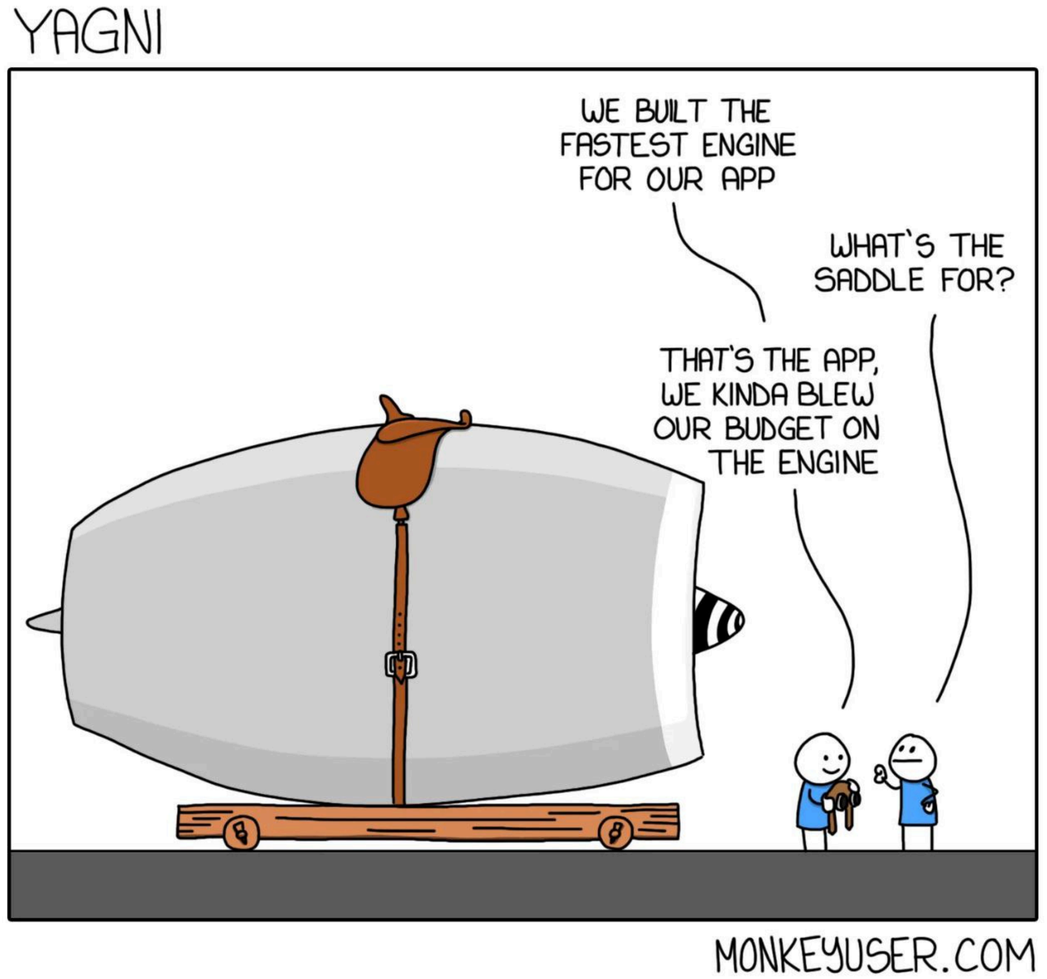
ООП


YAGNI
WE BUILT THE FASTEST ENGINE FOR OUR APP
WHAT'S THE SADDLE FOR?
THAT'S THE APP, WE KINDA BLEW OUR BUDGET ON THE ENGINE
MONKEYUSER.COM

Тёма

План семинаров на этот модуль

Всего семинаров 5

- ООП-1
- ООП-2
- Регулярки
- хз...
- хз.....

