<u>Объектная Модель Python</u>

ООП в Питончике это ещё одно наслоение пространств имён на всю иерархия этих пространств

Вспомним Три (+1) основные составляющие ООП ипосмотрим в чём они выражаются в питончике:

0. Абстракция - её отдельным полем обычно не выделяют, но логично подразумевают
    Вертикальная: Класс-экземпляр, базовые и производные классы итд
    Горизонтальная: Моделирование разных сущностей одинаковыми конструкциями языка

1. Инкапсуляция - минимализация необходимого информационно пространства, т.е. объединение данных и методов в одном месте
2. Наследование - повторное использование свойств объектов с описанием различий
3. Полиморфизм - возможность для одного и того же кода обрабатывать данные разных типов (но в питончике у нас утиная типизация, так что этот пункт выполняется автоматически)

Классы создаются ключевым словом ***class***, там описываются все его методы и поля. Пустой класс, на самом деле, не пустой, а заполнен базовыми (возможно, пустыми) методами для взаимодействия с ним

Т.к. класс это как бы просто пространство имён, его экземпляр это тоже как бы пространство имён, унаследованное от классового. Поэтому мы можем создавать и удалять поля для класса (и это будет видно в объекте), а можем чисто в объекте

```
>>> class C:
...     pass
...
>>> C
<class '__main__.C'>

>>> dir(C)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
```

```
>>> C.qweqw = 'QWEREWQ'

>>> dir(C)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'qweqw']

>>> C.qweqw
'QWEREWQ'

>>> c = C()
>>> c
<__main__.C object at 0x7fbdc966afd0>

>>> type(c)
<class '__main__.C'>

>>> d = type(c) # где-то в начале конспектов мы делали такую штуку с int и typ.
Тип объекта это же тоже объект))))
>>> dd = d()
>>> type(dd)
<class '__main__.C'>
>>> dd
<__main__.C object at 0x7fbdc966bfd0>

>>> dir(c)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'qweqw']
>>> c.qweqw
'QWEREWQ'
```

Работа с полями класса и экземпляра похожа на взаимоотношение между глобальными и локальными переменными: В экземпляре будет поле из класса с его значением, при этом мы можем в экземпляре его поменять, и как бы создастся новое локальное поле экземпляра. А у класса ничего не поменяется

```
>>> C.QQ = 100500
>>> dir(c)
['QQ', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'qweqw']
>>> c.QQ
100500
>>> c.QQQ = 42
>>> c.QQQ
42
>>> C.QQQ # Не было  - и нет
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'C' has no attribute 'QQQ'. Did you mean: 'QQ'?

>>> c.QQ = 42
>>> c.QQ
42
>>> C.QQ
100500
>>> del c.QQ
>>> c.QQ
100500

>>> dd.QQ
100500
>>> del C.QQ
>>> dd.QQ
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'QQ'
```

Поля можно из при создании класса задать

```
>>> class C:
...     a = "QQQ"
...     b = 100500
...
>>> C.a
'QQQ'
>>> C().a
'QQQ'
```

Здесь, кстати, ещё прикол. Вот мы создали НОВЫЙ класс со СТАРЫМ именем C. То, что обозначили новый класс старым именем C, просто не даст нам через C к старому классу дотягиваться. а через его экземпляры c и dd сможем

Но это было примечание, смотрим дальше:

```
>>> C
<class '__main__.C'>
>>> def fun(a, b):
...     return [a, b]
...
>>> C.m = fun
>>> C.m(1, 2)
[1, 2]
>>> C.m
<function fun at 0x7fbdc967e2a0>

>>> ex = C()
>>> ex.m
<bound method fun of <__main__.C object at 0x7fbdc96801d0>>
```

Если для самого класса это была обычная функция, то для экземпляра это связанное имя - метод класса. Чем нам это грозит?

```
>>> ex.m(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() takes 2 positional arguments but 3 were given
```

Вроде, до двух считать умеем, а тут почему-то три

```
>>> ex.m(1)
[<__main__.C object at 0x7fbdc96801d0>, 1]
>>> ex
<__main__.C object at 0x7fbdc96801d0>
```

