ООП

https://rutube.ru/plst/537372/

Оглавление

1. Методы и атрибуты ▲

> **Классы и объекты. Атрибуты классов и объектов**
>
> https://rutube.ru/video/38ef3a2db2919eecf4d5e7f62a2595e1/

In [9]:
```
from string import ascii_letters
```

```
class Point():
    "Документацию класса можно поместить здесь"
    color = 'red'
    circle = 2

a = Point()
b = Point()

a.x = 1
a.y = 2

b.x = 10
b.y = 20

print(Point.__doc__) # Вывести доку на экран
print()
print(Point.__dict__) # Вывести все атрибуты класса
print()
print(f"a - {a.__dict__}") # Вывести все атрибуты объекта класса
print(f"b - {b.__dict__}") # Вывести все атрибуты объекта класса
```

```
Документацию класса можно поместить здесь

{'__module__': '__main__', '__firstlineno__': 1, '__doc__': 'Документацию класса можно поместить здесь', 'color':
'red', 'circle': 2, '__static_attributes__': (), '__dict__': <attribute '__dict__' of 'Point' objects>, '__weakref
__': <attribute '__weakref__' of 'Point' objects>}

a - {'x': 1, 'y': 2}
b - {'x': 10, 'y': 20}
```

2. Параметр `self` ▲

> **Методы классов. Параметр self**
>
> https://rutube.ru/video/408086cd61a97647d042b3f2ff148847/

In [9]:
```
class Point:
    "Учебный класс"
    color = 'red'
    circle = 2

    def set_coords(self, x=0, y=0):
        self.x = x
        self.y = y

    def get_coords(self):
        return self.x, self.y

pt = Point()
pt2 = Point()
pt.set_coords(1, 2)
pt2.set_coords(10, 20)
print(f"Координаторы полученные через метод get_cords: {pt.get_coords()}")
print(f"Координаторы полученные через метод get_cords: {pt2.get_coords()}")
```

```
Координаторы полученные через метод get_cords: (1, 2)
Координаторы полученные через метод get_cords: (10, 20)
```

3. Метод `__init__` ▲

> **Инициализатор `__init__` и финализатор `__del__`**
>
> https://rutube.ru/video/8432f8f747b092b34b4d98351b0d2331/

In [14]:
```python
class Point:
    "Учебный класс"
    color = 'red'
    circle = 2

    # __init__  метод вызывается при создании объекта класса
    def __init__(self, x=0, y=0):
        print("Вызов __init__")
        self.x = x
        self.y = y

    def set_coords(self, x, y):
        self.x = x
        self.y = y

    def get_coords(self):
        return self.x, self.y

pt = Point(1,2)
```

```
Вызов __init__
```

4. Метод `__new__` ▲

> **Магический метод `__new__`. Пример паттерна `Singleton`**
>
> https://rutube.ru/video/dca643f23343a42a20d698d8287c78c7/

- Метод `__new__` используется для реализации логики `Singleton`
- Логика `Singleton` подразумевает что в программе должен существовать только 1 экземпляр класса

In [4]:
```python
class DataBase:
    __instance = None

    def __new__(cls, *args, **kwargs):
        if cls.__instance is None:
            cls.__instance = super().__new__(cls)
        return cls.__instance

    def __del__(self):
        DataBase.__instance = None

    def __init__(self, user, psw, port):
        self.user = user
        self.psw = psw
        self.port = port

    def connect(self):
        print(f"Соединение с БД: {self.user}, {self.psw}, {self.port}")

    def close(self):
        print("Закрытие соединения с БД")

    def read(self):
        return "Данные из БД"

    def write(self, data):
        print(f"Запись в БД {data}")
```

5. Декораторы методов ▲

> **Методы класса (`classmethod`) и статические методы (`staticmethod`)**
>
> https://rutube.ru/video/935ecbc0801cfb157134e0e1d5130077/

`@classmethod` - методы класса

- Его можно вызывать напрямую в формате `Class.Method()`
- Например `Vector.validate(5)`
- Плюс нельзя работать с атрибутами экземпляра класса

`@staticmethod` - статические методы

- Они не имеют доступа ни к атрибутам класса ни к атрибутам его экземпляра.
- Создается некая независимая функция объявленная внутри класса

In [35]:
```
class Vector:
    MIN_COORD = 0
    MAX_COORD = 100

    @classmethod
    def validate(cls, arg):
        return cls.MIN_COORD <= arg <= cls.MAX_COORD

    def __init__(self, x, y):
        self.x = self.y = 0

        if self.validate(x) and self.validate(y):
            self.x = x
            self.y = y
        print(self.norm2(self.x, self.y))

    def get_coords(self):
        return self.x, self.y

    @staticmethod
    def norm2(x, y):
        return x*x + y*y
```

In [37]:
```
v = Vector(10,20)
print(Vector.norm2(5,6))
res = Vector.get_coords(v)
print(res)
```

```
500
61
(10, 20)
```

6. Инкапсуляция ▲

> **Режимы доступа `public`, `private`, `protected`. Сеттеры и геттеры**
>
> https://rutube.ru/video/6561fce4974b923af19818fe21ab4e93/

Режим доступа	Описание
`attribute` (без подчеркиваний) = `public`	Публичное свойство
`_attribute` (одно подчеркивание) = `protected`	Служит для обращения внутри класса и во всех ег одочерних классах. Но оно дишь остерегает программиста от использования этого атрибута
`__attribute` (два подчеркивания) = `private`	Служит для обращения только внутри класса. Тут мы уже не можем обратиться напрямую извне