ЗОЧЕМ??7

- Посмотрите на это "творчество"
- Красиво ли это выглядит?
- Чего плохого в этом коде **стилистически**?
- Бонус: попробуйте за полминуты найти тут ошибку

```python
def gpa(student):
    return sum(student["grades"].values()) / len(student["grades"])

students = dict()

students["Nadya"]["age"] = {
    "age": 20,
    "grades": {
        "math": 10,
        "economics": 9,
    }
}
students["Nadya"]["faculty"] = "FES"

students["Tema"]["age"] = {
    "age": 22,
    "grades": {
        "math": 7,
        "programming": 10,
    },
    "faculty": "FCS",
}
```

А что если...

- Я хочу вывести какую-то информацию по студенту?
- Сделать проверку, что возраст студента не -1 год?
- Отчислить?

```
In [6]: print(students["Nadya"])
{'age': {'age': 20, 'grades': {'math': 10, 'economics': 9}}, 'faculty': 'FES'}

In [7]:
```

```
if students["Nadya"]["age"] < 16:
    raise ValueError("Should not be less than some threshold")
```

```
def expel(student):
    student["faculty"] = None
```

А как иначе?

```python
class Student:
    def __init__(self, name: str, faculty: str):
        self.name = name
        self.faculty = faculty
        self._grades = {}

    def gpa(self):
        return sum(self._grades.values()) / len(self._grades)

    def set_grade(self, subject, grade):
        self._grades[subject] = grade

    def __str__(self):
        return f"Student {self.name} from HSE {self.faculty}"
```

Извините за разные подсветки,
я забыл заскриншотить в VS Code :(((

Тогда и...

- Я хочу вывести какую-то информацию по студенту
- Сделать проверку, что возраст студента не -1 год
- Отчислить

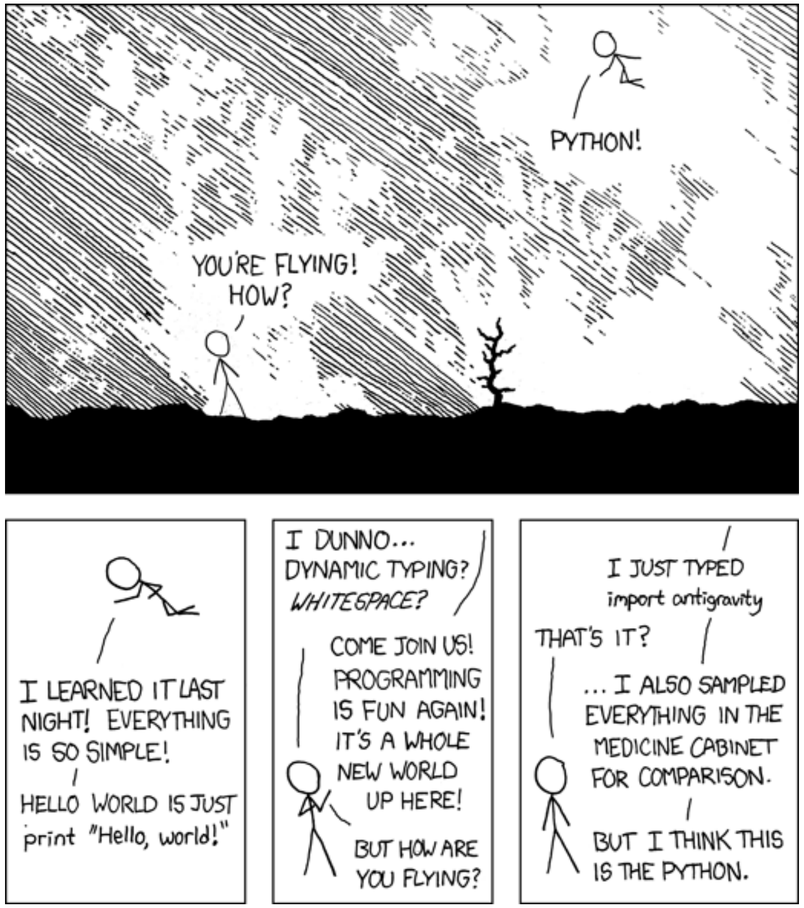
```
In [2]: nadya = Student("Nadya", "FES")

In [3]: print(nadya)
Student Nadya from HSE FES

In [4]:
```

```
class Student:
    def __init__(self, name: str, faculty: str):, age: int
        self.name = name
        self.faculty = faculty
        if age < 16:
            raise ValueError("Age cannot be less than some threshold")
        self.age = age
```

```
    def expel(self):
        self.faculty = None
```

