Metaprogramación con Python 3

Andrey Antukh | www.niwi.be | @niwibe | github.com/niwibe

Kaleidos
beautiful code

Las clases

Consideramos esta clase como ejemplo:

```
class Spam(object):
    def __init__(self, name):
        self.name = name

    def say_hello(self):
        print("Hello {0}".format(self.name))
```

Podemos deducir estos datos:

Nombre: “**Spam**”
Bases: “**object**”
Metodos: “**__init__**” y “**say_hello**”

¿Como se construye una clase?

Como primer paso, se crea un dict donde se van a almacenar los atributos de clase:

```
>>> cls_attrs = type.__prepare__()
>>> type(cls_attrs)
<class 'dict'>
```

¿Como se construye una clase?

Como segundo paso, se extrae el “body” de la clase, o es decir lo que representa la definicion de los metoros y atributos:

```
>>> body = """
def __init__(self, name):
    self.name = name
def say_hello(self):
    print("Hello {0}".format(self.name))
"""
```

¿Como se construye una clase?

Como tercer paso, se compila el “body” extraido y se rellena el contexto de la clase:

```
>>> exec(body, globals(), clsattrs)
>>> clsattrs
{'say_hello': <function say_hello at 0x7f0b840e5e60>,
'__init__': <function __init__ at 0x7f0b840e5dd0>}
```

¿Como se construye una clase?

Como cuarto paso, se crea un nuevo tipo:

```
>>> Spam = type("Spam", (object,), clsattrs)
>>> Spam
<class '__main__.Spam'>
>>> instance = Spam("Andrey")
>>> instance.say_hello()
Hello Andrey
```

¿Que es metaclase?

Metaclase, es la clase responsible de crear clases:

```
class SomeMeta(type):
    def __new__(clst, name, bases, attrs):
        print("SomeMeta.__new__", clst, name, bases, {})
        return super().__new__(clst, name, bases, attrs)

    def __init__(cls, name, bases, attrs):
        print("SomeMeta.__init__", cls, name, bases, {})
        super().__init__(name, bases, attrs)

    def __call__(cls, *args, **kwargs):
        print("SomeMeta.__call__", cls, args, kwargs)
        return super().__call__(*args, **kwargs)

class A(metaclass=SomeMeta):
    def __new__(cls, *args, **kwargs):
        print("A.__new__", cls, args, kwargs)
        return super().__new__(cls)

    def __init__(self, *args, **kwargs):
        print("A.__init__", self, args, kwargs)

a = A(2)
```

¿Que es metaclase?

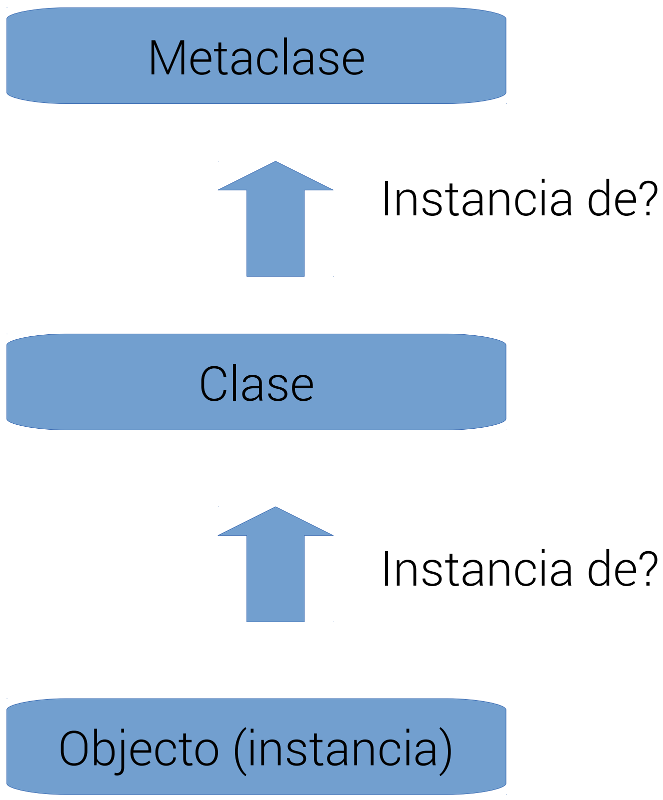
Ejecución en una shell interactiva:

```
~ python -i p1_meta.py
SomeMeta.__new__ <class '__main__.SomeMeta'> A () {}
SomeMeta.__init__ <class '__main__.A'> A () {}
SomeMeta.__call__ <class '__main__.A'> (2,) {}
A.__new__ <class '__main__.A'> (2,) {}
A.__init__ <__main__.A object at 0x7fc5f4b01b10> (2,) {}
```

¿Que es cada cosa?