In [5]:
```
# public
class Point():
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
pt = Point(1, 2)
pt.x = 200
```

```
pt.y = 'coords'
print(pt.x, pt.y)
```

```
200 coords
```

In [6]:
```
# protected
class Point():
    def __init__(self, x=0, y=0):
        self._x = x
        self._y = y
pt = Point(1, 2)
print(pt._x, pt._y)
```

```
1 2
```

In [16]:
```
# private
class Point():
    def __init__(self, x=0, y=0):
        self.__x = self.__y = 0
        if self.__check_value(x) and self.__check_value(y):
            self.__x = x
            self.__y = y

    # Проверяем является ли переменная числовой
    @classmethod
    def __check_value(cls, x):
        return type(x) in (int, float)

    # методы сеттеры (служат для того чтобы установить закрытые атрибуты)
    def set_coord(self, x, y):
        if self.__check_value(x) and self.__check_value(y): # применяем проверку
            # если проверка проходит присваиваем переменную
            self.__x = x
            self.__y = y
        else:
            # Если проверку не проходит выводим ошибку
            raise ValueError("Координаты должны быть числами")

    # методы геттеры (служат для того чтобы получить закрытые атрибуты)
    def get_coord(self):
        return self.__x, self.__y

pt = Point()
pt.set_coord(10, 20)
print(pt.get_coord())
```

```
(10, 20)
```

7. Магические методы ▲

> **Магические методы** `__setattr__` , `__getattribute__` , `__getattr__` , `__delattr__`
>
> https://rutube.ru/video/1b3a70fe079819d1b3cb2b0b0212f2a5/

Метод	Описание
`__setattr__(self, key, value)`	автоматически вызывается при изменении свойства (атрибута) key класса
`__getattribute__(self, item)`	автоматически вызывается при получении свойства класса с именем item
`__getattr__(self, item)`	автоматически вызывается при получении несуществующего свойства (атрибута) item класса
`__delattr__(self, item)`	автоматически вызывается при удалении свойства (атрибута) item. Неважно существует оно или нет

In [2]:
```
# Как обращаться именно к атрибутам класса

class Point():
    MIN_COORD = 0
    MAX_COORD = 100
```

```
    def __init__(self, x, y):
            self.x = x
            self.y = y

    def set_coord(self, x, y):
        if self.MIN_COORD <= x <= self.MAX_COORD:
            self.x = x
            self.y = y

    @classmethod
    def set_bound(cls, left):
        cls.MIN_COORD = left

pt1 = Point(1, 2)
pt1.set_bound(-100) # В данном примере в экземпляре класса не будет создан атрибут MIN_COORD
                    # вместо этого атрибут будет изменен в родительском классе
print(pt1.__dict__)
print(Point.__dict__)
```

```
{'x': 1, 'y': 2}
{'__module__': '__main__', '__firstlineno__': 3, 'MIN_COORD': -100, 'MAX_COORD': 100, '__init__': <function Point.
__init__ at 0x00000226527BF920>, 'set_coord': <function Point.set_coord at 0x00000226527BF560>, 'set_bound': <clas
smethod(<function Point.set_bound at 0x00000226527E82C0>)>, '__static_attributes__': ('x', 'y'), '__dict__': <attr
ibute '__dict__' of 'Point' objects>, '__weakref__': <attribute '__weakref__' of 'Point' objects>, '__doc__': Non
e}
```

In [15]:

```
# Как обращаться именно к атрибутам класса

class Point():
    MIN_COORD = 0
    MAX_COORD = 100

    def __init__(self, x, y):
            self.x = x
            self.y = y

    def set_coord(self, x, y):
        if self.MIN_COORD <= x <= self.MAX_COORD:
            self.x = x
            self.y = y

    def __getattribute__(self, item):
        print("__getattribute__")
        if item == "x":
            raise ValueError("Доступ запрещен")
        else:
            return object.__getattribute__(self, item) # Обращение к базовому классу object

    def __setattr__(self, key, value):
        print("__setattr__")
        if key == "z":
            raise AttributeError("Недопустимое имя аттрибута")
        else: object.__setattr__(self, key, value)

    def __getattr__(self, item):
        print("__getattr__ " + item)
        return False

    def __delattr__(self, item):
        print("__delattr__ " + item)
        object.__delattr__(self, item)  # Обращение к базовому классу object

pt1 = Point(1, 2)
pt2 = Point(10, 20)
a = pt1.y
print(a)
print(pt1.yy)
pt1.y = 5
```

```
__setattr__
__setattr__
__setattr__
__setattr__
__getattribute__
2
__getattribute__
__getattr__ yy
False
__setattr__
```

8. Паттерн "Моносоостояние"

> https://rutube.ru/video/751e7a7feb839e36040dae26aa76d03c/

Раскатывает атрибуты на все объекты класса и синхронизирует их.

Зачем применимо ХЗ.