У ООП только 4 союзника:

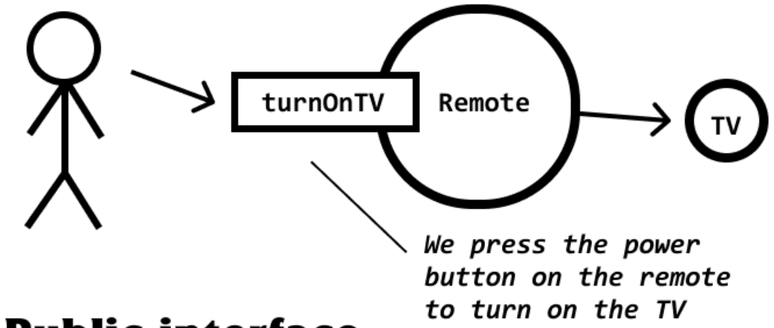
- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм



пока я болтаю,
напишите в питоне import antigravity

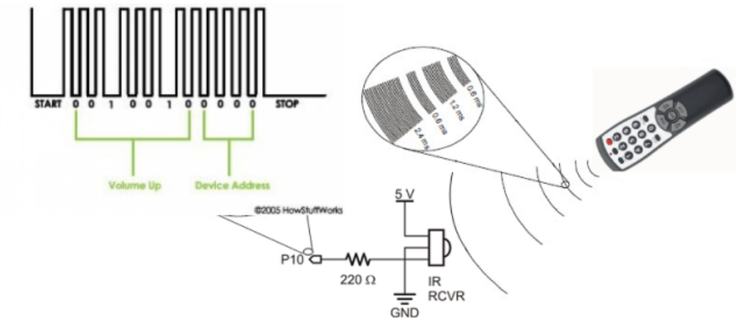
Абстракция

Abstraction example: Using a remote control to turn on a TV



Public interface

Public
Declarative
Human-centered
Highest level of abstraction



Implementation

Private
Imperative
More technical
Lower levels of abstraction

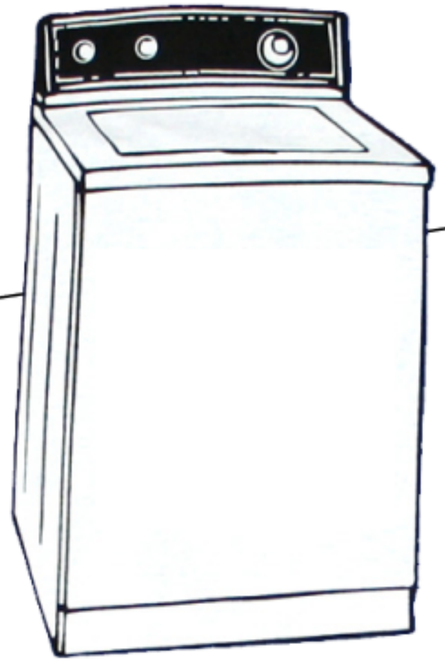
Абстракция

на примере какой-то древней стиралки

Implementation

```
startCycle (options) {
  // Parse the options
  // Get access to the physical layer
  // Convert options into commands
  // Lots of low-level code
  // And so on...
}
```

Implementation complexities abstracted away and understood only by developers and domain experts.



