article.reviewer = "jojo" ← Что произойдет ?  
print(article.__dict__) ← вызовет ошибку
```

Заключение

- Магический метод — это специальные методы, которые вызываются неявно.
- Также это подход python к перегрузке операторов, позволяющий классам определять свое поведение в отношении операторов языка. Из этого можно заключить, что интерпретатор имеет "некую таблицу" соответствия операторов к методам класса. Перегружая эти методы, вы можете управлять "поведением" операторов языка относительно вашего класса.

The background of the image is a dark, almost black, space. Overlaid on it are intricate, glowing fractal patterns. These patterns consist of numerous thin, curved lines that form complex, swirling shapes. The primary colors of these lines are shades of purple and blue, with some yellow and white highlights where the curves intersect or are more densely packed. The overall effect is one of depth and motion, resembling a microscopic view of a celestial body or a complex mathematical structure.

Продолжение следует...



Итераторы, генераторы и менеджеры контента

Итератор

- Итератор (iterator) - это объект, который используется для прохода по итерируемому элементу.
- В основном используется для коллекция(списки, словари и т.д)
- Протокол Iterator в Python включает две функции. Один - `iter()`, другой - `next()`. И два дандера `__iter__` `__next__`

Iterator in Python



`__iter__`(It convert Iterable to Iterator Object)

`__next__`(It return the next element from the collection)

Дандер iter

Рассмотрим пример:

```
num_list = [1, 2, 3, 4, 5]  
for i in num_list:  
    print(i)  
dir(num_list)
```

```
['__add__', '__class__', '__contains__',  
'__delattr__', '__delitem__', '__dir__', '__doc__',  
'__eq__', '__format__', '__ge__', '__getattribute__',  
'__getitem__', '__gt__', '__hash__', '__iadd__',  
'__imul__', '__init__', '__init_subclass__',  
'__iter__', ...}
```

Дандер `__iter__`

Рассмотрим пример:

```
num_list = [1, 2, 3, 4, 5]
```

1. Конструкция `for` в момент выполнения берет у коллекции объект `__iter__`

```
for i in num_list:  
    print(i)
```

```
print(num_list.__iter__())
```

```
<method-wrapper '__iter__' of list object at 0x7f602f9b0bc0>
```

2. Получим объект итерации через функцию `iter()`

```
it = num_list.__iter__()
```

3. Посмотрим тип объекта

```
print(it) → <list_iterator object at 0x7f602f730d90>
```

```
dir(it)
```

```
['__class__', '__delattr__', '__dir__', '__doc__',  
'__init__', '__init_subclass__', '__iter__',  
'__new__', '__next__', ...]
```

Дом, который построил Джек

Вот дом,
Который построил Джек.

А это пшеница,
Которая в тёмном чулане хранится
В доме,
Который построил Джек.

А это весёлая птица-синица,
Которая часто ворует пшеницу,
Которая в тёмном чулане хранится
В доме,
Который построил Джек.

..

--Самуил Маршак

Вывод.

- Как мы могли убедиться, цикл **for** использует так называемые итераторы.
- Коллекция содержит метод-wrapper **__iter__**. Который в свою очередь возвращает объект итератор. В котором реализован метод **__next__**. Который в свою очередь возвращает следующий итерируемый элемент
- Цикла **for** можно разложить на след. операции:

```
num_list = [1, 2, 3, 4, 5]
itr = iter(num_list)
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
```

Создание собственных итераторов

Разобравшись как это работает мы можем написать собственную реализацию итератора. Достаточно реализовать дандер `__next__`

```
class SimpleIterator:
    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration

s_iter1 = SimpleIterator(3)
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1)) → Что произойдет ?
```

Итератор для цикла

Для итерации в цикле нам необходимо реализовать дандер **`__iter__`** который возвращает `self`