Тут особенность методов класса в Питоне выпадает:
Метод - функция, положенная в класс, взятая из экземпляра - всегда имеет своим первым параметром сам экземпляр класса. А тк у нас функции в питоне это просто алгоритмы, которые корректно работают, если к объектам аргументов применимы операции, ошибки у нас никакой и не выпало

```
>>> c = C()
>>> C.m(ex, 1)
[<__main__.C object at 0x7fbdc96801d0>, 1]

>>> c.fun = lambda x: 2*x + 1
>>> c.fun
<function <lambda> at 0x7fbdc967e340>
>>>
>>> C.funC = lambda x: 2*x + 1
>>> c.funC
<bound method <lambda> of <__main__.C object at 0x7fbdc9681190>>
```

Поле класса не является полем экземпляра. У него нет этого поля. Поэтому удалить классовое поле через экземляр мы не можем. Как будто не можем из локальности удалить глобальность

```
>>> class C:
...     a = []
...
>>> c = C()
>>>
>>>
>>> del c.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'a'

>>> class C:
...     a = [1, 2, 3]
...
>>> c = C()
>>> c.a[1] = 100500
>>>
>>> C.a
[1, 100500, 3] # Изменяемые объекты можем менять через экземпляры, отражаться это на классе будет (потому что изменяемые объекты передаются по ссылкам и у нас ссылки на один и тот же объект). Поэтому при работе не надо городить сложности, будет трудно понимать, что мы модифицируем
```

Работа через методы:

```
>>> class C:
...     a = 42
...     def put(self, a):
...             self.a = a
...
>>> c = C()
>>> c.a
42
>>> c.put(100500)
>>> c.a
100500
>>> d = C()
>>> d.a
42
```

Явно в экземпляр всё сделали, в классе не поменялось ничего

Резонный вопрос при работе с ООП: А где конструктор с деструктором? Справедливости ради, это всё делают без вас. Какже тогда при создании объекта задавать ему какие-то данные? Через инициализатор спецметодом ***__init__()***

```
>>> class C:
...     def __init__(self, a, b):
...             self.var = a
...             self.value = b
...
>>> dir(C)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
>>>  # полей var, value нет

>>> c = C()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: C.__init__() missing 2 required positional arguments: 'a' and 'b'
>>> c = C(1, 2)
>>> dir(c)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'value', 'var']
>>> c.value
2
>>> c.var
1
```

И как в функциях, в классах есть свойство самодокументирования

```
>>>
>>> class C:
...     """This is
...             A Class C"""
...
>>> help(C)
```

Самая фанни часть - перегрузка операций (на самом деле никакой перегрузки нет ващпе)
Помним, что любая скобка итд это просто вызов соответствующего метода. То есть мы просто пишем реализацию этой функции так, как нам хочется

```
>>> class C:
...     var = 123
...
>>>
>>> print(C())
<__main__.C object at 0x7fbdc952de10>
>>> str(C())
'<__main__.C object at 0x7fbdc952de10>'

>>> class C:
...     var = 123
...     def __str__(self):
...             return f'<{self.var}>'
...
>>> str(C())
'<123>'
>>> print(C())
<123>
>>> d = C()
>>> d.var = 'QQ'
>>>
>>> print(d)
<QQ>

>>> class C:
...     var = 123
...     def __str__(self):
...             return f'<{self.var}>'
...     def __int__(self):
...             return hash(self.var)
...
>>> c = C()
>>> c.var = 'QQ'
>>> print(C())
<123>
>>> print(c)
<QQ>
>>> int(C())
123
>>> int(c)
-766166371305212996
```

```
>>> class C:
...     def __getattr__(self, name):
...         return f'{name}'
...
>>> c = C()
>>> c.qweqweqwe
'qweqweqwe'
>>> c.__getattr__("QWE")
'QWE'
>>> getattr(c, "asdfgrewerfg")
'asdfgrewerfg'

>>> c
<__main__.C object at 0x7fbdc966b550>
>>> setattr(c, "QWE", 100500)
>>> c.QWE
100500
>>> c.__dict__
{'QWE': 100500}
```

Ровно как инициализатор != конструктор, так и deleter != деструктор, просто завершающая жизнь экземпляра функция. Вызывается автоматически, а можно вызвать ручками

```
>>> class C:
...     def __del__(self):
...         print("ByeBye")
...

>>> c = C()
>>> dir()
['C', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'c', 'sys', 'ver']

>>> del c
ByeBye
>>> dir()
['C', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'sys', 'ver']
```