Объектно-ориентированное программирование на языке Python

Часть 2

Карташов А. А., Курилов Р.

Санкт-Петербургский Академический Университет Кафедра математических и информационных технологий

2010

Иерархия типов языка Python

С чего все начинается?

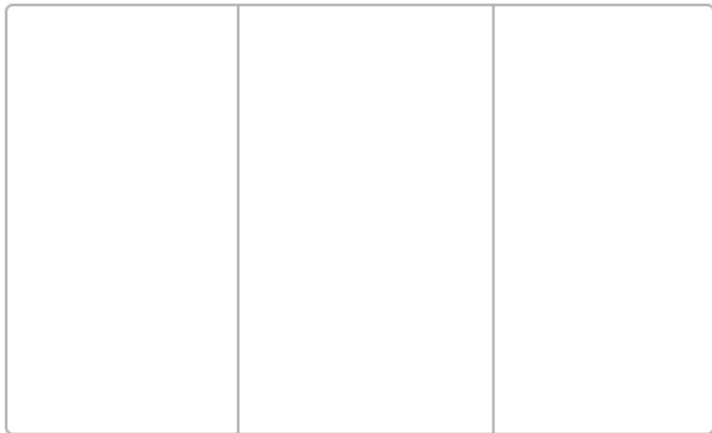


Рис.: Чистый лист бумаги с вертикальными серыми полосами

Курица или яйцо?

Проведем эксперимент:

```
>>> object
<type 'object'>
>>> type
<type 'type'>
>>> type(object)
<type 'type'>
>>> object.__class__
<type 'type'>
>>> object.__bases__
()
>>> type.__class__
<type 'type'>
>>> type.__bases__
(<type 'object'>,)
```

object и type

Объекты object и type являются базовыми в языке Python. Исходя из предыдущего эксперимента, можно построить диаграмму отношений между ними.

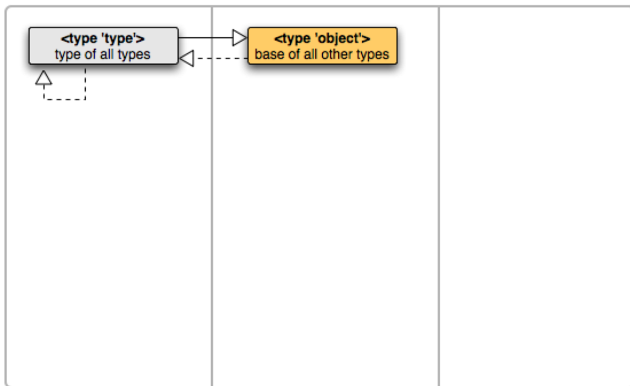


Рис.: Отношения между object и type

Объекты-типы

Продолжим эксперименты:

```
>>> isinstance(object, object)
True
>>> isinstance(type, object)
True
```

Объекты `object` и `type` — это объекты-типы в языке `Python`. Это означает, что

- они могут представлять абстрактные типы данных в программе;
- они могут быть *унаследованы* другими объектами;
- можно создавать их *экземпляры*;
- типом любого объекта-типа является `<type 'type'>`;
- одни их называют *типами*, другие — *классами*.

Еще о встроенных типах

Продолжим эксперименты:

```
>>> list
<type 'list'>
>>> list.__class__
<type 'type'>
>>> list.__bases__
(<type 'object'>,)
>>> tuple.__class__, tuple.__bases__
(<type 'type'>, (<type 'object'>,))
>>> dict.__class__, dict.__bases__
(<type 'type'>, (<type 'object'>,))
>>>
>>> mylist = [1,2,3]
>>> mylist.__class__
<type 'list'>
```

Объекты list, tuple, dict

Теперь мы можем расширить нашу диаграмму типов Python'а:

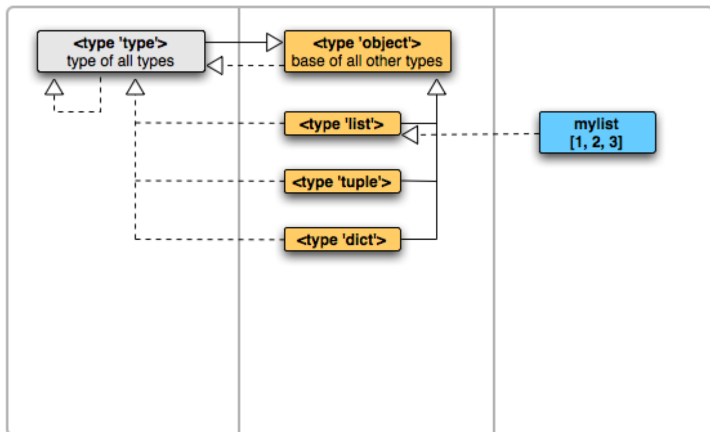


Рис.: Встроенные типы языка Python

Пользовательские типы

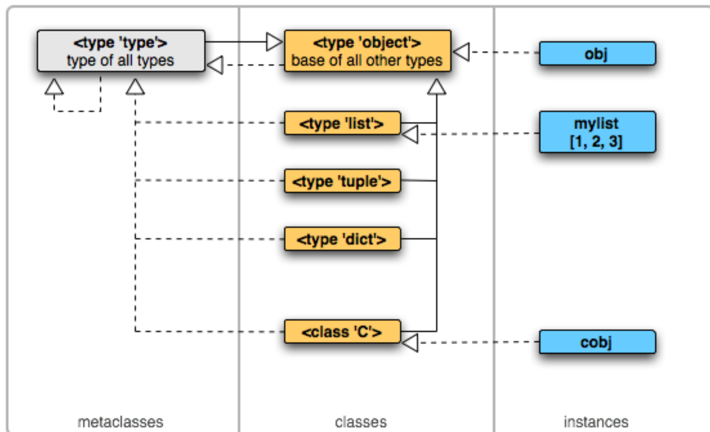
«Старый» стиль

```
class Old:
    pass

>>> old = Old()
>>> type(old)
<type 'instance'>
>>> type(Old)
<type 'classobj'>
>>> issubclass(Old, object)
False
```

«Новый» стиль

```
class New(object):
    pass

>>> new = New()
>>> type(new)
<class '__main__.New'>
>>> type(New)
<type 'type'>
```

Отношения объектов в языке Python



Атрибуты и методы

Пользовательские атрибуты и методы

```
>>> class C(object):
...     classattr = "attr on class"
...     def f(self):
...             return "function f"
...
>>> C.__dict__
{'classattr': 'attr on class', '__module__': '__main__',
 '__doc__': None, 'f': <function f at 0x008F6B70>}
>>> c = C()
>>> print c.__dict__
{}
>>> c.classattr is C.__dict__['classattr']
True
>>> c.f is C.__dict__['f']
False
```

Атрибут `__dict__` и поиск имен

Атрибут `__dict__` представляет собой таблицу «ключ—значение», которая хранит имена *пользовательских* атрибутов объекта.

Алгоритм поиска имен атрибутов

Пусть нужно определить значение выражения `<объект>.<атрибут>`. Интерпретатор `Python`'a будет действовать по следующему алгоритму:

1. поиск значения `атрибут` в таблице `<объект>.__dict__` и среди встроенных атрибутов объекта;
2. поиск значения `атрибут` в таблице `<объект>.__class__.__dict__` и среди встроенных атрибутов объекта `<объект>.__class__`;
3. поиск имени атрибута в объектах кортежа `<объект>.__class__.__bases__`.

От функции к методу

Еще раз обратимся к примеру:

```
>>> class C(object):
...     classattr = "attr on class"
...     def f(self):
...             return "function f"
...
>>> C.__dict__
{'classattr': 'attr on class', '__module__': '__main__',
 '__doc__': None, 'f': <function f at 0x008F6B70>}
>>> c = C()
>>> c.f is C.__dict__['f']
False
>>> c.f
<bound method C.f of <__main__.C instance at 0x008F9850>>
>>> C.__dict__['f'].__get__(c, C)
<bound method C.f of <__main__.C instance at 0x008F9850>>
```

Выводы

- Метод объекта не то же самое, что и метод класса, экземпляром которого этот объект является.
- При создании экземпляра `i` класса `C` атрибут `i.a` инициализируется следующим образом: `C.__dict__['a'].__get__(c, C)`.