```
class SimpleIterator:  
    def __iter__(self):  
        return self  
    def __init__(self, limit):  
        self.limit = limit  
        self.counter = 0  
    def __next__(self):  
        if self.counter < self.limit:  
            self.counter += 1  
            return 1  
        else:  
            raise StopIteration  
s_iter2 = SimpleIterator(5)  
for i in s_iter2:  
    print(i)
```



Генераторы – это упрощенные итераторы

Чтобы облегчить написание итераторов используются генераторы и ключевое слово `yield`

Меняем бесконечный генератор на итератор

Перепишем итератор

```
class Repeater:  
    def __init__(self, value):  
        self.value = value  
    def __iter__(self):  
        return self  
    def __next__(self):  
        return self.value  
  
def repeater(value):  
    while True  
        yield value  
  
for x in repeater(0):  
    Print(x)  
#0  
#0  
#0
```

Как это работает ?

Генераторы похожи на нормальные функции , но их поведение различаются. Вызов функции-генератора вообще не выполняет функцию а просто создает и возвращает объект-генератор

```
def repeater(value):  
    while True  
        yield value
```

```
generator_obj = repeater("Hello")
```

Программный код выполняется тогда когда функция next вызывается с объектом генератора

```
next(generator_obj)
```

Как прекратить генерацию?

Генераторы прекращают порождать значения, как только поток управления возвращается из функции-генератора каким-либо иным способом, кроме инструкции `yield`

```
bounder_repeater(value, max_repeats):
    count = 0
    while True:
        If count >= max_repeats:
            return
        Count += 1
        yield value
```

```
for x in bounder_repeater("Раз",
    print())
“Раз”
“Раз”
```

Что заставляет генератор прекращать работу ?

```
def repeater_tree_value():
    yield 1
    yield 2
    yield 3

for i in repeater_tree_value():
    print(i)
1
2
3

it = repeater_tree_value()
print(next(it))
print(next(it))
print(next(it))
print(next(it)) → StopIteration
```

Выводы

- Функции-генераторы являются синтаксическим сахаром для написания объектов, которые поддеживают протокол итератора. Генератор абстрагирует от всей кухни шаблонного кода.
- Инструкция **yield** позволяет временно приостановить исполнение функции-генератора и передавать из него значения назад.
- Генераторы начинают вызывать исключения **StopIteration** после того, как поток управления покидает функцию-генератор каким-либо иным способом, кроме инструкции `yield`

Контекстные менеджеры и инструкция **with**

```
with open(path, 'w') as f_obj:  
    f_obj.write(some_data)
```

В данном случае конструкция **with** гарантирует автоматическое закрытие открытых дискрипторов файла после того, как выполнение программы покидает контекст инструкции `with`. На внутреннем уровне данный пример кода выглядит следующим образом.

```
f = open('hello', 'w')  
try:  
    f.write('hello, my sun!')  
finally:  
    f.close()
```

При возникновении исключения будет его обработка.

Что такое менеджер контекста ?

Менеджер контекста – это интерфейс который должен соблюдать объект для того, чтобы поддерживать инструкцию **with**. Чтобы объект функционировал как менеджер контекста нужно добавить в него методы **`__enter__`** и **`__exit__`**

`__enter__` – Python вызывает метод когда входит в контекст инструкции `with` и наступает момент получения ресурса.

`__exit__` – метод вызывается для высвобождения ресурса

Пример

```
class WritableFile:  
    def __init__(self, file_path):  
        self.file_path = file_path  
  
    def __enter__(self):  
        self.file_obj = open(self.file_path, mode="w")  
        return self.file_obj  
  
    def __exit__(self, exc_type, exc_val, exc_tb):  
        if self.file_obj:  
            self.file_obj.close()  
  
with WritableFile("hello.txt") as file:  
    file.write("Hello, World!")
```

```
import sqlite3
class DataConn:

    def __init__(self, db_name):
        """Конструктор"""
        self.db_name = db_name

    def __enter__(self):
        """
        Открываем подключение с базой данных.
        """
        self.conn = sqlite3.connect(self.db_name)
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        """
        Закрываем подключение.
        """
        self.conn.close()
        if exc_val:
            raise