In [16]:
```
class ThreadData:
    __shared_attrs = {
        'name': 'thread_1',
        'data': {},
        'id': 1
    }

    def __init__(self):
        self.__dict__ = self.__shared_attrs

th1 = ThreadData()
th2 = ThreadData()
th2.id = 3
th1.attr_new = 'new_attr'

print(th1.__dict__)
print(th2.__dict__)
```

```
{'name': 'thread_1', 'data': {}, 'id': 3, 'attr_new': 'new_attr'}
{'name': 'thread_1', 'data': {}, 'id': 3, 'attr_new': 'new_attr'}
```

9. Свойства `property`

> **Свойства property. Декоратор `@property`**
>
> https://rutube.ru/video/43c95b1abb1c2812e4588aa394d5af39/

In [18]:
```
class Person:
    def __init__(self, name, old):
        self.__name = name
        self.__old = old

    def get_old(self):
        return self.__old

    def set_old(self, old):
        self.__old = old

p = Person("Сергей", 20)
print(p.get_old())
p.set_old(35)
print(p.get_old())
```

```
20
35
```

In [26]:
```python
# Так правильно реализовать Сеттеры и Геттеры
class Person:
    def __init__(self, name, old):
        self.__name = name
        self.__old = old

    @property # реализуем геттер
    def old(self):
        return self.__old

    @old.setter
    def old(self, old):
        self.__old = old

    @old.deleter
    def old(self):
        del self.__old

p = Person("Сергей", 20)
print(p.old, p.__dict__)
p.old = 35
print(p.old, p.__dict__)
```

```
20 {'_Person__name': 'Сергей', '_Person__old': 20}
35 {'_Person__name': 'Сергей', '_Person__old': 35}
```

10. `property` практика ▲

> **Пример использования объектов `property`**
>
> https://rutube.ru/video/9a37005a96efd742eb02a069d4a5c703/

Требования:

- ФИО
- Возраст (целое число от 14 до 120)
- Серию и номер паспорта в формате `xxxx xxxxxx`, где x-цифра (от 0 до 9)
- Вес, в кг (вещественное число от 20 и выше)

Как реализовать:

- ФИО - список из трех строк
- Возраст - целое число
- Паспорт - строка в нужном формате
- Вес - вещественное число

In [12]:
```python
from string import ascii_letters

class Person:
    S_RUS = 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя'
    S_RUS_UPPER = S_RUS.upper()

    def __init__(self, fio, old, ps, weight):
        self.verify_fio(fio) # Тут нужна функция verify так как мы прогоняем данные не через сеттер
                             # Для остальных атрибутов (свойств) функция проверки вызывается в сеттере

        self.__fio = fio.split() # ФИО принимаем в таком виде, так как для него не прописан сеттер
        self.old = old
        self.passport = ps
        self.weight = weight

    # Блок проверок на правильность формата вводимых данных
    @classmethod
    def verify_fio(cls, fio):
        if type(fio) != str:
            raise TypeError("ФИО должно быть строкой")
        f = fio.split()
```

```
        if len(f) != 3:
            raise TypeError("Неверный формат ФИО")
        letters = ascii_letters + cls.S_RUS + cls.S_RUS_UPPER
        for s in f:
            if len(s) < 1:
                raise TypeError("Должен быть хотя бы один символ")
            if len(s.strip(letters)) != 0:
                raise TypeError("В ФИО можно использовать только буквенные символы и дефис")

    @classmethod
    def verify_old(cls, old):
        if type(old) != int or old < 14 or old > 120:
            raise TypeError("Возраст должен быть целым числом в диапазоне [14 - 120]")

    @classmethod
    def verify_weight(cls, w):
        if type(w) != float or w < 20:
            raise TypeError("Вес должен быть вещественным числом от 20 и выше")

    @classmethod
    def verify_ps(cls, ps):
        if type(ps) != str:
            raise TypeError("Паспорт должен быть строкой")
        s = ps.split()
        if len(s) !=2 or len(s[0]) !=4 or len(s[1]) !=6:
            raise TypeError("Неверный формат паспорта")
        for p in s:
            if not p.isdigit():
                raise TypeError("Серия и номер паспорта должна быть числами")

    # Геттер для ФИО
    @property
    def fio(self):
        return self.__fio

    # Геттер и сеттер для возраста
    @property
    def old(self):
        return self.__old
    @old.setter
    def old(self, old):
        self.verify_old(old)
        self.__old = old

    # Геттер и сеттер для веса
    @property
    def weight(self):
        return self.__weight
    @weight.setter
    def weight(self, weight):
        self.verify_weight(weight)
        self.__weight = weight

    # Геттер и сеттер для паспорта
    @property
    def passport(self):
        return self.__passport
    @passport.setter
    def passport(self, ps):
        self.verify_ps(ps)
        self.__passport = ps


p = Person('Балакирев Сергей Михайлович', 20, '1234 567890', 80.0)
p.old = 100
p.passport = '4567 123456'
p.weight = 70.0
print(p.__dict__)
print(p.old)
```

```
{'_Person__fio': ['Балакирев', 'Сергей', 'Михайлович'], '_Person__old': 100, '_Person__passport': '4567 123456',
'_Person__weight': 70.0}
100
```

11. Дескрипторы ▲

> **Дескрипторы (`data descriptor` и `non-data descriptor`)**
>
> https://rutube.ru/video/80d2bfb904b72ca981d33830b278d330/

Применяются когда надо оптимизировать дублирующий код сеттеров и геттеров

- **`non-data descriptor` (дескрипотор не данных)**

```
# class A
def __get__(self, instance, owner):
    return ...
```

- **`data descriptor` (дескрипотор данных)**

```
# class B
def __get__(self, instance, owner):
    return ...
def __set__(self, instance, value):
def __del__(self):
```

In [16]:
```
class Point3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    @classmethod
    def verify_coords(cls, coords):
        if type(coords) != int:
            raise TypeError("Координата должна быть целым числом")

    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, coord):
        self.verify_coords(coord)
        self._x = coord

    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, coord):
        self.verify_coords(coord)
        self._y = coord

    @property
    def z(self):
        return self._z
    @z.setter
    def z(self, coord):
        self.verify_coords(coord)
        self._z = coord

p = Point3D(1, 2, 3)
print(p.__dict__)
```

```
{'_x': 1, '_y': 2, '_z': 3}
```