Интерфейс дескриптора

Если у объекта есть методы `__get()__` и, возможно, `__set()__` и `__delete()__`, то говорят, что такой объект реализует *интерфейс дескриптора*.

Пример реализации интерфейса дескриптора

```
class Desc(object):
    def __get__(self, obj, cls=None):
        pass

    def __set__(self, obj, val):
        pass

    def __delete__(self, obj):
        pass
```

Два типа дескрипторов

«Слабый» дескриптор

```
>>> class Weak(object):
...     def __get__(self, obj, cls):
...             return "WeakValue"

>>> class A(object):
...     a = Weak()

>>> i = A()
>>> print i.a
WeakValue
>>> i.a = "NewValue"
>>> print i.a
NewValue
```

«Сильный» дескриптор

```
>>> class Strong(object):
...     def __get__(self, obj, cls):
...             return "StrongValue"
...
...     def __set__(self, obj, cls):
...             pass

>>> class A(object):
...     a = Weak()

>>> i = A()
>>> print i.a
StrongValue
>>> i.a = "NewValue"
>>> print i.a
StrongValue
```

Вывод

«Сильный» дескриптор полностью изменяет поведение связанного с ним атрибута.

Дескриптор property

Модуль __builtins__ содержит класс property, позволяющий создавать атрибуты, обладающие привычным поведением свойств.

Пример использования

Без декоратора

```
class C(object):
    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value
```

С декоратором

```
class C(object):
    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value
```

Дескриптор staticmethod

Атрибут, являющийся экземпляром дескриптора staticmethod, ведет себя как статический метод таких языков, как C++ или Java.

Пример

```
class C(object):
    @staticmethod
    def f(arg1, arg2, ...): ...
```

Дескриптор staticmethod не передает обернутому методу ссылку на вызывающий объект в качестве первого аргумента.

Дескриптор `super`

Дескриптор `super` предназначен для реализации предотвращения вызова переопределенного метода базового класса, который в иерархии наследования встречается более одного раза.

Пример использования super

```
class A(object):
    def foo():
        print "In class A"

class B(A):
    def foo():
        super(B, self).foo()
        print "In class B"

class C(A):
    def foo():
        super(C, self).foo()
        print "In class C"

class D(B, C):
    def foo():
        super(D, self).foo()
        print "In class D"
```

Пример использования super

Протестируем пример:

```
>>> d = D()
>>> d.foo()
In class A
In class C
In class B
In class D
```

Однако...

Такой код не работает:

```
class A(object):
    def foo():
        print "In class A"

class B(object):
    def foo():
        print "In class B"

class C(A, B):
    def foo():
        super(C, self).foo()
        print "In class C"
```

Действительно

```
>>> c = C()
>>> c.foo()
In class A
In class C
```

Запрет модификации объектов-классов и их экземпляров

В Python'е классы могут вести себя как классы в языках со статической типизацией — программист может явно запретить модификацию объектов-классов. Для этого в классе достаточно объявить атрибут `__slots__`, который заменит `__dict__` в объекте-классе и всех его экземплярах. Атрибут `__slots__` представляет собой список имен пользовательских атрибутов класса.

Шаблоны проектирования в языке Python

Шаблон Factory

Фабрика абстрагирует от деталей создания экземпляра объекта. Вызывающая функция в результате даже не знает тип создаваемого объекта.
Благодаря динамической типизации, фабрики в Python'е не требуют специального синтаксического оформления.

```
import random

def listOfRandom(n):
    return [random.random() for i in range(n)]
```

Интермеццо — Duck typing

Шаблон Borg

Был придуман для преодоления проблем шаблона Singleton.

Пример

```
class Borg:
    __shared_state = {}
    def __init__(self):
        self.__dict__ = self.__shared_state
```

Фабрика синглтонов (Flyweight)

Используется в том случае, когда нужно много экземпляров класса, но не все они должны быть разными.

