

Python 学习笔记

第 2 版

好好学习，天天向上



前言

全新编写的 Python 笔记，作为《蟒原 —— Python 发现之旅》的配套资料。

和《蟒原》深入源码分析具体实现过程不同，本笔记力求简洁，用平直文字和浅显示例说明基本编码操作。目的是为了便于阅读，以作日常复习之用。当然，如对某话题有兴趣，不妨和《蟒原》对照着看，不仅能满足好奇心，还会有意外收获。毕竟 CPython 的开发人员都是 "牛牛"，其久经考验的代码和理论有许多可学习借鉴之处。

- 示例代码主要在 IPython、CPython 2.7 中编写。
- 为阅读方便，代码和输出结果有所剪辑。
- 因版本和运行期环境不同，输出结果，尤其是内存地址会存在差异。
- 如不做特别说明，书中 Python 均指 CPython (www.python.org)。
- 第一部分主要讲述语言相关内容，不会涉及太多标准库内容。
- 本书不定期更新，可以到 github.com/qyuheng 下载。

如果您发现错漏，请与我联系，以便及时修正。谢谢！

代码测试环境：

- CPython 2.7, IPython 0.13
- MacBook Pro, 8GB, Mac OS X Lion 10.8.2

联系方式：

email: qyuheng@hotmail.com

weibo: <http://weibo.com/qyuheng>

QQ: 1620443

雨痕 2012年冬于北京家中



更新记录

2012-12-15 开始。

2012-12-17 完成第 1 章。

2012-12-22 完成第 2 章。

2012-12-23 完成第 3 章。

2012-12-25 完成第 4 章。

2012-12-27 完成第 5 章。

2012-12-30 完成第 6 章。

2013-01-02 完成第 7 章。

2013-01-03 完成第 8 章。

2013-01-04 完成第 9 章。

2013-01-05 完成第 10、11 章。

2013-01-06 补充 7.5 super 的部分内容，附录 ipython、pdb。

草稿，尚未开始文字校正。



目录

第一部分 Python 语言	7
第 1 章 基本环境	8
1.1 虚拟机	8
1.2 类型和对象	8
1.3 名字空间	10
1.4 内存管理	12
1.5 编译	18
1.6 执行	20
第 2 章 内置类型	22
2.1 数字	22
2.2 字符串	25
2.3 列表	31
2.4 元组	33
2.5 字典	34
2.6 集合	38
第 3 章 表达式	41
3.1 语法规则	41
3.2 命名规则	43
3.3 赋值	44
3.4 表达式	45
3.5 运算符	50
3.6 类型转换	53
3.7 常用函数	54
第 4 章 函数	57

4.1 创建	57
4.2 参数	58
4.3 作用域	60
4.4 闭包	63
4.5 堆栈帧	65
4.6 包装	67
第 5 章 迭代器	68
5.1 迭代器	68
5.2 生成器	69
5.3 模式	71
5.4 宝藏	72
第 6 章 模块	78
6.1 模块对象	78
6.2 搜索路径	79
6.3 导入模块	80
6.4 构建包	83
第 7 章 类	86
7.1 名字空间	86
7.2 字段	87
7.3 属性	90
7.4 方法	93
7.5 继承	95
7.6 开放类	102
7.7 操作符重载	104
第 8 章 异常	109
8.1 异常	109

8.2 断言	111
8.3 上下文	111
第 9 章 装饰器	115
第 10 章 描述符	121
第 11 章 元类	125
第二部分 标准库	130
第三部分 扩展库	131
附录	132
CPython	133
IPython	134
PDB	136

第一部分 Python 语言

Python 语言相关.....