Теперь напишем тоже самое с дескриптором

In [26]:
```
class Integer: # имя может быть любым

    @classmethod
    def verify_coord(cls, coords):
        if type(coords) != int:
            raise TypeError("Координата должна быть целым числом")
```

```
    # формирует локальное свойство (атрибут)
    # через которое потом будем обращаться в геттере и сеттере
    def __set_name__(self, owner, name):
        self.name = "_" + name

    def __get__(self, instance, owner):
        return getattr(instance, self.name) # return instance.__dict__[self.name] - еще можно так

    def __set__(self, instance, value):
        self.verify_coord(value)
        print(f"__set__:{self.name} = {value}")
        setattr(instance, self.name, value) # instance.__dict__[self.name]=value - еще можно так

class Point3D:
    # формируем дескрипторы
    x = Integer()
    y = Integer()
    z = Integer()

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

p = Point3D(1, 2, 3)
print(p.__dict__)
```

```
__set__:_x = 1
__set__:_y = 2
__set__:_z = 3
{'_x': 1, '_y': 2, '_z': 3}
```

12. `__call__` функторы и классы-декораторы ▲

> **Магический метод `__call__`**
>
> **Функторы и классы-декораторы**
>
> https://rutube.ru/video/89f14316f178c4e02fb8c854adba2d65/

Магический метод `__call__` делает класс вызываемым. То есть мы можем к нему обращаться как к функции.

Классы которые себя так ведут называются **Функторы**

Классы - функторы

In [55]:
```
class Counter:
    def __init__(self):
        self.__counter = 0
    def __call__(self, step=1, *args, **kwargs):
        print("__call__")
        self.__counter += step
        return self.__counter

c = Counter()
c2 = Counter()
c()
c(2)
res = c(10)
res2 = c2(-5)
print(res, res2)
```

```
__call__
__call__
__call__
__call__
13 -5
```

In [65]:
```python
class StripChars:
    def __init__(self, chars):
        self.__chars = chars

    def __call__(self, *args, **kwargs):
        if not isinstance(args[0], str):
            raise TypeError("Аргумент должен быть строкой")

        return args[0].strip(self.__chars)

s1 = StripChars("?:!.; ")
s2 = StripChars(" ")
res = s1("    Hello World!?    ")
res2 = s2("   Hello World!?    ")
print(res, res2, sep="\n")
```

```
Hello World
Hello World!?
```

Классы - декораторы

In [68]:
```python
import math

class Derivate:
    def __init__(self, func):
        self.__fn = func

    def __call__(self, x, dx=0.0001, *args, **kwargs):
        return (self.__fn(x + dx) - self.__fn(x)) / dx

@Derivate # Применяем декоратор
def df_sin(x):
    return math.sin(x)

# Можно еще применить вот так df_sin = Derivate(df_sin)

print(df_sin(math.pi/3))
```

```
0.4999566978958203
```

In [74]:
```python
class Logger:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        # Это то, что декоратор ДЕЛАЕТ ДОПОЛНИТЕЛЬНО
        print(f"Вызывается функция {self.func.__name__}")
        print(f"Аргументы: args={args}, kwargs={kwargs}")

        # Это сам ВЫЗОВ исходной функции
        result = self.func(*args, **kwargs)

        # Это то, что декоратор ДЕЛАЕТ ДОПОЛНИТЕЛЬНО после вызова
        print(f"Функция {self.func.__name__} вернула: {result}")
        print()  # Пустая строка для читаемости

        return result

@Logger
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

@Logger
def add(a, b):
    return a + b

# Тестируем
greet("Анна")
greet("Петр", greeting="Привет")
add(5, 3)
```

```
Вызывается функция greet
Аргументы: args=('Анна',), kwargs={}
Функция greet вернула: Hello, Анна!

Вызывается функция greet
Аргументы: args=('Петр',), kwargs={'greeting': 'Привет'}
Функция greet вернула: Привет, Петр!

Вызывается функция add
Аргументы: args=(5, 3), kwargs={}
Функция add вернула: 8
```

Out[74]: 8

In [93]:
```python
import time
import random

class Retry:
    def __init__(self, max_attempts=3, delay=1):
        self.max_attempts = max_attempts
        self.delay = delay

    def __call__(self, func):
        def wrapper(*args, **kwargs):
            for attempt in range(self.max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == self.max_attempts - 1:
                        raise e
                    print(f"Попытка {attempt + 1} не удалась: {e}\n")
                    time.sleep(self.delay)
        return wrapper

@Retry(max_attempts=5, delay=2)
def unstable_operation():
    # Может иногда падать
    rnd = random.random()
    print(f"Рандом = {rnd}")
    if rnd < 0.7:
        raise ValueError("Временная ошибка")
    return "Успех"

unstable_operation()
```

```
Рандом = 0.6246320497868814
Попытка 1 не удалась: Временная ошибка

Рандом = 0.7206077191315317
```

Out[93]: 'Успех'

13. Методы `__str__`, `__repr__`, `__len__`, `__abs__`

> **Магические методы __str__, __repr__, __len__, __abs__**
>
> https://rutube.ru/video/24ee7e7f528bbc4e5f65395c2a1853ab/

Метод	Описание
`__str__()`	Для отображения информации об объекте класса для пользователей (например `print`, `str`)
`__repr__()`	Для отображения информации об объекте класса в режиме отладки (для разработчиков)
`__len__()`	Позволяет применять функцию `len()` к экземплярам класса
`__abs__()`	Позволяет применять функцию `abs()` к экземплярам класса

