Метаклассы Python

Создание классов на лету

```
>>> def create_class(name):
>>>     template = “class {}(object): pass”
>>>     exec(template.format(name), globals())


>>> create_class(“Test”)
>>> test = Test()
>>> print(test)
<__main__.Test object at 0x7f6a9c283350
```

Классы тоже объекты

```
>>> def class_with_method(method):
>>>     class Class(object):
>>>         pass
>>>     setattr(Class, method.__name__, method)
>>>     return Class

>>> def say_hello(self):
>>>     print('hello')

>>> Foo = class_with_metod(say_hello)
>>> foo = Foo()
>>> foo.say_hello()
hello

>>> print(Foo.say_hello)
<function say_hello at 0x7f8a01427b00>
>>> print(foo.say_hello)
<bound method Class.say_hello of
<__main__.class_with_method.<locals>.Class object at 0x7f89ff292e90>>
```

type - фабрика классов

```
>>> Foo = type('Foo', (object,), {'say_hello':
lambda self: print('hello')})
>>> foo = Foo()
>>> foo.say_hello()
hello

>>> print(Foo.say_hello)
<function <lambda> at 0x7f89fd695e60>

>>> print(foo.say_hello)
<bound method Foo.<lambda> of <__main__.Foo
object at 0x7f89ff292e90>>
```

type тоже класс

```
>>> class ClassWithMethod(type):
>>>     def __new__(cls, name, bases, dct):
>>>         return type.__new__(cls, name, bases, dct)
>>>     def __init__(cls, name, bases, dct):
>>>         super(ClassWithMethod, cls).__init__(name, bases, dct)
>>>         setattr(cls, 'say_hello', lambda self: print('hello'))

>>> class Foo(object, metaclass = ClassWithMethod):
>>>     pass

>>> foo = Foo()
>>> foo.say_hello()
hello

>>> print(Foo.say_hello)
<function ClassWithMethod.__init__.<locals>.<lambda> at
0x7f89fd69b440>

>>> print(foo.say_hello)
<bound method Foo.<lambda> of <__main__.Foo object at
0x7f89fd693a10>>
```

Проблемы

```
>>> class Printable(type):
>>>     def whoami(cls):
>>>         print(cls.__name__)

>>> Foo = Printable('Foo', (), {})
>>> Foo.whoami()
Foo

>>> foo = Foo()
>>> foo.whoami()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AttributeError: 'Foo' object has no attribute
'whoami'
```

Наследование метакласса

```
>>> class ClassWithHello(type):
>>>     def __init__(cls, name, bases, dct):
>>>         super(ClassWithHello, cls).__init__(name, bases, dct)
>>>         setattr(cls, 'hello', lambda self: print('hello, ' +
name)

>>> class Base(object, metaclass = ClassWithHello):
>>>     pass

>>> class Derived(Base):
>>>     pass

>>> b = Base()
>>> b.hello()
hello, Base

>>> d = Derived()
>>> d.hello()
hello, Derived
```

Проблемы - 2

```
>>> class MetaA(type):
>>>     pass
>>> class MetaB(type):
>>>     pass

>>> class A(object, metaclass = MetaA):
>>>     pass
>>> class B(object, metaclass = MetaB):
>>>     pass

>>> class C(A, B):
>>>     pass
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: metaclass conflict: the metaclass of a derived class must
be a (non-strict) subclass of the metaclasses of all its bases

>>> class MetaAB(MetaA, MetaB): pass
>>> class C(A, B, metaclass = MetaAB): pass
```

Зачем нужны метаклассы?

Метаклассы - это очень глубокая материя, о которой 99% пользователей даже не нужно задумываться. Если вы не понимаете, зачем они вам нужны – значит, они вам не нужны (люди, которым они на самом деле требуются, точно знают, что они им нужны, и им не нужно объяснять - почему).

Тим Питерс

final класс

```
>>> class final(type):
>>>     def __init__(cls, name, bases, dct):
>>>         super(final, cls).__init__(name, bases, dct)
>>>         finals = [base.__name__ for base in bases if
isinstance(base, final)]
>>>         if len(finals) > 0:
>>>             raise TypeError('Classes ' + str(finals) + '
are final')

>>> class A(object): pass
>>> class B(A, metaclass = final): pass
>>> class C(object, metaclass = final): pass

>>> class D(B, C): pass
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "<stdin>", line 6, in __init__
TypeError: Classes ['B', 'C'] are final
```

класс singleton

```
>>> class singleton(type):
>>>     instance = None
>>>     def __call__(cls, *args, **kwargs):
>>>         if not cls.instance:
>>>             cls.instance = super(singleton,
cls).__call__(*args, **kwargs)
>>>         return cls.instance

>>> class Singleton(object, metaclass = singleton):
>>>     pass

>>> a = Singleton()
>>> b = Singleton()
>>> a is b
```

Регистрация подклассов

```
>>> class RegisterSubclasses(type):
>>>     def __init__(cls, name, bases, dct):
>>>         super(RegisterSubclass, cls).__init__(name,
bases, dct)
>>>         if not hasattr(cls, 'subclass_registry'):
>>>             cls.subclass_registry = {}
>>>         cls.subclass_registry[name] = cls


>>> class Base(object, metaclass = RegisterSubclasses): pass
>>> class DerivedA(Base): pass
>>> class DerivedB(Base): pass
>>> class DerivedC(Base): pass
>>> print(Base.subclass_registry)
{'Base': <class '__main__.Base'>, 'DerivedB': <class
'__main__.DerivedB'>, 'DerivedA': <class
'__main__.DerivedA'>, 'DerivedC': <class
'__main__.DerivedC'>}
```

Подсчет экземпляров

```
>>> class CountObjects(type):
>>>     count = {}
>>>     def __call__(cls, *args, **kwargs):
>>>         instance = super().__call__(*args, **kwargs)
>>>         cls.update()
>>>         return instance
>>>
>>>     def update(cls):
>>>         cls.count[cls] = cls.count.get(cls, 0) + 1
>>>         for base in cls.__bases__:
>>>             if hasattr(base, 'update'):
>>>                 base.update()
>>>
>>>     @staticmethod
>>>     def stat():
>>>         for key, value in CountObjects.count.items():
>>>             print(repr(key), value)
```

Method Resolution Order

```
>>> class A(object):
>>>     pass

>>> class B(A):
>>>     pass

>>> class C(B, A):
>>>     pass

>>> class C(A, B):
>>>     pass
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method
resolution
order (MRO) for bases B, A
```

“Решение”

```
>>> class A(object):
>>>     pass

>>> class B(A):
>>>     pass

>>> class TrckyMRO(type):
>>>     def mro(cls):
>>>         return (cls, A, B, object)

>>> class C(A, B, metaclass = TrickyMRO):
>>>     pass
```