<u>Слоты Дескрипторы Декораторы</u>

Начнём издалека. Вот у нас есть функция

```
>>> def fun(a, b):
...     return a*2 + b
...
```

Хотим написать обёртку, которая будет печатать аргументы функции

```
>>> def newfun(fun, *args):
...     print(args)
...     res = fun(*args)
...     print(res)
...
>>> fun(1, 2)
4
>>> newfun(fun, 1, 2)
(1, 2)
4
```

Теперь сделаем обёртку непосредственно для ***fun***(не надо будет указывать функцию в параметрах)

```
>>> def newfun(*args):
...     print(args)
...     res = fun(*args)
...     print(res)
...     return res
...
>>> r = newfun(1, 2)
(1, 2)
4
```

Возникает проблема - везде, где было fun, надо теперь писать newfun

Напишем функционал-обёртку для fun: она будет принимать изначальную функциюи возвращать другую, изменённую непосредственно под заданную(т.е. на вход получаем функцию, и возвращаемое значение - функция)

```
>>> def newfun(fun):
...     def wrap(*args):
...         print(args)
...         res = fun(*args)
...         print(res)
...         return res
...     return wrap

>>> def fun(a, b):
...     return 2*a + b
...
>>> newf = newfun(fun)
>>> newf(1, 2)
(1, 2)
4
4
```

А теперь возьмём и переименуем под старое название

```
>>> fun = newfun(fun)
>>> fun(1, 2)
(1, 2)
4
4
```

По сути своей мы изобрели декоратор для функции - обёртку, которая изменяет изначальную функцию с каким-то дополнением
Для декораторов придуман специальный синтаксис. Вместо того чтобы сначала создать функцию, а потом к ней применить декоратор, можно сразу при объявлении декорировать функцию

```
>>> @newfun
... def multi(a, b, c):
...     return a*b * c
...
>>> multi
<function newfun.<locals>.wrap at 0x7f2553a7f060>
>>> multi(1, 2, 3)
(1, 2, 3)
6
6
```

Декорирование - своего рода "наследование для функций" - объектный подход в отсутствие объектов

Декораторы срабатывают снизу вверх

```
>>> @newfun
... @newfun
... def multi(a, b, c):
...     return a*b*c
...
>>> multi(1, 2, 3)
(1, 2, 3)
(1, 2, 3)
6
6
6
```

А что если У декоратора есть параметры?

```
>>> @decorator(args)
    def fun(...):
        ...
```

Тогда fun будет отдекорирован РЕЗУЛЬТАТОМ работы декоратора

```
>>> def repeater(n):
...     def dec(fun):
...             def newfun(*args):
...                     return [fun(*args) for i in range(n)]
...             return newfun
...     return dec
...
>>> @repeater(4)
... def multi(a, b, c):
...     return a*b*c
...
>>> multi(2, 3, 4)
[24, 24, 24, 24]
```

Тк класс является Callable (есть операция () “круглые скобки”), он может выступать декоратором (И его тоже можно декорировать)

```
>>> class repeater4:
...     def __init__(self, fun):
...             self.fun = fun
...     def __call__(self, *args):
...             return [self.fun(*args) for i in range(4)]
>>> @repeater4
... def multi(a, b):
...     return a * b

>>> multi(2, 4)
[8, 8, 8, 8]
>>> multi
<__main__.repeater4 object at 0x7f2553a63110>

>>> def bracer(cls):
...     cls.__str = cls.__str__
...     cls.__str__ = lambda self: "[" + cls.__str(self) + "]"
...     return cls
>>> @bracer
... class C:
...     pass

>>> c = C()
>>>
>>> print(c)
[<__main__.C object at 0x7f2553a8df90>]

>>> def bracer(cls):
...     class newcls(cls):
...             def __str__(self):
...                     return "[" + super().__str__() + "]"
...     return newcls
>>> @bracer
... class C:
...     pass

>>> c = C()
>>> print(c)
[<__main__.bracer.<locals>.newcls object at 0x7f2553a8d2d0>]
>>> c
<__main__.bracer.<locals>.newcls object at 0x7f2553a8d2d0>
>>> C()
<__main__.bracer.<locals>.newcls object at 0x7f2553a8cf50>
```