- meta.**__new__**: se encarga de crear la metaclase
- meta.**__init__**: se encarga de inicializar la metaclase
- meta.**__call__**: hace que la clase que se crea a partir de esta meta clase sea callable
- A.**__new__**: se encargade crear la instancia
- A.**__init__**: se encarga de inicializar la instancia

¿Que es metaclase?



Constructor generico

```
class Person(object):
    def __init__(self, first_name, last_name, birthday,
                 location, zipcode, country, sex):
        self.first_name = first_name
        self.last_name = last_name
        self.birthday = birthday
        self.location = location
        self.zipcode = zipcode
        self.country = country
        self.sex = sex
```

Constructor generico

Vamos a definir estructuras de datos y queremos eliminar la repeticion de definicion del metodo **__init__**.

Constructor generico

Primera aproximación:

```
from inspect import Signature, Parameter

class Struct(object):
    _fields = []

    def __init__(self, *args, **kwargs):
        params = [Parameter(field, Parameter.POSITIONAL_OR_KEYWORD)
                  for field in self._fields]
        sig = Signature(params)

        bound_values = sig.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

class Point(Struct):
    _fields = ["x", "y"]
```

Constructor generico

Ejemplo de uso en una shell:

```
~ python -i example3-signatures.py
>>> p = Point(2, 5)
>>> p.x, p.y
(2, 5)
>>> p = Point(2, 7, z=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example3-signatures.py", line 12, in __init__
    bound_values = sig.bind(*args, **kwargs)
  File "/usr/lib64/python3.3/inspect.py", line 2036, in bind
    return __bind_self._bind(args, kwargs)
  File "/usr/lib64/python3.3/inspect.py", line 2027, in _bind
    raise TypeError('too many keyword arguments')
TypeError: too many keyword arguments
```

Comprobación de tipo de atributos

Primera aproximación, usando properties:

```
class Point(Struct):
    _fields = ["x", "y"]

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        if not isinstance(value, int):
            raise TypeError("unexpected type for x")
        self._x = value
```

Comprobación de tipo de atributos

Ejemplos en la shell interactiva:

```
>>> Point(2, 3)
<__main__.Point object at 0x7f10fd76a150>
>>> Point(2.2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "x4_prop.py", line 16, in __init__
    setattr(self, name, value)
  File "x4_prop.py", line 29, in x
    raise TypeError("unexpected type for x")
TypeError: unexpected type for x
```

Descriptores

Segunda aproximación para comprobar los tipos usando descriptores:

```
class Descriptor(object):
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
```

Descriptores

```
class TypedDescriptor(Descriptor):
    _type = None

    def __set__(self, instance, value):
        if not isinstance(value, self._type):
            raise TypeError("unexpected type for {0}".format(self.name))
        super().__set__(instance, value)
```

Descriptores

Aplicamos los descriptors a nuestra estructura de ejemplo:

```
class Integer(TypedDescriptor):
    _type = int

class Point(Struct):
    _fields = ["x", "y"]

    x = Integer("x")
    y = Integer("y")
```

Observaciones:

- **_fields** sirve para el constructor.
- Los descriptores reciben repetidamente el nombre del atributo.
- Obtenemos el mismo comportamiento que con properties.

