Классы.

Определение класса

```
# минимально возможное определение класса
class A:
    pass

# создание объекта класса
# как видно, всегда есть "конструктор" по-умолчанию
a = A()
```

Определение класса

```
class C:
    # первым параметром передается self, т.е. сам объект;
    # нет неявного this; объект либо передан как параметр,
    # либо нет
    # все, что мы делаем в __init__ -- присваиваем аттрибуты
    # в уже существующий объект
    # имя self -- соглашение, можно называть как угодно
    def __init__(self, a, b):
        self.a, self.b = a, b

# создание экземпляра класса:
c = C(10, 20)

# обращение к атрибутам
print(c.a, c.b)
# 10 20
```

Определение класса

```
class SomeClass:
    # атрибуты присваиваются именно объекту
    # класса, а не самому классу
    def __init__(self, a, b):
        self.a, self.b = a, b

    # классу также можно присвоить атрибут
    a = 30

a = SomeClass(10, 20)
print(a.a)
# 10
print(SomeClass.a)
# 30
```

Определение класса

Тело класса вычисляется как выражение, поэтому возможен такой код:

```
class B:
    n = 10
    for i in range(10):
        n += i
    print(n) # в какой момент отработает этот код?

#?
```

Модификаторы доступа

Если коротко, то их нет.

```
class A:
    def __init__(self, a, b):
        # если подробнее, то можем с помощью одного _ в
        # начале имени сказать другим программистам, что
        # атрибут предназначен для внутреннего использования,
        # но это не больше, чем соглашение
        self._a = a  # РЕБЯТА, ПОЖАЛУЙСТА, НЕ ИСПОЛЬЗУЙТЕ _a
СНАРУЖИ

        # с помощью двух _, т.е. __ в начале имени можем
        # сказать интерпретатору: можно и манглингинг,
        # мне уже ничего не поможет
        self.__b = b
```

Модификаторы доступа

```
a = A(10, 20)

# _a остается доступен
print(a._a)
# 10

# __b больше недоступен
print(a.__b)
# ---------------
# AttributeError   Traceback (most recent call last)
# ----> 1 print(a.__b)
#
# AttributeError: 'A' object has no attribute '__b'

# но зато теперь доступен _A__b; то, что сделал с именем
# интерпретатор, называется __манглинг имен__
print(a._A__b)
```

Декоратор property

Мы поняли, что инкапсуляцией нигде и не пахнет. Но как можем скрыть что-то от пользователей?

Можем добавить в начало названия атрибута два нижних подчеркивания: __a и написать get и set, но в python вместо этого принято использовать декоратор **@property**

Для любознательных: property на самом деле создает дескриптор

Декоратор property

```
class A:
    def __init__(self, a):
        self.__foo = a

    @property
    def foo(self):
        print("Called foo getter")
        return self.__foo

    # если хочется разрешить присваивание, то
    # @foo.setter
    # def foo(self, value):
    #     print("Called foo setter")
    #     self.__foo = value

    # аналогично с @foo.deleter для удаления
```

Декоратор property

```
a.foo
# Called foo getter
# 5

a.foo = 50
# ------------
# AttributeError   Traceback (most recent call last)
# ----> 1 a.foo = 50
#
# AttributeError: can't set attribute
```

Магические методы

> *If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck.*

Утиная типизация - вид динамической типизации, когда границы использования объекта определяются его текущим набором методов и свойств

Python — язык с утиной типизацией и, как правило, то, каким образом объект может быть использован, определяется магическими (или dunder) методами.

Именуются они всегда одинаково: `__method_name__` (__ в начале и конце имени)

Магические методы

Аккуратно оформленный блогпост со списком существующих магических методов здесь. Возможно, список не полный и/или уже устарел или когда-то устареет.

Поэтому за актуальным списком лучше обратиться к менее читаемой, но правдивой документации.

Поиск атрибутов

```
class SomeClass:
    def __init__(self, a, b):
        self.a, self.b = a, b

a = SomeClass(10, 20)
# объект можно менять и после создания
a.c = -1
print(a.c)
# -1
print(SomeClass.c) # такого атрибута у класса нет
# ---> 15 print(SomeClass.c)
# AttributeError: type object 'SomeClass' has no attribute 'c'
```

Поиск атибутов

```
# попробуем наоборот
SomeClass.e = 30

print(a.e)
# 30

print(SomeClass.e) # так все работает, почему?
# 30
```

Поиск атрибутов

Для того, чтобы понять почему так происходит, нужно знать две вещи.