第 1 章 基本环境

1.1 虚拟机

Python 是一种半编译半解释型运行环境。首先，它会在模块 "载入(或导入)" 时将源码编译成类似字节码 (Byte code)。而后，这些字节码会被虚拟机在一个 "巨大" 的函数里解释执行。这是导致 Python 性能较低的重要原因，好在现在有了内置 Just-in-time 二次编译器的 [PyPy](#) 可供选择。

当虚拟机开始运行时，它通过初始化函数完成整个运行环境设置：

- 创建解释器和主线程状态对象，这是整个进程的根。
- 初始化内置类型。数字、列表等都有专门的性能缓存策略需要处理。
- 创建 `__builtins__` 模块，这个模块持有所有内置类型和函数。
- 创建 `sys` 模块，其中包含了 `sys.path`、`modules` 等重要的运行期信息。
- 初始化 import 机制。
- 初始化内置 Exception。
- 创建 `__main__` 模块，准备运行所需的名字空间、还包括 `__name__` 等。
- 通过 `site.py` 将 `site-packages` 中的第三方扩展库添加到 `sys.path` 搜索路径列表。
- 执行 `py` 源码。执行前会将 `__main__.__dict__` 作为名字空间传递进去。
- 源码执行结束。
- 执行清理动作，包括调用退出函数，GC 清理现场，释放所有导入模块等。
- 终止进程。

对此有兴趣的可以阅读《蟒原》一书，其中有详细的代码分析。

1.2 类型和对象

先有类型 (Type)，而后才能生成实例对象 (Instance)。Python 中的一切都是对象，包括类型在内的每个对象都包含一个标准头，通过头部信息就可以明确知道其具体类型。

头信息由 "引用计数" 和 "类型指针" 组成，前者在对象被引用时增加，超出作用域或手工释放后递减。等于 0 时其内存被虚拟机回收 (某些被缓存的对象计数器永远不会为 0)。

以 `int` 为例，对应 Python 结构定义是：

```
#define PyObject_HEAD          \
    Py_ssize_t ob_refcnt;      \
    struct _typeobject *ob_type;

typedef struct _object {
    PyObject_HEAD
} PyObject;
```



```
typedef struct {
    PyObject_HEAD          // 在 64 位平台上，头部长度为 16 字节。
    long ob_ival;          // long 是 8 字节。
} PyIntObject;
```

可以用 `sys` 中的函数测试一下。

```
>>> import sys

>>> x = 0x1234          # 不要使用 [-5, 257) 之间的小数字，它们有专门的缓存机制。

>>> sys.getsizeof(x)     # 符合长度预期。
24

>>> sys.getrefcount(x)   # sys.getrefcount() 读取头部引用计数，注意形参也会增加一次引用。
2

>>> y = x
>>> sys.getrefcount(x)
3

>>> del y
>>> sys.getrefcount(x)
2
```

类型指针则指向 `Type` 对象，其中包含了其继承关系以及静态成员 (比如方法) 信息。所有的内置类型对象都能从 `types` 模块中找到，至于 `int`、`long`、`str` 这些则是一种简短别名。

```
>>> import types

>>> x = 20

>>> type(x) is types.IntType      # is 通过指针判断是否指向同一对象。
True

>>> x.__class__                  # __class__ 通过类型指针来获取类型对象。
<type 'int'>

>>> x.__class__ is type(x) is int is types.IntType
True

>>> y = x
>>> hex(id(x)), hex(id(y))        # id 返回对象标识，其实就是内存地址。
('0x7fc5204103c0', '0x7fc5204103c0')

>>> hex(id(int)), hex(id(types.IntType))
('0x1088cebd8', '0x1088cebd8')
```

除了 `int` 这样的固定长度类型外，还有 `long`、`str` 这些变长对象。其头部会多出一个记录元素项数量的字段。比如 `str` 的字节数量，`list` 列表的长度等等。

```
#define PyObject_VAR_HEAD          \
    PyObject_HEAD                  \
    Py_ssize_t ob_size; /* Number of items in variable part */

typedef struct {
    PyObject_VAR_HEAD
} PyVarObject;
```

有关类型和对象更多的信息，将在后续章节中详述。

1.3 名字空间

名字空间是 Python 最核心的内容，是必须要搞明白的。

```
>>> haha
NameError: name 'haha' is not defined
```

和 C 变量名是内存地址别名不同，Python 的名字实际上是一个字符串对象，它和目标对象一起在名字空间中构成一个 `name/object` 关联项。名字空间决定了对象的作用域和生存周期。

可以用内置函数 `globals()` 获取模块级别名字空间，`locals()` 获取当前上下文，比如函数、方法内部的名字空间。

```
>>> x = 123
>>> globals()
{'x': 123, .....}
```

可以看出，名字空间就是一个字典。也就说我们完全可以在名字空间添加项来创建“变量”。

```
>>> globals()["y"] = "Hello, World!"
>>> y
'Hello, World!'
```

在 Python 源码中，有句话：Names have no type, but objects do.