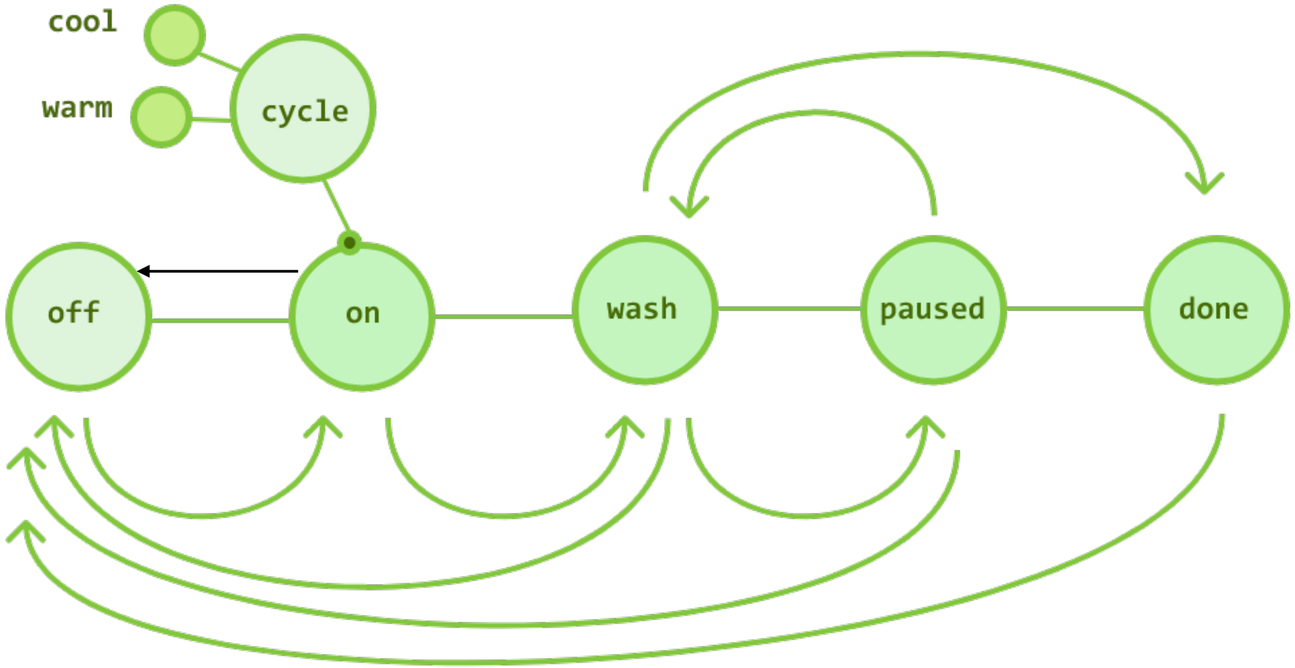
Public interface

```
startCycle(options)
```