Первое — у каждого объекта есть атрибут __dict__, в котором содержатся атрибуты объекта. У класса свой __dict__, отличный от того, что у объекта.

На самом деле не у всех, т.к. еще есть __slots__.

Поиск атрибутов

Второе — интерпретатор ищет аттрибут `attr` в следующем порядке:

1. Зовем obj.__class__.__getattribute__ передавая наш объект, по умолчанию он проверяет
   a. Есть ли `attr` в obj.__dict__? Если есть, вернем
   b. Есть ли `attr` в obj.__class__.__dict__? Если есть, вернем
2. Если поиск неуспешный, зовем __getattr__

Присваиванее аналогично, но с __setattr__

Все, что написано выше сильное упрощение и привирание. Позже поймем почему

Методы

Методы тоже являются атрибутами

```
class A:
    def foo(self):
        return 42

print(A.foo) # <function A.foo at 0x10322bea0>
a = A()

print(a.foo) # <bound method A.foo of <__main__.A object at
0x10316d208>>
# Как видим, в случае с методами атрибут достается не совсем
# по-честному, но связывается с объектом, у которого его взяли
# И это тоже реализовано с помощью дескрипторов :)
#
# a.foo() == A.foo(a)
```

Наследование

```
class Base: # тоже самое, что и class Base(object):
    def foo(self):
        print("Base")

class A(Base):
    def foo(self):
        # super() ищет ближайшего родственника в порядке mro
        # super() ~ super(A, self)
        #
        # можно в качестве первого аргумента передать
        # что-нибудь другое: self(C, self)
        # предлагается выяснить самостоятельно
        # что будет в этом случае
        print("A", end=" ")
        super().foo()
```

Наследование

```
class B(Base):
    def foo(self):
        print("B", end=" ")
        super().foo()


class C(A, B):
    def foo(self):
        print("C", end=" ")
        super().foo()
```

Наследование

```
# в каком порядке все будет выведено?
C().foo()

# в таком, в каком определяет method resolution order:
C A B Base

# несмотря на название, MRO определят порядок и для разрешения
# других атрибутов
# посмотреть порядок для конкретного класса можно так:
print(C.mro())
# [<class '__main__.C'>, <class '__main__.A'>, <class
'__main__.B'>, <class '__main__.Base'>, <class 'object'>]
```

Method resolution order

Пока достаточно знать, что метод `mro` линеаризует иерархию наследования (складывает все в список, если угодно), при этом гарантируя, что в полученной последовательности:

- если A и B были перечислены через запятую в родителях класса (например, class C(A, B)), то A будет идти раньше B
- если B — родитель A, то A будет идти раньше B
- классы не повторяются

Полезные встроенные функции

`isinstance(obj, class_or_tuple, /)` — проверить является ли obj объектом какого-то класса или нескольких:

`issubclass(cls, class_or_tuple, /)` — проверить является ли cls наследником класса или нескольких:

```
isinstance(3, int) # ?
isinstance(true, (bool, int, float)) # ?

issubclass(bool, (str, int)) # ?
```

Полезные встроенные функции

Самостоятельно изучите `getattr`, `setattr`, `delattr`, `vars`, `dir`

Документацию про них можно найти здесь

Код про Hashable

```
from abc import ABC

class MyHashable(ABC):
    @classmethod
    def __subclasshook__(cls, subclass):
        return '__hash__' in dir(subclass) and \
               getattr(subclass, '__hash__') is not None

# см. след. слайд
```

Код про Hashable

```
class A:
    pass

class UnHashable:
    # для того чтобы объект стал unhashable необходимо, чтобы
    # __hash__ был равен None
    # единственный способ это сделать -- задать только __eq__,
    # не задавая __hash__, тогда интерпретатор самостоятельно
    # сделает __hash__ = None
    #
https://docs.python.org/3/reference/datamodel.html#object.__hash
__
    def __eq__(self, other):
        return other is self
```

Код про Hashable

```
print(isinstance(A(), MyHashable)) #?
print("Not here") #?
print(isinstance(UnHashable(), MyHashable)) #?
```

Сломанный mro

```
class Base:
    def foo(self):
        print("Base")


class A(Base):
    def foo(self):
        print("A", end=' ')
        super().foo()


class B(Base):
    def foo(self):
        print("B", end=' ')
        super().foo()
```

Сломанный mro

```
class C(A, B):
    def foo(self):
        print("C", end=' ')
        super().foo()

class E(B, A):
    def foo(self):
        print("E", end=' ')
        super().foo()

class F(E, C):
    pass
```