Приятная фича - Встроенный декоратор wraps

```
>>> from functools import wraps
>>>
>>> def repeater(n):
...     def dec(fun):
...         def newfun(*args):
...             return [fun(*args) for i in range(n)]
...         return newfun
...     return dec
...
>>> @repeater(4)
... def multi(a, b):
...     "Help on multi"
...     return a*b
...
>>> help(multi)

    # Противный непонятный Хелп

>>> multi
<function repeater.<locals>.dec.<locals>.newfun at 0x7f2553a982c0>
```

А теперь так

```
>>> def repeater(n):
...     def dec(fun):
...         @wraps(fun)
...         def newfun(*args):
...             return [fun(*args) for i in range(n)]
...         return newfun
...     return dec
...
>>>
>>> @repeater(4)
... def multi(a, b):
...     "Help on multi"
...     return a*b
...
>>> help(multi)

    # Приятный стильный Хелп, как надо

>>> multi
<function multi at 0x7f25539274c0>
```

Следующая часть - Дескрипторы
Протокол Дескриптора - объект с тремя методами__get__(), __set__(), __delete__()
{Обязательно поддержание get; (хотя бы с get)}

```
>>> class Dsc:
...     def __get__(self, obj, cls):
...             print(f"GET {obj} {cls}")
...             return 100500
...
>>>
>>> class C:
...     f = Dsc()
...
>>> c = C()
>>>
>>> c.f
GET <__main__.C object at 0x7f2553997790> <class '__main__.C'>
100500
```

```
>>> class Dsc:
...     def __set__(self, obj, val):
...         print(f"SET {obj} to {val}")
...         obj.__value = val
...
...     def __get__(self, obj, cls):
...         print(f"GET {obj} {cls}")
...         return obj.__value
...
>>>
>>> class C:
...     f = Dsc()
...
>>> c = C()
>>> c.f = 20
SET <__main__.C object at 0x7f2553a79350> to 20
>>> c.f
GET <__main__.C object at 0x7f2553a79350> <class '__main__.C'>
20
```

```
>>> # Можно попасться на рекурсию в себя же
>>>
>>> # В __set__ он вызывает __str__, в котором дёргает __get__ и по кругу
>>>
>>> # решение - в set выдавать repr от объекта
>>>
>>> class Dsc:
...     def __set__(self, obj, val):
...         print(f"SET {repr(obj)} to {val}")
...         obj.__value = val
...
...     def __get__(self, obj, cls):
...         print(f"GET {obj} {cls}")
...         return obj.__value
...
>>>
>>> class C:
...     f = Dsc()
...     def __str__(self):
...             return f"<{self.f}>"
...
>>> c = C()
>>>
>>> c.f = 20 # Сделали repr - всё ок
SET <__main__.C object at 0x7f2553997810> to 20
>>> c.f # Не сделали repr - рекурсия

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __get__
  File "<stdin>", line 4, in __str__
  File "<stdin>", line 7, in __get__
  File "<stdin>", line 4, in __str__
...
  File "<stdin>", line 4, in __str__
  File "<stdin>", line 7, in __get__
RecursionError: maximum recursion depth exceeded
```

```
>>> class Dsc:
...     def __set__(self, obj, val):
...         print(f"SET {repr(obj)} to {val}")
...         obj.__value = val
...
...     def __get__(self, obj, cls):
...         print(f"GET {repr(obj)} {cls}")
...         return obj.__value
...
>>>
>>> class C:
...     f = Dsc()
...     def __str__(self):
...             return f"<{self.f}>"
...
>>>
>>> c = C()
>>>
>>> c.f = 20
SET <__main__.C object at 0x7f255380f190> to 20
>>> c.f
GET <__main__.C object at 0x7f255380f190> <class '__main__.C'>
20
>>>
```