Simple and easy for human beings to understand

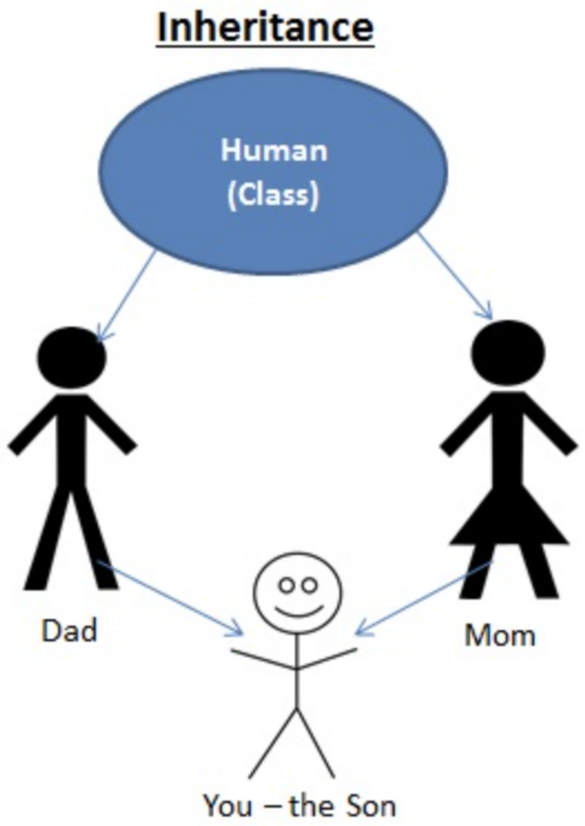
Инкапсуляция

все еще на примере стиралки лол



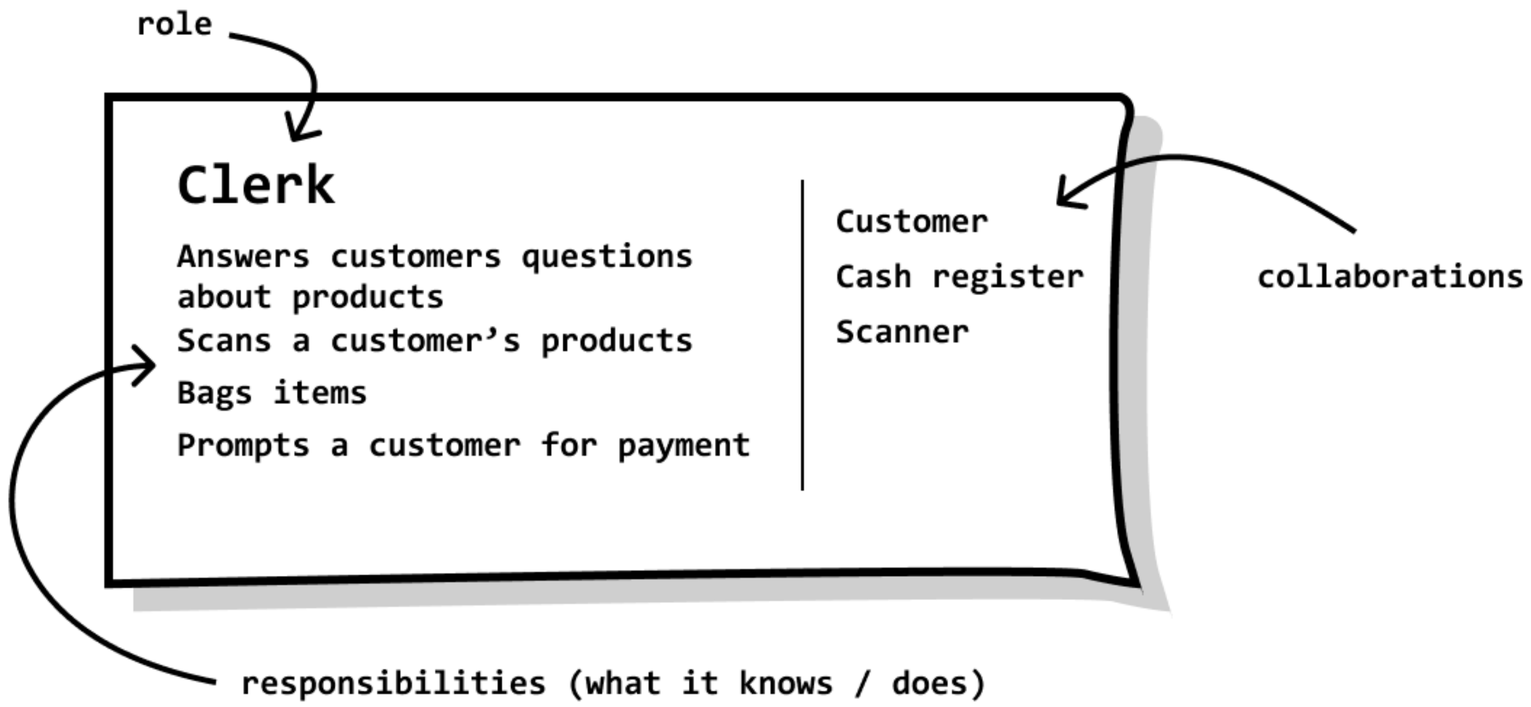
Наследование

~~на примере книжки "как я появился на свет"~~



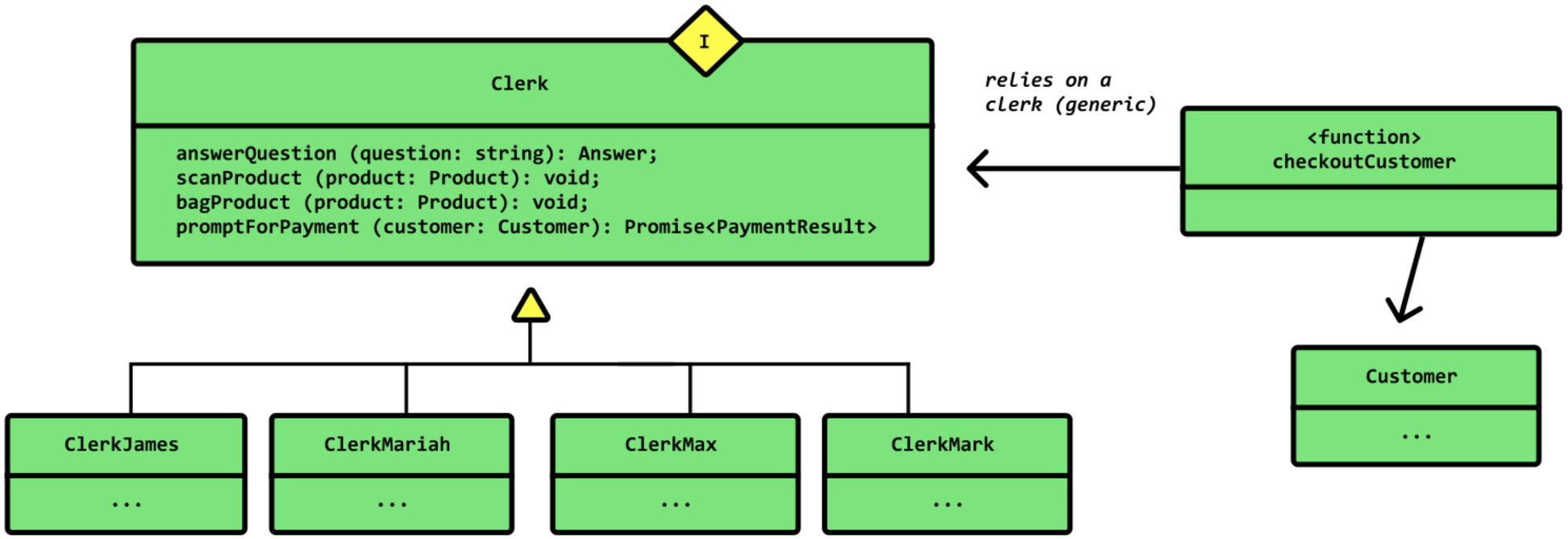
Полиморфизм

И не реклама фильма "Клерки" (но очень рекомендую)



Полиморфизм

Главное "Клерки 2" не смотрите...



```
first = Student("Tema", "Computer science")
print(first)
```

Как работает ООП в Python?

```python
class Student:
    def __init__(self, name: str, faculty: str):
        self.name = name
        self.faculty = faculty
        self._grades = {}

    def gpa(self):
        return sum(self._grades.values()) / len(self._grades)

    def set_grade(self, subject, grade):
        self._grades[subject] = grade

    def __str__(self):
        return f"Student {self.name} from HSE {self.faculty}"
```

Это типизация.
Ее можно не писать, но
когда будем проходить dataclass,
то это будет полезно)

__init__ -- это специальная
функция конструктора,
говорит, как будет собираться наш экземпляр.
Первый аргумент (обычно self) -- особенный
и ссылается на текущий объект.

В остальные функции тоже **обычно**