名字的作用仅仅是在某个时刻与名字空间中的目标对象进行关联。名字并不包含目标对象的任何信息，只有通过对象的类型指针才能获知其具体类别。正因为名字的弱类型特征，我们可以在运行期随时将其关联到任何类型的对象。

```
>>> y
'Hello, World!'
>>> type(y)
<type 'str'>
```

```
>>> y = __import__("string")
>>> type(y)
<type 'module'>

>>> y.digits
'0123456789'
```

在函数外部 `locals()` 和 `globals()` 作用完全相同。而在函数内部调用时，`locals()` 则是获取当前堆栈帧的名字空间，其中存储的是函数参数、局部变量等信息。

```
>>> import sys

>>> globals() is locals()
True

>>> locals()
{
    '__builtins__': <module '__builtin__' (built-in)>,
    '__name__': '__main__',
    'sys': <module 'sys' (built-in)>,
    '__doc__': None,
    '__package__': None
}

>>> def test(x):                                # 请对比下面的输出内容。
...     y = x + 100
...     print locals()                          # 可以看到 locals 名字空间中包含当前局部变量。
...     print globals() is locals()            # 此时 locals 和 globals 指向不同名字空间。

...     frame = sys._getframe(0)               # _getframe() 可以获取调用堆栈链上的堆栈帧。
...     print locals() is frame.f_locals        # locals 名字空间实际就是当前堆栈帧的名字空间。
...     print globals() is frame.f_globals     # 通过 frame 我们也可以访问外部模块的名字空间。

>>> test(123)
{'y': 223, 'x': 123}
False
True
True
```

在函数中调用 `globals()` 时，总是获取包含该函数定义的模块名字空间，而非调用处。

```
>>> pycat test.py
"""
    Hello, World!
"""

a = 1
def test():
```

```

        print {k:v for k, v in globals().items() if k != "__builtins__"}

>>> import test

>>> test.test()
{
    'a': 1,
    '__file__': 'test.pyc',
    '__package__': None,
    'test': <function test at 0x10bd85e60>,
    '__name__': 'test',
    '__doc__': '\n    Hello, World!\n'
}

```

可以通过 `<module>.__dict__` 访问其他模块的名字空间。

```

>>> test.__dict__                                     # test 模块的名字空间
{
    'a': 1,
    '__file__': 'test.pyc',
    '__package__': None,
    'test': <function test at 0x10bd85e60>,
    '__name__': 'test',
    '__doc__': '\n    Hello, World!\n'
}

>>> import sys
>>> sys.modules[__name__].__dict__ is globals()      # 当前模块名字空间和 globals 相同。
True

```

函数的名字空间还会涉及到 **LEGB** 查找规则，以及超出作用域等问题。而对象成员则另有一套名字空间。所有这些问题，将在相关章节中做出说明。

透过名字空间来管理上下文对象，带了无与伦比的灵活性，但也牺牲了部分性能。毕竟从一个 `dict` 查找对象远比指针要低效许多。各有得失！

1.4 内存管理

为了提升性能，**Python** 在内存管理上做了大量工作。最直接的做法就是用内存池来减少操作系统内存分配和回收操作，那些小于等于 256 字节对象，将直接从内存池中获取存储空间。

根据需要，**Python** 每次会从操作系统分配一块 256KB，名为 **arena** 的大块内存。进而按系统页大小，分成多个 **pool**。每个 **pool** 用于存储一种大小规格 (8 字节的倍数) 对象。**pool** 里面存储对象的小块内存被叫做 **block**，这是内存池的最小单位。所有这些都用头信息和链表管理起来，以便于