Теперь про слоты

Иерархия пространств имён для классов и объектов это классная штука, но обычно оно нам не нужно. Ну не будем мы в обычной жизни создавать поля у объектов, не создавая в классах. Так зачем нам отдельные простанства имён?
Решение: фиксированное пространство имён __slots__. Мы фиксируем изменяемые имена переменных, неизменяемые, всё в классе. больше нигде ничего не сможет создаваться. Получаем буквально поля, как в Плюсах

```
>>> class C:
...     __slots__ = "a", "b", "c"
...     d = 100500
...

>>> c = C()
>>> c.a = 123
>>> c.a
123
>>> c.c = 100
>>> c.c
100
>>> c.d
100500
>>> c.d = 2000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object attribute 'd' is read-only
>>> c.e = 234
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute 'e'
```

Тк фиксация идёт только по экземплярам, класс остаётся открытым пространством имён, можем менять любые поля из Класса, но не из объектов

```
>>> C.d = 123
>>> c.d
123
>>> c.d = 124
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object attribute 'd' is read-only
```

И вместо словарика экземпляра у нас теперь только слоты

```
>>> dir(c)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', 'a', 'b',
'c', 'd']
>>> c.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute '__dict__'. Did you mean: '__dir__'?
>>> c.__slots__
('a', 'b', 'c')
>>> C.__dict__
mappingproxy({'__module__': '__main__', '__slots__': ('a', 'b', 'c'), 'd': 123, 'a':
<member 'a' of 'C' objects>, 'b': <member 'b' of 'C' objects>, 'c': <member 'c' of 'C'
objects>, '__doc__': None})
```

Слоты ведут себя, как дескрипторы

```
>>> type(C.a)
<class 'member_descriptor'>
>>> C.a.__get__(c)
123
>>> c.a = 100500
>>> C.a.__get__(c)
100500
```

Обсудим немного стандартных декораторов

```
>>> class C:
...     def method(*args): # это будет просто питоновский метод: функция для класса, метод для объекта
...         print(args)
...     @classmethod
...     def cmethod(*args): # это будет классовый метод: и для класса, и для экземпляра будет отображаться методом
...         print(args)
...     @staticmethod
...     def smethod(*args): # это будет статический метод: и там, и там это будет просто функция
...         print(args)
...
>>> C.method
<function C.method at 0x7f2553a98220>
>>> C.cmethod
<bound method C.cmethod of <class '__main__.C'>>
>>> C.smethod
<function C.smethod at 0x7f255380b380>
>>>
>>> c = C()
>>>
>>> c.method
<bound method C.method of <__main__.C object at 0x7f2553a79190>>
>>> c.cmethod
<bound method C.cmethod of <class '__main__.C'>>
>>> c.smethod
<function C.smethod at 0x7f255380b380>

>>> c.method(1, 2, 3)
(<__main__.C object at 0x7f2553a79190>, 1, 2, 3)
>>> c.cmethod(1, 2, 3)
(<class '__main__.C'>, 1, 2, 3)
>>> c.smethod(1, 2, 3)
(1, 2, 3)
```

@property - обёртка для дескриптора

```
>>> class C:
...     @property
...     def x(self):
...             return 100500
...
>>> c = C()
>>> c.x
100500
>>> C.x
<property object at 0x7f2553993740>
>>> c.x = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: property 'x' of 'C' object has no setter
>>>
```

```
   # Проперти сделал объект, в котором мы можем задавать сеттер и
делитер
>>> class C:
...     @property
...     def x(self):
...             return 100500
...     @x.setter
...     def x(self, val):
...             print(f"Don't wanna set x to {val}")
...
>>> c = C()
>>> c.x
100500
>>> c.x = 123
Don't wanna set x to 123
>>> c.x
100500
```