Метаклассы

Прежде чем падать непосредственно в них, ещё чуть-чуть про обычные классы и наследования.

При создании производных классов можно задавать именованные параметры инициализации, указывая их в описании базового класса через специальный инициализатор ***__init_subclass__***

```
>>> class Titled:
...     def __init_subclass__(cls, title, *args, **kwargs):
...             cls.title = title
...             super().__init_subclass__(*args, **kwargs)
...     def __str__(self):
...             return f'[{self.title}] {super().__str__()}'
...

>>> class C(Titled):
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Titled.__init_subclass__() missing 1 required positional argument: 'title'

>>> class C(Titled, title = 'This is C'):
...     pass
...
>>> c = C()
>>> c.title
'This is C'
>>> print(c)
[This is C] <__main__.C object at 0x7fcc5e17a490>
```

Следующая фишка это работа с дескрипторами. Напомним, что это объект с методами ***get, set, delete (обязателен get)***, реализующий алгоритм обработки записи, печати и удаления какого-то поля

```
>>> class Descr:
...     def __get__(self, obj, cls):
...         print("GET")
...         return obj._x
...     def __set__(self, obj, val):
...         print("SET")
...         obj._x = val
...
>>>
>>> class C:
...     f = Descr()
...
>>> c = C()
>>> c.f = 123
SET
>>> c.f
GET
123
>>>
```

А что будет, если мы сделаем в классе два дескриптора (внутри самого описания дескриптора мы работаем с переменной одного и того же имени ***_x***)?

```
>>> class C:
...     f = Descr()
...     g = Descr()
...
>>> c = C()
>>> c.f = 123
>>> c.g
123
>>> c.__dict__
{'_x': 123}
>>> c.f = 1234
>>> c.__dict__
{'_x': 1234}
```

Хотим, чтобы обращение было разное - надо ходить в словарик непосредственно. Но для этого нужно знать будущее имя переменной. Для этого есть метод ***__set_name__***, который будет нам делать разные имена переменных, когда мы непосредственно работаем с этими полями-дескрипторами

```
>>> class Descr:
...     def __get__(self, obj, cls):
...         return obj.__dict__[self.name]
...     def __set__(self, obj, val):
...         obj.__dict__[self.name] = val
...     def __set_name__(self, obj, name):
...         self.name = name
...
>>> class C:
...     f = Descr()
...     g = Descr()
...
>>> c = C()
>>> c.__dict__
{}
>>> C.__dict__
mappingproxy({'__module__': '__main__', 'f': <__main__.Descr object at
0x7fcc5e17b150>, 'g': <__main__.Descr object at 0x7fcc5e17b110>, '__dict__':
<attribute '__dict__' of 'C' objects>, '__weakref__': <attribute '__weakref__' of 'C'
objects>, '__doc__': None})

>>> c.f = 123
>>> c.g = 42
>>> c.f
123
>>> c.g
42
```

Ещё прикол с дескрипторами: тк мы там при операциях прямо алгоритм прописываем, можно явно присвоить какую-то логику фильтрации значений или типов
Н-р, вот, можем запретить все данные кроме int

```
>>> class Descr:
...     def __get__(self, obj, cls):
...         return obj.__dict__[self.name]
...     def __set__(self, obj, val):
...         if type(val) is not int:
...             raise TypeError("Must be int")
...         obj.__dict__[self.name] = val
...     def __set_name__(self, obj, name):
...         self.name = name
...
>>> class C:
...     f = Descr()
...     g = Descr()
...
>>> c = C()
>>> c.f = 123
>>> c.g = 'QQ'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in __set__
TypeError: Must be int
>>> c.g
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __get__
KeyError: 'g'
>>> c.f
123
>>> C.__dict__
mappingproxy({'__module__': '__main__', 'f': <__main__.Descr object at
0x7fcc5e17b6d0>, 'g': <__main__.Descr object at 0x7fcc5e17b690>, '__dict__':
<attribute '__dict__' of 'C' objects>, '__weakref__': <attribute '__weakref__' of 'C'
objects>, '__doc__': None})
>>> c.__dict__
{'f': 123}
>>> c.g = 234
>>> c.__dict__
{'f': 123, 'g': 234}
```

А теперь МетаКлассы

Основная посылка: в питоне всё — объект. Объекты-экземпляры класса конструируются с помощью вызова самого класса. А кто конструирует класс? Мета-класс!

Зачем нужны мета-классы?