Descriptores con Metaclases

Automatizamos ciertas partes de la creacion de clases de nuestras estructuras en el proceso de su definición (compilación):

```
class MetaStruct(type):
    def __prepare__(cls, *args, **kwargs):
        return collections.OrderedDict()

    def __new__(clst, name, bases, attrs):
        params = []
        param_type = Parameter.POSITIONAL_OR_KEYWORD

        for name, attr in attrs.items():
            if isinstance(attr, Descriptor):
                params.append(Parameter(name, param_type))
                attr.name = name

        attrs = dict(attrs)
        attrs["__signature__"] = Signature(params)

        return super().__new__(clst, name, bases, attrs)
```

Descriptores con Metaclases

Automatizamos ciertas partes de la creacion de clases de nuestras estructuras en el proceso de su definición (compilación):

```
class MetaStruct(type):
    def __prepare__(cls, *args, **kwargs):
        return collections.OrderedDict()

    def __new__(clst, name, bases, attrs):
        params = []
        param_type = Parameter.POSITIONAL_OR_KEYWORD

        for name, attr in attrs.items():
            if isinstance(attr, Descriptor):
                params.append(Parameter(name, param_type))
                attr.name = name

        attrs = dict(attrs)
        attrs["__signature__"] = Signature(params)

        return super().__new__(clst, name, bases, attrs)
```

Descriptores con Metaclases

Automatizamos ciertas partes de la creacion de clases de nuestras estructuras en el proceso de su definición (compilación):

```
class MetaStruct(type):
    def __prepare__(cls, *args, **kwargs):
        return collections.OrderedDict()

    def __new__(clst, name, bases, attrs):
        params = []
        param_type = Parameter.POSITIONAL_OR_KEYWORD

        for name, attr in attrs.items():
            if isinstance(attr, Descriptor):
                params.append(Parameter(name, param_type))
                attr.name = name

        attrs = dict(attrs)
        attrs["__signature__"] = Signature(params)

        return super().__new__(clst, name, bases, attrs)
```

Descriptores con Metaclases

Ahora el constructor de la clase base de estructuras:

```
class Struct(object, metaclass=MetaStruct):
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)
```

Descriptores con Metaclases

Y así queda la definición final de estructuras, sin atributos innecesarios y sin repetir el nombre para los descriptores.:

```
class Point(Struct):
    x = Integer()
    y = Integer()
```

¿Como afecta todo esto al rendimiento?

Clase simple, con comprobación de tipos en el constructor:

```
python -m timeit -s "import x2_sim as s" "x = s.Point(2, 5)"
1000000 loops, best of 3: 1.01 usec per loop
```

Structura usando signaturas genericas y properties:

```
python -m timeit -s "import x4_prop as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 61.6 usec per loop
```

Structura usando signaturas genericas y descriptores:

```
python -m timeit -s "import x5_desc as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 64.8 usec per loop
```

Structura usando signaturas genericas, descriptores y metaclases:

```
python -m timeit -s "import x6_meta as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 38.8 usec per loop
```

Generación dinamica código

Para evitar la constante de resolcion de herencia de los descriptores, intentaremos generar en tiempo de definicion (compilación) el codigo del setter y del constructor:

```
def _make_setter_code(descriptor):
    code = ["def __set__(self, instance, value):"]
    for cls in descriptor.__mro__:
        if "code" in cls.__dict__:
            for line in cls.code():
                code.append("    " + line)
    return "\n".join(code)

def _make_init_code(fields):
    code = ["def __init__(self, {0}):\n".format(
                              ", ".join(fields))]
    for field in fields:
        code.append("    self.{0} = {0}\n".format(field))
    return "".join(code)
```

Generación dinamica código

Esto es lo que hacen las funciones para generar el setter y el constructor:

```
python -i x7_exec.py
>>> print(_make_init_code(["x", "y"]))
def __init__(self, x, y):
    self.x = x
    self.y = y
```

```
>>> print(_make_setter_code(Point.x.__class__))
def __set__(self, instance, value):
    if not isinstance(value, self._type):
        raise TypeError('unexpected type')
    instance.__dict__[self.name] = value
```

Generación dinamica código

Este es el aspecto que tendiria el nuevo decriptor:

```
class DescriptorMeta(type):
    def __init__(cls, name, bases, attrs):
        if "__set__" not in attrs:
            exec(_make_setter_code(cls), globals(), attrs)
            setattr(cls, "__set__", attrs["__set__"])
        return super().__init__(name, bases, attrs)

class Descriptor(object, metaclass=DescriptorMeta):
    @staticmethod
    def code():
        return ["instance.__dict__[self.name] = value"]

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]
```