with DataConn(db) as conn:
    cursor = conn.cursor()
```

Выводы

- Инструкция **with** упрощает обработку исключений путем инкапсуляции стандартных случаев применения инструкции try/finally в так называемые менеджеры контекста
- Чаще всего менеджер контекста используется для управления безопасным подключением и высвобождением системных ресурсов. Ресурсы выделяются при помощи инструкции **with** и высвобождаются автоматически , когда поток исполнения покидает **with**
- Эффективное применение инструкции **with** помогает избежать утечки ресурсов и облегчает ее восприятие.



Итераторы, генераторы и менеджеры контента

Итератор

- Итератор (iterator) - это объект, который используется для прохода по итерируемому элементу.
- В основном используется для коллекция(списки, словари и т.д)
- Протокол Iterator в Python включает две функции. Один - `iter()`, другой - `next()`. И два дандера `__iter__` `__next__`

Iterator in Python



`__iter__`(It convert Iterable to Iterator Object)

`__next__`(It return the next element from the collection)

Дандер iter

Рассмотрим пример:

```
num_list = [1, 2, 3, 4, 5]  
for i in num_list:  
    print(i)  
dir(num_list)
```

```
['__add__', '__class__', '__contains__',  
'__delattr__', '__delitem__', '__dir__', '__doc__',  
'__eq__', '__format__', '__ge__', '__getattribute__',  
'__getitem__', '__gt__', '__hash__', '__iadd__',  
'__imul__', '__init__', '__init_subclass__',  
'__iter__', ...}
```

Дандер `__iter__`

Рассмотрим пример:

```
num_list = [1, 2, 3, 4, 5]
```

1. Конструкция `for` в момент выполнения берет у коллекции объект `__iter__`

```
for i in num_list:  
    print(i)
```

```
print(num_list.__iter__())
```

```
<method-wrapper '__iter__' of list object at 0x7f602f9b0bc0>
```

2. Получим объект итерации через функцию `iter()`

```
it = num_list.__iter__()
```

3. Посмотрим тип объекта

```
print(it) → <list_iterator object at 0x7f602f730d90>
```

```
dir(it)
```

```
['__class__', '__delattr__', '__dir__', '__doc__',  
'__init__', '__init_subclass__', '__iter__',  
'__new__', '__next__', ...]
```

Дом, который построил Джек

Вот дом,
Который построил Джек.

А это пшеница,
Которая в тёмном чулане хранится
В доме,
Который построил Джек.

А это весёлая птица-синица,
Которая часто ворует пшеницу,
Которая в тёмном чулане хранится
В доме,
Который построил Джек.

..

--Самуил Маршак

Вывод.

- Как мы могли убедиться, цикл **for** использует так называемые итераторы.
- Коллекция содержит метод-wrapper **__iter__**. Который в свою очередь возвращает объект итератор. В котором реализован метод **__next__**. Который в свою очередь возвращает последующий итерируемый элемент
- Цикла **for** можно разложить на след. операции:

```
num_list = [1, 2, 3, 4, 5]
itr = iter(num_list)
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
```

Создание собственных итераторов

Разобравшись как это работает мы можем написать собственную реализацию итератора. Достаточно реализовать дандер `__next__`

```
class SimpleIterator:
    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration

s_iter1 = SimpleIterator(3)
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1)) → Что произойдет ?
```

Итератор для цикла

Для итерации в цикле нам необходимо реализовать дандер **`__iter__`** который возвращает `self`

```
class SimpleIterator:  
    def __iter__(self):  
        return self  
    def __init__(self, limit):  
        self.limit = limit  
        self.counter = 0  
    def __next__(self):  
        if self.counter < self.limit:  
            self.counter += 1  
            return 1  
        else:  
            raise StopIteration  
s_iter2 = SimpleIterator(5)  
for i in s_iter2:  
    print(i)
```



Генераторы – это упрощенные итераторы

Чтобы облегчить написание итераторов используются генераторы и ключевое слово `yield`

Меняем бесконечный генератор на итератор

Перепишем итератор