In [99]:
```python
class Cat:
    def __init__(self, name):
```

```
        self.name = name

    def __repr__(self):
        return f"{self.__class__}: {self.name}"

    def __str__(self):
        return f"{self.name}"

cat = Cat("Васька")
print(cat) # Вывод информации __str__
cat        # Вывод информации __repr__
```

```
Васька
```

Out[99]: <class '__main__.Cat'>: Васька

In [106…

```
class Point:
    def __init__(self, *args):
        self.__coords = args
    def __len__(self):
        return len(self.__coords)
    def __abs__(self):
        return list(map(abs, self.__coords))
p = Point(1, -2, 3, -7)
print(len(p))
print(abs(p))
```

```
4
[1, 2, 3, 7]
```

14. Методы `__add__`, `__sub__`, `__mul__`, `__truediv__`

▲

> **Магические методы** `__add__`, `__sub__`, `__mul__`, `__truediv__`
>
> https://rutube.ru/video/5afa6b18770357d578cbbeefaa8238ef/

Метод	Описание
`__add__()`	Для операции сложения
`__sub__()`	для операции вычитания
`__mul__()`	Для операции умножения
`__truediv__()`	Для операции деления

In [121…

```
class Clock:
    __DAY = 86400 # Число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def get_time(self):
        s = self.seconds % 60
        m = (self.seconds // 60) % 60
        h = (self.seconds // 3600) % 24
        return f"{self.__get_formatted(h)}:{self.__get_formatted(m)}:{self.__get_formatted(s)}"

    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")

    def __add__(self, other):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть int или Clock")

        sc = other
```

```
        if isinstance(other, Clock):
            sc = other.seconds
        return Clock(self.seconds + sc)

    def __radd__(self, other): # Этот метод нужен чтобы был возможен такой порядок c1 = 100 + c1
        return self + other

    def __iadd__(self, other): # Этот метод нужен чтобы был возможен такой способ c1 += 100
        print("__iadd__")
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть типом int или объектом Clock")

        sc = other
        if isinstance(other, Clock):
            sc = other.seconds

        self.seconds += sc
        return self

c1 = Clock(1000)
print(c1.get_time())

c1 = c1 + 100 # Без __add__ можно записать так: c1.seconds = c1.seconds + 100
print(c1.get_time())

c2 = Clock(2000)
c3 = c1 + c2
print(c3.get_time())
```

```
00:16:40
00:18:20
00:51:40
```

15. Методы сравнений `__eq__`, `__ne__`, `__lt__`, `__gt__`

▲

> **Методы сравнений `__eq__`, `__ne__`, `__lt__`, `__gt__` и другие**
>
> https://rutube.ru/video/d7b75fc04c75cfdca04ca8021b2f7147/

Метод	Описание
`__eq__()`	Для равенства `==`
`__ne__()`	Для неравенства `!=`
`__lt__()`	Для оператора меньше `<`
`__le__()`	Для оператора меньше или равно `<=`
`__gt__()`	Для оператора больше `>`
`__ge__()`	Для оператора больше или равно `>=`

In [8]:
```
class Clock:
    __DAY = 86400 # Число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    def __eq__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return self.seconds == sc

    def __lt__(self, other):
```

```
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")

        sc = other if isinstance(other, int) else other.seconds
        return self.seconds < sc


c1 = Clock(1000)
c2 = Clock(2000)

print(c1 == c2)
print(c1 != c2)
print(c1 < c2)
```

```
False
True
True
```

В таком виде идет дублирование кода.

Чтобы избавится от этого применим следующую конструкцию:

In [18]:
```
class Clock:
    __DAY = 86400 # Число секунд в одном дне

    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY

    @classmethod
    def __verify_data(clscls, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")

        return other if isinstance(other, int) else other.seconds

    def __eq__(self, other): # Этот метод можно применять и для == и для !=
        sc = self.__verify_data(other)
        return self.seconds == sc

    def __lt__(self, other): # Этот метод можно применять и для > и для <
        sc = self.__verify_data(other)
        return self.seconds < sc

    def __le__(self, other): # Этот метод можно применять и для > и для <
        sc = self.__verify_data(other)
        return self.seconds <= sc


c1 = Clock(1000)
c2 = Clock(1000)

print(c1 == c2)
print(c1 != c2)
print(c1 < c2)
print(c1 <= c2)
```

```
True
False
False
True
```

16. Магические методы `__eq__` и `__hash__` ▲

> **Магические методы `__eq__` и `__hash__`**
>
> https://rutube.ru/video/f542fd971f0b270665f8a53f87341d40/

- Если объекты `a == b` (равны), то равен и их хеш.

- Равные хеши: `hash(a) == hash(b)` не гарантируют равенство объектов
- Если хеши не равны `hash(a) != hash(b)`, то объекты точно не равны

Хеши можно вычислять только для не изменяемых объектов: `hash(123)` | `hash("text")` | `hash((1,2,3))`

In [27]:
```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # При переопределении методов сравнения перестает стандартный механизм определения хеша
    # То есть hash(p1) выдаст ошибку "TypeError: unhashable type: 'Point'"
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __hash__(self):
        return hash((self.x, self.y))

p1 = Point(1, 2)
p2 = Point(1, 3)

print(hash(p1), hash(p2), sep="\n")
print(p1 == p2)

# Теперь мы можем формировать словарь и использовать в качестве его ключей объекты класса

d = {}
d[p1] = 1
d[p2] = 2

print(d)
```

```
-3550055125485641917
-1440771752368011620
False
{<__main__.Point object at 0x000001CD82FECC20>: 1, <__main__.Point object at 0x000001CD83046E90>: 2}
```