Generación dinamica código

Este es el aspecto que tendiria el nuevo decriptor:

```
class DescriptorMeta(type):
    def __init__(cls, name, bases, attrs):
        if "__set__" not in attrs:
            exec(_make_setter_code(cls), globals(), attrs)
            setattr(cls, "__set__", attrs["__set__"])
        return super().__init__(name, bases, attrs)

class Descriptor(object, metaclass=DescriptorMeta):
    @staticmethod
    def code():
        return ["instance.__dict__[self.name] = value"]

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]
```

Generación dinamica código

Y este es el aspecto que tendria la clase base de las estructuras:

```
class MetaStruct(type):
    def __prepare__(cls, *args, **kwargs):
        return collections.OrderedDict()

    def __new__(clst, name, bases, attrs):
        fields = [k for k,v in attrs.items()
                        if isinstance(v, Descriptor)]
        if fields:
            exec(_make_init_code(fields), globals(), attrs)

        for name in fields:
            attrs[name].name = name

        attrs = dict(attrs)
        return super().__new__(clst, name, bases, attrs)


class Struct(object, metaclass=MetaStruct):
    pass
```

Generación dinamica código

Y este es el aspecto que tendria la clase base de las estructuras:

```
class MetaStruct(type):
    def __prepare__(cls, *args, **kwargs):
        return collections.OrderedDict()

    def __new__(clst, name, bases, attrs):
        fields = [k for k,v in attrs.items()
                        if isinstance(v, Descriptor)]
        if fields:
            exec(_make_init_code(fields), globals(), attrs)

        for name in fields:
            attrs[name].name = name

        attrs = dict(attrs)
        return super().__new__(clst, name, bases, attrs)


class Struct(object, metaclass=MetaStruct):
    pass
```

Generación dinamica código

Y este es el aspecto que tendria la clase base de las estructuras:

```
class MetaStruct(type):
    def __prepare__(cls, *args, **kwargs):
        return collections.OrderedDict()

    def __new__(clst, name, bases, attrs):
        fields = [k for k,v in attrs.items()
                    if isinstance(v, Descriptor)]
        if fields:
            exec(_make_init_code(fields), globals(), attrs)

        for name in fields:
            attrs[name].name = name

        attrs = dict(attrs)
        return super().__new__(clst, name, bases, attrs)


class Struct(object, metaclass=MetaStruct):
    pass
```

¿Hemos mejorado en el rendimiento?

Clase simple, con comprobación de tipos en el constructor:

```
python -m timeit -s "import x2_sim as s" "x = s.Point(2, 5)"
1000000 loops, best of 3: 1.01 usec per loop
```

Structura usando signaturas genericas y properties:

```
python -m timeit -s "import x4_prop as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 61.6 usec per loop
```

Structura usando signaturas genericas y descriptores:

```
python -m timeit -s "import x5_desc as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 64.8 usec per loop
```

Structura usando signaturas genericas, descriptores y metaclases:

```
python -m timeit -s "import x6_meta as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 38.8 usec per loop
```

Structura usando generación dinamica de codigo:

```
python -m timeit -s "import x7_exec as s" "x = s.Point(2, 5)"
100000 loops, best of 3: 2.08 usec per loop
```

¿Hemos mejorado en el rendimiento?

Clase simple, con comprobación de tipos en el constructor:

```
python -m timeit -s "import x2_sim as s" "x = s.Point(2, 5)"
1000000 loops, best of 3: 1.01 usec per loop
```

Structura usando signaturas genericas y properties:

```
python -m timeit -s "import x4_prop as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 61.6 usec per loop
```

Structura usando signaturas genericas y descriptores:

```
python -m timeit -s "import x5_desc as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 64.8 usec per loop
```

Structura usando signaturas genericas, descriptores y metaclases:

```
python -m timeit -s "import x6_meta as s" "x = s.Point(2, 5)"
10000 loops, best of 3: 38.8 usec per loop
```

Structura usando generación dinamica de codigo:

```
python -m timeit -s "import x7_exec as s" "x = s.Point(2, 5)"
100000 loops, best of 3: 2.08 usec per loop
```