надо передавать self, чтобы получить
поля объекта.

__str__ это magic-функция, так же как и __init__.
Задает, как будет выглядеть ваш объект, когда вы
сделаете str(obj) или print(obj)

Что еще есть в классах?

Специальные методы, переопределяющие операторы

```
p1 = Point(1, 2)
p2 = Point(3, 4)

p3 = p1 + p2
```

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

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)
```

Атрибуты экземпляра и класса

```
class A:
    for i in range(10):
        print(i)
```

```
In [12]: class Student:
    ...:     presence = {}
    ...:
    ...:     def __init__(self, name):
    ...:         self.name = name
    ...:         self.presence[name] = "present"
    ...:

In [13]: s1 = Student("Artem")

In [14]: s1.presence
Out[14]: {'Artem': 'present'}

In [15]: Student.presence
Out[15]: {'Artem': 'present'}

In [16]: s2 = Student("Tema")

In [17]: s2.presence
Out[17]: {'Artem': 'present', 'Tema': 'present'}

In [18]: Student.presence
Out[18]: {'Artem': 'present', 'Tema': 'present'}
```

```
class Student:
    MAX_NAME_LENGTH = 255

    def __init__(self, name):
        if len(name) < self.MAX_NAME_LENGTH:
            raise ValueError
```

HO!