```
class Repeater:  
    def __init__(self, value):  
        self.value = value  
    def __iter__(self):  
        return self  
    def __next__(self):  
        return self.value  
  
def repeater(value):  
    while True  
        yield value  
  
for x in repeater(0):  
    Print(x)  
#0  
#0  
#0
```

Как это работает ?

Генераторы похожи на нормальные функции , но их поведение различаются. Вызов функции-генератора вообще не выполняет функцию а просто создает и возвращает объект-генератор

```
def repeater(value):  
    while True  
        yield value
```

```
generator_obj = repeater("Hello")
```

Программный код выполняется тогда когда функция next вызывается с объектом генератора

```
next(generator_obj)
```

Как прекратить генерацию?

Генераторы прекращают порождать значения, как только поток управления возвращается из функции-генератора каким-либо иным способом, кроме инструкции `yield`

```
bounder_repeater(value, max_repeats):
    count = 0
    while True:
        if count >= max_repeats:
            return
        count += 1
        yield value
```

```
for x in bounder_repeater("Раз",
    print())
“Раз”
“Раз”
```

Что заставляет генератор прекращать работу ?

```
def repeater_tree_value():
    yield 1
    yield 2
    yield 3

for i in repeater_tree_value():
    print(i)
1
2
3

it = repeater_tree_value()
print(next(it))
print(next(it))
print(next(it))
print(next(it)) → StopIteration
```

Выводы

- Функции-генераторы являются синтаксическим сахаром для написания объектов, которые поддеживают протокол итератора. Генератор абстрагирует от всей кухни шаблонного кода.
- Инструкция **yield** позволяет временно приостановить исполнение функции-генератора и передавать из него значения назад.
- Генераторы начинают вызывать исключения **StopIteration** после того, как поток управления покидает функцию-генератор каким-либо иным способом, кроме инструкции `yield`

Контекстные менеджеры и инструкция **with**

```
with open(path, 'w') as f_obj:  
    f_obj.write(some_data)
```

В данном случае конструкция **with** гарантирует автоматическое закрытие открытых дискрипторов файла после того, как выполнение программы покидает контекст инструкции `with`. На внутреннем уровне данный пример кода выглядит следующим образом.

```
f = open('hello', 'w')  
try:  
    f.write('hello, my sun!')  
finally:  
    f.close()
```

При возникновении исключения будет его обработка.

Что такое менеджер контекста ?

Менеджер контекста – это интерфейс который должен соблюдать объект для того, чтобы поддерживать инструкцию **with**. Чтобы объект функционировал как менеджер контекста нужно добавить в него методы **`__enter__`** и **`__exit__`**

`__enter__` – Python вызывает метод когда входит в контекст инструкции `with` и наступает момент получения ресурса.

`__exit__` – метод вызывается для высвобождения ресурса

Пример

```
class WritableFile:  
    def __init__(self, file_path):  
        self.file_path = file_path  
  
    def __enter__(self):  
        self.file_obj = open(self.file_path, mode="w")  
        return self.file_obj  
  
    def __exit__(self, exc_type, exc_val, exc_tb):  
        if self.file_obj:  
            self.file_obj.close()  
  
with WritableFile("hello.txt") as file:  
    file.write("Hello, World!")
```

```
import sqlite3
class DataConn:

    def __init__(self, db_name):
        """Конструктор"""
        self.db_name = db_name

    def __enter__(self):
        """
        Открываем подключение с базой данных.
        """
        self.conn = sqlite3.connect(self.db_name)
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        """
        Закрываем подключение.
        """
        self.conn.close()
        if exc_val:
            raise

with DataConn(db) as conn:
    cursor = conn.cursor()
```

Выводы

- Инструкция **with** упрощает обработку исключений путем инкапсуляции стандартных случаев применения инструкции try/finally в так называемые менеджеры контекста
- Чаще всего менеджер контекста используется для управления безопасным подключением и высвобождением системных ресурсов. Ресурсы выделяются при помощи инструкции **with** и высвобождаются автоматически , когда поток исполнения покидает **with**
- Эффективное применение инструкции **with** помогает избежать утечки ресурсов и облегчает ее восприятие.