Generar codigo a partir de estructuras en json

Supongamos que tenemos la siguiente estructura de datos en json:

```
[
    {
        "name": "Foo",
        "fields": [
            {"name": "foo", "type": "Integer"},
            {"name": "bar", "type": "Integer"}
        ]
    },
    {
        "name": "Person",
        "fields": [
            {"name": "age", "type": "Integer"}
        ]
    }
]
```

Generar codigo a partir de estructuras en json

Podemos generar codigo a partir de esa estructura:

```
def _json_struct_to_code(struct):
    code = ["class {0}(_ts.Struct):".format(struct["name"])]
    for field in struct["fields"]:
        c = "{0} = _ts.{1}()".format(field["name"],
                                     field["type"])
        code.append("    " + c)

    code.append("\n")
    return code

def _json_to_code(filename):
    with io.open(filename, "rt") as f:
        data = json.load(f)
    code = ["import x7_exec as _ts"]
    for struct in data:
        code.extend(_json_struct_to_code(struct))
    return "\n".join(code)
```

Generar codigo a partir de estructuras en json

Este es el resultado:

```
~ python -i x8_json.py
>>> print(_json_to_code("jsonstruct.json"))
import x7_exec as _ts
class Foo(_ts.Struct):
    foo = _ts.Integer()
    bar = _ts.Integer()


class Person(_ts.Struct):
    age = _ts.Integer()
```

Generar codigo a partir de estructuras en json

Creamos la clase responsible de crear el modulo:

```
import imp

class JsonLoader(object):
    def __init__(self, filename):
        self._filename = filename

    def load_module(self, fullname):
        mod = imp.new_module(fullname)
        mod.__file__ = self._filename
        mod.__loader__ = self

        code = _json_to_code(self._filename)
        exec(code, mod.__dict__, mod.__dict__)
        sys.modules[fullname] = mod
        return mod
```

Generar codigo a partir de estructuras en json

Creamos la clase responsible de importar el fichero json:

```
class StructImporter(object):
    def __init__(self, path):
        self._path = path

    def find_module(self, fullname, path=None):
        name = fullname.partition(".")[0]
        if path is None:
            path = self._path

        for dir in path:
            final_name = os.path.join(dir,
                       "{0}.json".format(name))
            if os.path.exists(final_name):
                return JsonLoader(final_name)
        return None

import sys
sys.meta_path.append(StructImporter(sys.path))
```

Generar codigo a partir de estructuras en json

Y este es el resultado:

```
~ python -i x8_json.py
>>> import jsonstruct as jt
>>> jt.__file__
'/home/niwi/niwi-slides/meta-py3/sources/jsonstruct.json'
>>> jt.Person
<class 'jsonstruct.Person'>
>>> jt.Person(2)
<jsonstruct.Person object at 0x7ffb841b0a90>
>>> jt.Person(2.2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 2, in __init__
  File "<string>", line 3, in __set__
TypeError: unexpected type
```

Generar codigo a partir de estructuras en json

Y este es el resultado:

```
~ python -i x8_json.py
>>> import jsonstruct as jt
>>> jt.__file__
'/home/niwi/niwi-slides/meta-py3/sources/jsonstruct.json'
>>> jt.Person
<class 'jsonstruct.Person'>
>>> jt.Person(2)
<jsonstruct.Person object at 0x7ffb841b0a90>
>>> jt.Person(2.2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 2, in __init__
  File "<string>", line 3, in __set__
TypeError: unexpected type
```

Generar codigo a partir de estructuras en json

Y este es el resultado:

```
~ python -i x8_json.py
>>> import jsonstruct as jt
>>> jt.__file__
'/home/niwi/niwi-slides/meta-py3/sources/jsonstruct.json'
>>> jt.Person
<class 'jsonstruct.Person'>
>>> jt.Person(2)
<jsonstruct.Person object at 0x7ffb841b0a90>
>>> jt.Person(2.2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 2, in __init__
  File "<string>", line 3, in __set__
TypeError: unexpected type
```

¿Preguntas?

Andrey Antukh | www.niwi.be | @niwibe | github.com/niwibe

Kaleidos
beautiful code