17. Магический метод `__bool__` ▲

> **Магический метод `__bool__` определения правдивости объектов**
>
> https://rutube.ru/video/52fd12c92124c54aa2da297968bde141/

Используется в программах, где надо описать способы проверки истинности объектов

In [38]:
```python
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        print("__len__")
        return self.x * self.x + self.y * self.y

    def __bool__(self): # Имеет приоритет и явно описывает логику работы функции bool
        print("__bool__")
        return self.x == self.y

p = Point(10, 15)
print("Классы")
print(bool(p)) # В явном виде это редко вызывается

# Обычно вызывается так
print("\n", "Вызовы bool неявно")
if p:
    print("Объект p дает True")
else:
    print("Объект p дает False")
```

```
print("\n","Стандартные")
print(bool(1))
print(bool(0))
print(bool("Hello"))
print(bool(""))
```

```
Классы
__bool__
False

 Вызовы bool неявно
__bool__
Объект p дает False

 Стандартные
True
False
True
False

 a
```

18. Магические методы `__getitem__`, `__setitem__` и `__delitem__` ▲

> **Магические методы `__getitem__`, `__setitem__` и `__delitem__`**
>
> https://rutube.ru/video/e6c3b76f225ebcf3123e4b9c5e17b70e/

- `__getitem__(self, item)` - получение значения по ключу `item`
- `__setitem__(self, key, value)` - запись значения `value` по ключу `key`
- `__delitem__(self, key)` - удаление элемента по ключу `key`

Используется чтобы атрибут экземпляра класса можно было итерировать без явного указания например вместо, писать:

- `s1.marks[2]`
- `s1[2]`

In [32]:
```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = list(marks)

    def __getitem__(self, item):
        if 0 <= item < len(self.marks):
            return self.marks[item]
        else:
            raise IndexError("Неверный индекс")

    def __setitem__(self, key, value):
        if not isinstance(key, int) or key < 0:
            raise TypeError("Индекс должен быть целым не отрицательным числом")
        if key >= len(self.marks):
            off = key + 1 - len(self.marks)
            self.marks.extend([None] * off)

        self.marks[key] = value

    def __delitem__(self, key):
        if not isinstance(key, int):
            raise TypeError("Индекс должен быть целым числом")
        del self.marks[key]

s1 = Student("Сергей", [5,4,3,2,1])
print(s1.marks)
print(s1.marks[2])
print(s1[2])
```

```
# Чтобы менять значения внутри списка надо реализовать магический метод
# __setitem__

s1[2] = 0
s1[10] = 1
print(s1.marks)
print(s1[2])

del s1[2]
print(s1.marks)
```

```
[5, 4, 3, 2, 1]
3
3
[5, 4, 0, 2, 1, None, None, None, None, None, 1]
0
[5, 4, 2, 1, None, None, None, None, None, 1]
```

19. Магические методы `__iter__` и `__next__` ▲

> **Магические методы `__iter__` и `__next__`**
>
> https://rutube.ru/video/0fbc1917924a02247bd33f99c9498d80/

- `__iter__(self)` - получение итератора для перебора объекта
- `__next__(self)` - переход к следующему значению и его считываение

In [55]:
```
print(list(range(5)))

a = iter(range(5)) # Создаем итератор
# Вызываем следующее значение итерируемого объекта
print(next(a), next(a), next(a), next(a), next(a))
print()

class FRange:
    def __init__(self, start=0.0, stop=0.0, step=1.0):
        self.start = start
        self.stop = stop
        self.step = step
        self.value = self.start - self.step

    def __iter__(self):
        self.value = self.start - self.step
        return self

    def __next__(self):
        if self.value + self.step < self.stop:
            self.value += self.step
            return self.value
        else:
            raise StopIteration

fr = FRange(0, 2, 0.5)

# тут функция next вызывает магический метод __next__ из класса
# а сам объект FRange выступает в роли итератора, но пока не прописано явно метод __iter__
# итерировать его нельзя, то есть for i in fr: даст ошибку TypeError: 'FRange' object is not iterable
print("Используем __next__")
print(next(fr), next(fr), next(fr), next(fr))

print()
print("Используем __iter__")
for i in fr:
    print(i)

print()
print("Класс двумерный итератор")
class FRange2D:
    def __init__(self, start=0.0, stop=0.0, step=1.0, rows=5):
        self.rows = rows
```

```
        self.fr = FRange(start, stop, step)

    def __iter__(self):
        self.value = 0
        return self

    def __next__(self):
        if self.value < self.rows:
            self.value += 1
            return iter(self.fr)
        else:
            raise StopIteration

fr2 = FRange2D(0,2,0.5,4)

for row in fr2:
    for x in row:
        print(x, end=" ")
    print()
```

```
[0, 1, 2, 3, 4]
0 1 2 3 4

Используем __next__
0.0 0.5 1.0 1.5

Используем __iter__
0.0
0.5
1.0
1.5

Класс двумерный итератор
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
0.0 0.5 1.0 1.5
```

In [ ]:

20. Наследование в объектно-ориентированном программировании ▲

> **Наследование в объектно-ориентированном программировании**
>
> https://rutube.ru/video/c52f01e60ac617d23888d8175115b3f8/

Наследование используется для оптимизации кода. Чтобы не дублировать одинаковые блоки кода. Например установка координат для разных геометрических фигур.