Пример

```
import weakref

class Instrument(object):
    _InstrumentPool = weakref.WeakValueDictionary()

    def __new__(cls, name):
        obj = Instrument._InstrumentPool.get(name, None)

        if not obj:
            print "new",name
            obj = object.__new__(cls)
            Instrument._InstrumentPool[name] = obj

        return obj
```

Фабрика синглтонов (Flyweight)

```
class Instrument(object):
    def __init__(self, name):
        '''Complete object construction'''
        self.name = name
        print "New instrument @%04x, %s"
              % (id(self), name)
```

Итератор

Обеспечивает прозрачный последовательный доступ к элементам составного объекта.

Пример

```
class NodeIterator:
    def __init__(self, node):
        self.stack = [node]

    def __iter__(self):
        return self

    def next(self):
        if not self.stack:
            raise StopIteration

         node = self.stack.pop(-1)
         while isinstance(node, Node):
             self.stack.append(node.right)
            node = node.left
            return node
```

Итератор

```
class Node(object):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __iter__(self):
      return Node.NodeIterator(self)
```

Генератор

Генератор — это аналог итератора, в котором не нужно явно хранить внутреннее состояние. Любая функция, в теле которой встречается оператор `yield`, является итератором.

Пример

```
class Node(object):
    def __iter__(self):
        if (isinstance(self.left, Node)):
            for n in self.left:
                yield n
        else:
            yield self.left

        if (isinstance(self.right, Node)):
            for n in self.right:
                yield n
        else:
            yield self.right
```

Миксины

Миксин — класс, реализующий определенную часть функциональности, предназначенной только для производных классов.

Пример

```
class CursorWarningMixIn:
class CursorStoreResultMixIn:
class CursorUseResultMixIn
class CursorTupleRowsMixIn:
class CursorDictRowsMixIn(CursorTupleRowsMixIn):

conn = MySQLdb.connection (cursorclass=MySQLdb.DictCursor)
```

Пример миксина

```
class NamedValueAccessible:
    def valueForKey(self, key, default=_NoDefault):
        klass     = self.__class__
        attr      = None
        method    = getattr(klass, key, None)
        if not method:
            method = getattr(klass, '_' + key, None)
            if not method:
                attr = getattr(self, key, None)
                if not attr:
                    attr = getattr(self, '_' + key, None)
                    if not attr:
                        if default!=_NoDefault:
                            return default
                        else:
                            raise NamedValueError, key
        if method:
            return method(self)
        if attr:
            return attr
```

Представление типов в интерпретаторе языка Python

Структура интерпретатора

- Исполнительная часть CPython представляет собой стековую виртуальную машину.
- При загрузке модуля он компилируется в байт-код для этой виртуальной машины.
- При работе в интерактивном режиме интерпретатор Python'а компилирует вводимые команды и исполняет их на виртуальной машине.

Виртуальная машина CPython'a

Команды виртуальной машины (файл `Include/opcode.h`):

- операции со стеком,
- унарные операции,
- бинарные операции, не модифицирующие операнды,
- бинарные операции, модифицирующие операнды,
- команды вывода на консоль,
- команды управления потоком исполнения (переходы, циклы, вызов функции, обработка исключений),
- операции со встроенными агрегатными типами (списками, кортежами, словарями, классами),
- команды обращения к переменным локальной и глобальной области видимости.

Тип PyObject — основа CPython'a

Любой объект Python'a в реализации CPython внутри интерпретатора представлен объектом типа PyObject (Include/object.h).

```
typedef struct _object {
  struct _object *_ob_next;
  struct _object *_ob_prev;

  Py_ssize_t ob_refcnt;
  struct _typeobject *ob_type;
} PyObject;
```

Представление типа `int`

Реализация типа `int` довольно очевидна (`Include/intobject.h`):

```c
typedef struct {
  PyObject_HEAD

  long ob_ival;
} PyIntObject;
```

Представление типа `int`

При выполнении операций над целыми числами методы `PyIntObject` не вызываются: (`Python/ceval.c`):

```
case BINARY_ADD:
  w = POP();
  v = TOP();

  if (PyInt_CheckExact(v) && PyInt_CheckExact(w)) {
    /* INLINE: int + int */
    register long a, b, i;
    a = PyInt_AS_LONG(v);
    b = PyInt_AS_LONG(w);
    /* cast to avoid undefined behaviour
       on overflow */
    i = (long)((unsigned long)a + b);
    if ((i^a) < 0 && (i^b) < 0)
      goto slow_add;
    x = PyInt_FromLong(i);

    SET_TOP(x);
  }
```