“Если вы думаете, нужны ли вам мета-классы, вам не нужны мета-классы” (c)FBGeorge

По сути, метаклассы отвечают на вопрос бесконечности генерации всего иерархического мира питончика.
Для нас этим великик мета-конструктором вселенной выступает мета-класс ***type***, в котором можно задать имя создаваемого класса, кортеж его родителей по mro, а также пространство имён этого класса

```
>>> C
<class '__main__.C'>
>>> int
<class 'int'>
>>> type(C)
<class 'type'>
>>> type(int)
<class 'type'>
>>> type(type)
<class 'type'>

>>> C = type('C', (), {'a': 1, 'b': 2})
>>> C
<class '__main__.C'>
>>> C.a
1
>>> C.b
2

>>> C = type('C', (), {'a': 1, 'b': 2, 'get': lambda self: (self.a, self.b)})
>>> c = C()
>>> c.get()
(1, 2)
```

Теперь унаследуемся от Тайпа, и сможем сами клепать классы так, как нам надо

```
>>> class otype(type):
...     def __init__(self, Name, Parents, Namespace):
...         print("INIT:", Name, Parents, Namespace)
...         super().__init__(Name, Parents, Namespace)

>>> C = otype('C', (), {'a': 1, 'b': 2, 'get': lambda self: (self.a, self.b)})
INIT: C () {'a': 1, 'b': 2, 'get': <function <lambda> at 0x7fcc5e17fba0>}
>>> C
<class '__main__.C'>
>>> class C(metaclass=otype):
...     pass
...
INIT: C () {'__module__': '__main__', '__qualname__': 'C'}
```

Основные методы, которые мы можем перебить для метакласса

```
>>> class ctype(type):
...     @classmethod
...     def __prepare__(metacls, name, bases, **kwds): # метод создания, если
просто, подготовительного поля под изготавливаемый класс
...         print("prepare", name, bases, kwds)
...         return super().__prepare__(name, bases, **kwds)
...     @staticmethod
...     def __new__(metacls, name, parents, ns, **kwds): # метод конструирования
самого объекта (то, что скрыто от нас до инициализатора)
...         print("new", metacls, name, parents, ns, kwds)
...         return super().__new__(metacls, name, parents, ns)
...     def __init__(cls, name, parents, ns, **kwds):
...         print("init", cls, parents, ns, kwds)
...         return super().__init__(name, parents, ns)
...     def __call__(cls, *args, **kwargs):
...         print("call", cls, args, kwargs)
...         return super().__call__(*args, **kwargs)
...
```

```
>>> class C(metaclass=ctype):
...     pass
...
prepare C () {}
new <class '__main__.ctype'> C () {'__module__': '__main__', '__qualname__': 'C'} {}
init <class '__main__.C'> () {'__module__': '__main__', '__qualname__': 'C'} {}

>>> class C(metaclass=ctype, parameter="QQ"):
...     pass
...
prepare C () {'parameter': 'QQ'}
new <class '__main__.ctype'> C () {'__module__': '__main__', '__qualname__': 'C'}
{'parameter': 'QQ'}
init <class '__main__.C'> () {'__module__': '__main__', '__qualname__': 'C'}
{'parameter': 'QQ'}

>>> c = C()
call <class '__main__.C'> () {}
>>> c = C(234)
call <class '__main__.C'> (234,) {}

>>> c = C()
call <class '__main__.C'> () {}

>>> c = ctype('C', (), {}, qq = "QKRQ")
new <class '__main__.ctype'> C () {} {'qq': 'QKRQ'}
init <class '__main__.C'> () {} {'qq': 'QKRQ'}
```

Сделаем класс, от которого нельзя наследоваться - надо перебить new

```
>>> class final(type):
...     def __new__(metacls, name, parents, namespace):
...         for cls in parents:
...             if isinstance(cls, final):
...                 raise TypeError
...         return super().__new__(metacls, name, parents, namespace)
...
>>> class E(metaclass=final): pass
>>> class D(E): pass
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in __new__
TypeError
```

Singleton - Класс одного экземпляра
Надо перебить call - Если уже есть один экземпляр - вернуть старый. Иначе изготовить

```
>>> class Singleton(type):
...     _instance = None
...     def __call__(cls, *args, **kw):
...         if cls._instance is None:
...             cls._instance = super().__call__(*args, **kw)
...         return cls._instance
...
>>>
>>> class S(metaclass=Singleton):
...     A = 3
>>> s, t = S(), S()
>>> s.newfield = 100500
>>> print(f"{s.newfield=}, {t.newfield=}")
s.newfield=100500, t.newfield=100500
>>> print(f"{s is t=}")
s is t=True
```