In [81]:
```
# Базовый или родительский класс
class Geom:
    name = 'Geom'

    def set_coords(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

    def draw(self):
        print("Рисование примитива")

# Подкласс или дочерний класс
class Line(Geom): # в скобках определяем наследование
    name = 'Line' # при обращении к атрибутам, сначала они ищутся в локальных атрибутах, а потом в базовом классе
    # Такая конструкция называется переопределением атрибута. В IDE это отображается синим значком.
    def draw(self):
        print("Рисование линии")
```

```
# Подкласс или дочерний класс
class Rect(Geom): # в скобках определяем наследование
    pass
    # def draw(self):
    #     print("Рисование прямоугольника")

g = Geom()
l = Line()
r = Rect()

l.set_coords(1,1,2,2)
r.set_coords(10,10,20,20)

print(l.__dict__)
print(r.__dict__)

print("Вывод из класса Geom:", g.name)
print("Вывод из класса Line:", l.name) # так как атрибут name определен в классе Line, будет выведен он
print("Вывод из класса Rect:", r.name)

l.draw()
r.draw() # метода draw в классе Rect нет, по этому поиск пойдет в базовом классе
g.draw()
```

```
{'x1': 1, 'y1': 1, 'x2': 2, 'y2': 2}
{'x1': 10, 'y1': 10, 'x2': 20, 'y2': 20}
Вывод из класса Geom: Geom
Вывод из класса Line: Line
Вывод из класса Rect: Geom
Рисование линии
Рисование примитива
Рисование примитива
```

21. Функция `issubclass()` ▲

> **Функция `issubclass()`. Наследование от встроенных типов и от `object`**
>
> https://rutube.ru/video/6929a9e932349cd2d5c6a810929a55d5/

Все стандартные объекты Python, типа `int`, `str`, `list` и т.п являются классами.

А значит мы можем их переопределять в своих классах. Такое используется редко. Этот урок больше для понимания работы.

In [15]:
```
class Geom:
    pass

class Line(Geom):
    pass

g = Geom()
l = Line()

print(g)
print(l)

# Функция issubclass(class, class) работает только напрямую с классами, а не с их экземплярами
print(issubclass(Line, Geom)) # показывает что класс Line наследуется от класса Geom
print(isinstance(l, Geom)) # показывает наследование от экземпляра класса

print(issubclass(int, object))
print()

class Vector(list):
    def __str__(self):
        return " ".join(map(str, self))

v = Vector([1,2,3])
```

```
print([1,2,3])
print(v)
```

```
<__main__.Geom object at 0x000001EFD985D940>
<__main__.Line object at 0x000001EFD985C440>
True
True
True

[1, 2, 3]
1 2 3
```

22. Наследование. Функция `super()` и делегирование ▲

> **Наследование. Функция `super()` и делегирование**
>
> https://rutube.ru/video/f9b037e2e4ec9963b052b5d52735fd61/

In [25]:
```
class Geom:
    name = 'Geom'

    def __init__(self, x1, y1, x2, y2):
        print(f"Инициализатор Geom для {self.__class__}")
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

class Line(Geom):
    def draw(self):
        # Здесь не нужна функция super() так как этот класс полностью наследует метод __init__ от базового класса
        print("Рисование линии")


class Rect(Geom):
    def __init__(self, x1, y1, x2, y2, fill=None):
        super().__init__(x1, y1, x2, y2) # Функция возвращает ссылку на объект посредник,
                                         # который вызывает базовый (родительский) класс
        print("Инициализатор Rect")
        self.fill = fill


    def draw(self):
        print("Рисование прямоугольника")

l = Line(0,0,10,20)
r = Rect(1,2,3,4)

print(r.__dict__)
```

```
Инициализатор Geom для <class '__main__.Line'>
Инициализатор Geom для <class '__main__.Rect'>
Инициализатор Rect
{'x1': 1, 'y1': 2, 'x2': 3, 'y2': 4, 'fill': None}
```

23. Наследование. Атрибуты `private` и `protected` ▲

> **Наследование. Атрибуты `private` и `protected`**
>
> https://rutube.ru/video/634396c634e953e5a02ce885a1c334dd/

- `attribute` (без подчеркиваний вначале) - публичное свойство `public`
- `_attribute` (с одним подчеркиванием) - режим доступа `protected` служит для обращения внутри класса и во всех его дочерних классах

- `__attribute` (с двумя подчеркиваниями) - публичное свойство `private` служит для обращения только внутри класса

In [5]:
```
class Geom:
    name = 'Geom'

    def __init__(self, x1, y1, x2, y2):
        print(f"Инициализатор Geom для {self.__class__}")
        self.__x1 = x1
        self.__y1 = y1 # при использовании подчеркиваний перед атрибутом,
        self.__x2 = x2 # они будут доступны только внутри этого класса
        self.__y2 = y2 # их нельзя вызвать в дочерних классах

    # А вот тут будет прекрасно работать ###################################################
    def get_coords(self):                                                                  #
        return (self.__x1, self.__y1)                                                      #
                                                                                           #
                                                                                           #
class Rect(Geom):                                                                          #
    def __init__(self, x1, y1, x2, y2, fill=None):                                         #
        super().__init__(x1, y1, x2, y2) # Функция возвращает ссылку на объект посредник,  #
        self.__fill = fill                # который вызывает базовый (родительский) класс  #
                                                                                           #
    # Вот тут код упадет ###################################################################
    # def get_coords(self):
    #     return (self.__x1, self.__y1)

r = Rect(1,2,3,4)
print(r.__dict__)
r.get_coords()
```

```
Инициализатор Geom для <class '__main__.Rect'>
{'_Geom__x1': 1, '_Geom__y1': 2, '_Geom__x2': 3, '_Geom__y2': 4, '_Rect__fill': None}
```