Представление типа float

Тип float также представляется довольно просто (Include/floatobject.h):

```
typedef struct {
  PyObject_HEAD

  double ob_fval;
} PyFloatObject;
```

Функция сложения чисел с плавающей запятой Python/floatobject.c):

```
static PyObject *
float_add(PyObject *v, PyObject *w)
{
      double a,b;
      CONVERT_TO_DOUBLE(v, a);
      CONVERT_TO_DOUBLE(w, b);
      PyFPE_START_PROTECT("add", return 0)
      a = a + b;
      PyFPE_END_PROTECT(a)
      return PyFloat_FromDouble(a);
}
```

Представление типа list

Список в CPython'е есть массив указателей на PyObject:

```
typedef struct {
  PyObject_HEAD

  PyObject **ob_item;
  Py_ssize_t allocated;
} PyListObject;
```

Представление типа list

Процедура создания списка:

```
PyObject *
PyList_New(Py_ssize_t size)
{
    PyListObject *op;
    size_t nbytes;

    nbytes = size * sizeof(PyObject *);
    if (numfree) {
        numfree--;
        op = free_list[numfree];
        _Py_NewReference((PyObject *)op);
    } else {
        op = PyObject_GC_New(PyListObject, &PyList_Type);
        if (op == NULL)
            return NULL;
    }
    if (size <= 0)
        op->ob_item = NULL;
    else {
        op->ob_item = (PyObject **) PyMem_MALLOC(nbytes);
        if (op->ob_item == NULL) {
            Py_DECREF(op);
            return PyErr_NoMemory();
        }
        memset(op->ob_item, 0, nbytes);
    }
    Py_SIZE(op) = size;
    op->allocated = size;
    _PyObject_GC_TRACK(op);
    return (PyObject *) op;
}
```

Представление типа list

Процедура обращения к элементу списка по целому индексу:

```
case BINARY_SUBSCR:
  w = POP();
  v = TOP();
  if (PyList_CheckExact(v) && PyInt_CheckExact(w)) {
    Py_ssize_t i = PyInt_AsSsize_t(w);
    if (i < 0)
      i += PyList_GET_SIZE(v);
    if (i >= 0 && i < PyList_GET_SIZE(v)) {
      x = PyList_GET_ITEM(v, i);
      Py_INCREF(x);
    }
    else
      goto slow_get;
  }
  else
    slow_get:
      x = PyObject_GetItem(v, w);
```

Представление функций

```
typedef struct {
    PyObject_HEAD
    PyObject *func_code;
    PyObject *func_globals;
    PyObject *func_defaults;
    PyObject *func_closure;
    PyObject *func_doc;
    PyObject *func_name;
    PyObject *func_dict;
    PyObject *func_weakreflist;
    PyObject *func_module;
} PyFunctionObject;
```

Представление типов

Описание типа _typeobject (Include/object.h):

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    cmpfunc tp_compare;
    reprfunc tp_repr;

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    long tp_flags;

    const char *tp_doc; /* Documentation string */

    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    struct _typeobject *tp_base;
    PyObject *tp_dict;
} PyTypeObject;
```

Так что же такое type

Выполним команду

```
>>> type(type)
```

в интерпретаторе, работающем под отладчиком, в котором стоит точка останова на функции `PyEval_EvalCodeEx` (`Python/ceval.c`)
Первая команда скомпилированной программы ищет имя `type` (как аргумент функции `type`). Это имя является глобальным, поэтому мы сможем получить указатель на этот объект после выполнения строчки

```
x = PyDict_GetItem(f->f_globals, w);
```

Что же такое type

Посмотрим содержимое переменной `w`:

```
(gdb) p x
$98 = (PyObject *) 0x81e1160
(gdb) p p *x->ob_type
$99 = {_ob_next = 0x81dc5e0, _ob_prev = 0x81e1420,
       ob_refcnt = 40, ob_type = 0x81e1160, ...}
```

Вывод

Поле `ob_type` объекта `type` указывает на сам объект `type`. Этим как раз и объясняются результаты экспериментов, описанных в первой части.

Q?

A!