```
In [1]: class Student:
   ...:     global_number = 1
   ...:
   ...:     def __init__(self, name):
   ...:         self.name = name
   ...:         self.number = self.global_number
   ...:         self.global_number += 1
   ...:
   ...:

In [2]: s1 = Student("Artem")

In [3]: s2 = Student("Tema")

In [4]: s1.global_number
Out[4]: 2

In [5]: s1.number
Out[5]: 1

In [6]: s2.number
Out[6]: 1
```

Договоренность

- self.some_field -- публичное поле
- self._some_field -- protected, его также можно достать из объекта, но нежелательно
- self.__some_field -- приватное поле. Просто так достать его нельзя (но вообще, можно)

```
In [6]: class A:
   ...:     def __init__(self):
   ...:         self.public = 1
   ...:         self._protected = 2
   ...:         self.__private = 3
   ...:

In [7]: a = A()

In [8]: print(a.public)
1

In [9]: print(a._protected)
2

In [10]: print(a.__private)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [10], line 1
----> 1 print(a.__private)

AttributeError: 'A' object has no attribute '__private'

In [11]: print(a.__dict__)
{'public': 1, '_protected': 2, '_A__private': 3}

In [12]: a._A__private
Out[12]: 3
```

Python -- это все объекты

- a = A() -- это экземпляр класса. Он лежит где-то в памяти и является некоторым объектом
- При этом забавно, что сам class A -- это тоже объект (а вот объект чего -- это достойно отдельной лекции)
- Совсем мозговыносящее: type -- это тоже объект и type(type) равен type

```
In [13]: class A:
    ...:     ...
    ...:

In [14]: a = A()

In [15]: type(A)
Out[15]: type

In [16]: type(a)
Out[16]: __main__.A

In [17]: id(A)
Out[17]: 5857054848

In [18]: id(a)
Out[18]: 4372510944

In [19]: id(1)
Out[19]: 4336500976
```

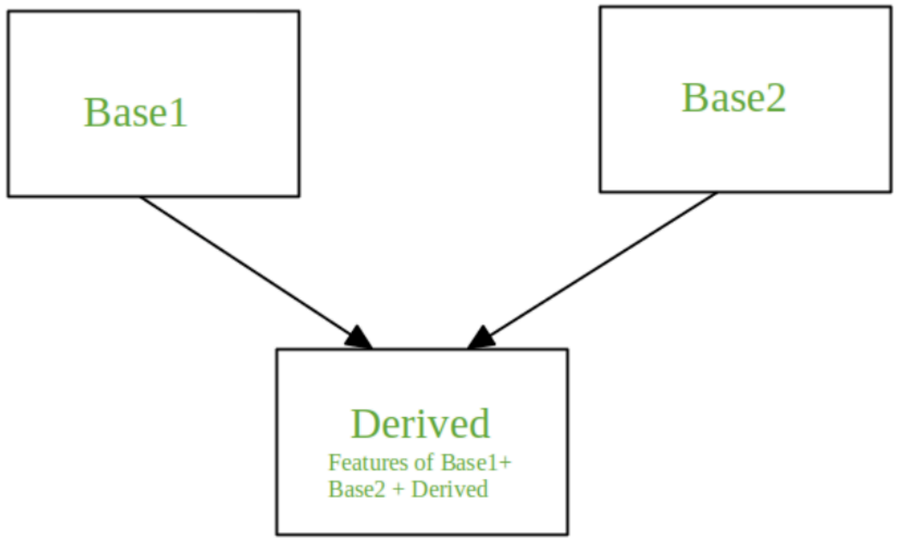
Inheritance

```
In [12]: class Employee:
    ...:     def __init__(self, name: str):
    ...:         self.name = name
    ...:
    ...:     def work(self):
    ...:         print("working hard")
    ...:

In [13]: class OutstaffEmployee(Employee):
    ...:     def __init__(self, name: str, salary_type: str):
    ...:         # вызываем родительский конструктор
    ...:         super().__init__(name)
    ...:         # добавляем какое-нибудь новое поле
    ...:         self.salary_type = salary_type
    ...:

In [14]: emp1 = Employee("Vasya")

In [15]: emp2 = OutstaffEmployee("Vasya", "piercework_payment")

In [16]: emp1.work()
working hard

In [17]: emp2.work()
working hard
```

```
class Hound(Dog):
    def bark(self):
        raise NotImplementedError()
```

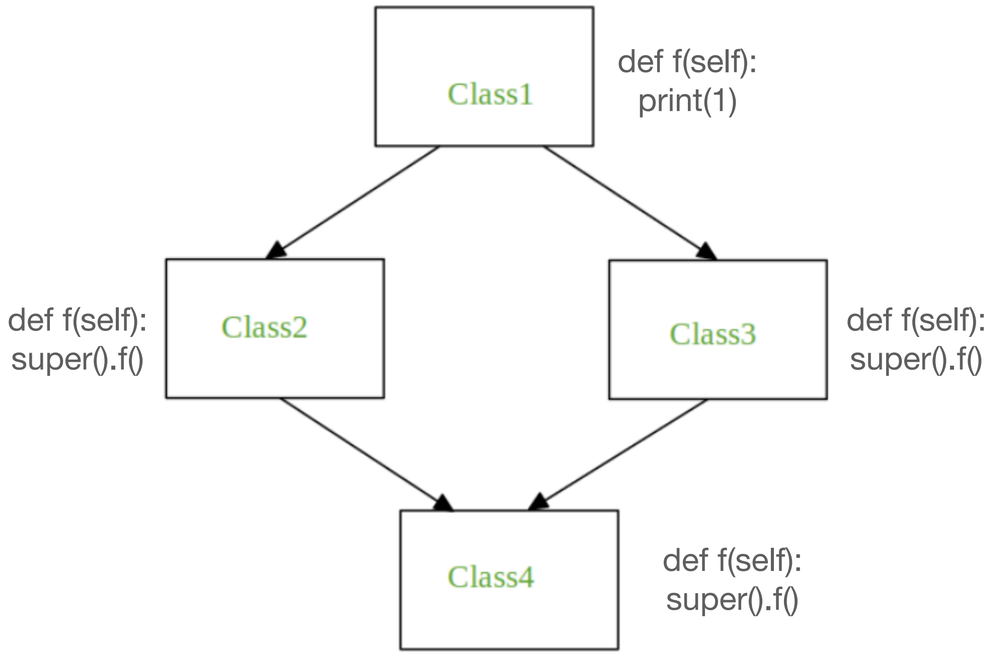
Multi-inheritance

```
class A:
    ...

class B:
    ...

class C(A, B):
    ...
```



Diamond inheritance problem

class 4 -> class 2 -> class 3 -> class 1



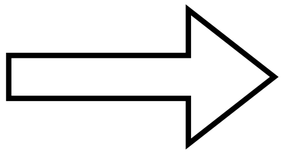
Diamond inheritance problem

Зачем питону нужен __mro__

- __mro__ (method resolution order) -- это способ зафиксировать, в каком порядке у класса/родителей будет вызываться метод
- Например, у класса есть функция f, тогда она вызовется у него
- Если этой функции f нет у класса, но есть у родителя, то вызовется у него
- А если родителей несколько, что будет делать super()?

Diamond inheritance problem

```
In [3]: class A:
   ...:     def display(self):
   ...:         print("This is class A")
   ...:
   ...: class B(A):
   ...:     ...  # тут нет def display
   ...:
   ...: class C(A):
   ...:     def display(self):
   ...:         print("This is class C")
   ...:
   ...: class D(B, C):
   ...:     pass
   ...:
   ...: obj = D()
   ...: obj.display()
This is class C
```



```
In [2]: D.__mro__
Out[2]: (__main__.D, __main__.B, __main__.C, __main__.A, object)

In [3]:
```

Dataclasses