Out[5]: (1, 2)

In [7]:
```
class Geom:
    name = 'Geom'

    def __init__(self, x1, y1, x2, y2):
        print(f"Инициализатор Geom для {self.__class__}")
        self._x1 = x1
        self._y1 = y1
        self._x2 = x2
        self._y2 = y2

class Rect(Geom):
    def __init__(self, x1, y1, x2, y2, fill=None):
        super().__init__(x1, y1, x2, y2) # Функция возвращает ссылку на объект посредник,
        self._fill = fill                # который вызывает базовый (родительский) класс

    def get_coords(self):
        return (self._x1, self._y1)

r = Rect(1,2,3,4)
print(r.__dict__)
r.get_coords()
```

```
Инициализатор Geom для <class '__main__.Rect'>
{'_x1': 1, '_y1': 2, '_x2': 3, '_y2': 4, '_fill': None}
```

Out[7]: (1, 2)

24. Полиморфизм и абстрактные методы ▲

> **Полиморфизм и абстрактные методы**
>
> https://rutube.ru/video/00d06afb1f1ea155cfa3996a1f063f02/

Полиморфизм - это возможность работы с совершенно разными объектами (языка Python) единым образом

In [17]:
```python
class Rectangle:
    def __init__(self, w, h):
        self.w = w
        self.h = h
    def get_rect_pr(self):
        return 2*(self.w + self.h)

class Square:
    def __init__(self, a):
        self.a = a
    def get_sq_pr(self):
        return 4 * self.a

class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def get_tr_pr(self):
        return self.a + self.b + self.c

r1 = Rectangle(1,2)
r2 = Rectangle(3,4)
s1 = Square(10)
s2 = Square(20)
# t1 = Triangle(1,2,3)
# t2 = Triangle(4,5,6)

geom = [r1, r2, s1, s2] #, t1, t2]

# Решение в лоб
for g in geom:
    if isinstance(g, Rectangle):
        print(g.get_rect_pr())
    else:
        print(g.get_sq_pr())
```

```
6
14
40
80
```

Теперь применим **Полиморфизм**

In [24]:
```python
# Если мы создадим базовый класс Geom с методом get_pr()
# и генератором ошибки NotImplementedError
# то будет контролироваться целостность метода во всех дочерних классах
class Geom:
    def get_pr(self):
        raise NotImplementedError("В дочернем классе должен быть переопределен метод get_pr()")

class Rectangle(Geom):
    def __init__(self, w, h):
        self.w = w
        self.h = h
    def get_pr(self):
        return 2*(self.w + self.h)

class Square(Geom):
    def __init__(self, a):
        self.a = a
    def get_pr(self):
        return 4 * self.a

class Triangle(Geom):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def get_pr(self):
        return self.a + self.b + self.c
```

```
geom = [Rectangle(1,2), Square(10), Triangle(1,2,3),
        Rectangle(3,4), Square(20), Triangle(4,5,6)]

for g in geom:
    print(g.get_pr())
```

```
6
40
6
14
80
15
```

25. Множественное наследование ▲

> **Множественное наследование**
>
> https://rutube.ru/video/fd9b6a2924296f186aebd3c65ece2f6d/

In [37]:
```
class Goods:
    def __init__(self, name, weight, price):
        super().__init__() # Запускапет метод __init__ для следующего базового класса в MRO списке
        print("init Goods")
        self.name = name
        self.weight = weight
        self.price = price

    def print_info(self):
        print(f"{self.name}, {self.weight}, {self.price}")

class MixinLog:
    ID = 0

    def __init__(self):
        print("init MixinLog")
        self.ID += 1
        self.id = self.ID

    def save_sell_log(self):
        print(f"{self.id}: товар был продан в 00:00 часов")

class NoteBook(Goods, MixinLog): # Прописываем наследование от Goods и от MixinLog
    pass

n = NoteBook('Acer', 1.5, 30_000)
n.print_info()
n.save_sell_log()

print(NoteBook.__mro__)
```

```
init MixinLog
init Goods
Acer, 1.5, 30000
1: товар был продан в 00:00 часов
(<class '__main__.NoteBook'>, <class '__main__.Goods'>, <class '__main__.MixinLog'>, <class 'object'>)
```

26. Коллекция `__slots__` ▲

> **Коллекция `__slots__`**
>
> https://rutube.ru/video/92f1421eb1104fad53e321b4e3e2b045/

Коллекция `__slots__` используется для:

- ограничения создаваемых локальных свойств
- уменьшения занимаемой памяти

- ускорение работы с локальными свойствами

In [46]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Point2D:
    __slots__ = ('x', 'y') # указывает на то, что в объекте класса Point2D могут присутствовать только локальные
    def __init__(self, x, y):
        self.x = x
        self.y = y

pt = Point(1, 2)
print(pt.__dict__)

pt.z = 40
print(pt.__dict__)


pt2 = Point2D(10, 20)
print(pt2.x)
print(pt2.y)
```

```
{'x': 1, 'y': 2}
{'x': 1, 'y': 2, 'z': 40}
10
20
```

27. Как работает `__slots__` с `property` и при наследовании ▲

> **Как работает `__slots__` с `property` и при наследовании**
>
> https://rutube.ru/video/7c7ceb3948c9c027827bdec2c3b44d6b/

28. ▲

> https://

29. ▲

> https://

30. ▲

> https://

- ▲

> https://