快速查找和分配。比如存储 13 字节大小的对象时，就先找到 16 字节容量规格的可用 **pool**，并从中获取一块空闲的 **block**。

大于 256 字节的对象，直接用 **malloc/free** 在堆上分配内存。绝大多数对象都小于这个阈值，因此内存池策略可有效提升性能。

arena 的数量总数是有限制的，当总容量超出特定限制 (默认是 64MB) 时，就不再请求分配 **arena** 内存。而是如同大对象一样，直接在堆上分配对象内存。另外，**arena** 不再使用时，会被虚拟机释放，将内存交还给操作系统。

引用传递

在 **Python** 中，对象总是引用传递的。也就是说通过复制对象指针来实现多个名字指向同一个对象。因为 **arena** 也是在 **Heap** 上分配的，所以无论何种类型何种大小的对象，总是存储在 **heap** 上。**Python** 也没有值类型和引用类型一说，就算是一个简单的整数也拥有标准头。

```
>>> a = object()

>>> b = a
>>> a is b
True

>>> hex(id(a)), hex(id(b))           # 地址相同，意味着对象是同一个。
('0x10b1f5640', '0x10b1f5640')

>>> def test(x):
...     print hex(id(x))

>>> test(a)
0x10b1f5640                          # 地址依旧相同。
```

如果不希望对象被修改，那么需要不可变类型，或者是对象复制品。

不可变类型: `int`, `long`, `str`, `tuple`, `frozenset`

除了某些类型自带的 **copy** 方法外，还可以：

- 使用标准库的 **copy** 模块，它支持深度复制。
- 对象序列化，比如标准库中的 **pickle**、**cPickle**、**marshal**。

下面的测试建议不要用数字等不可变对象，因为可能会被其内部的缓存和复用机制干扰。

```
>>> import copy

>>> x = object()
>>> l = [x]                          # 创建一个列表。
```

```

>>> l2 = copy.copy(l)           # 浅复制，仅复制对象自身，而不会递归复制其成员。
>>> l2 is l, l2[0] is x         # 可以看到复制列表的元素依然是原对象。
(False, True)

>>> l3 = copy.deepcopy(l)       # 深度复制，会递归复制所有深度成员。
>>> l3 is l, l3[0] is x         # 列表元素也被复制了。
(False, False)

```

循环引用等问题会影响 `deepcopy` 函数的运作，建议仔细阅读官方文档。

引用计数

Python 默认通过引用计数来管理对象的内存回收。当引用计数为 0 时，虚拟机将 "立即" 回收对象内存，要么将对应的 `block` 块标记为空闲，要么返还给操作系统。

我们可以用 `__del__` 监控对象被回收。

```

>>> class User(object):
...     def __del__(self):
...         print "Will be dead!"

>>> a = User()
>>> b = a

>>> import sys
>>> sys.getrefcount(a)
3

>>> del a           # 删除引用，计数减小。
>>> sys.getrefcount(b)
2

>>> del b           # 删除最后一个引用，计数器为 0，对象被回收。
Will be dead!

```

某些内置类型，比如小整数什么的，因为缓存的缘故，计数器永远不会为 0，直到进程结束时才由相应的虚拟机清理函数释放。

除了直接引用外，Python 还支持弱引用。允许在不增加引用计数，也就是不妨碍对象回收的情况下间接引用对象。(不是所有类型都支持弱引用，诸如 `list`、`dict`、`object` 这些，弱引用会引发异常，详情请参考标准库文档或本书第二部分)

我们改用弱引用回调监控对象回收。

```

>>> class User(object): pass

```

```

>>> a = User()

>>> import weakref

>>> def callback(r):                                # 回调函数会在原对象被回收时调用。
...     print "weakref object:", r
...     print "target object dead!"

>>> r = weakref.ref(a, callback)                    # 创建弱引用对象。

>>> import sys
>>> sys.getrefcount(a)                              # 可以看到弱引用没有导致目标对象引用计数增加。
2                                                  #      2 是因为 getrefcount 形参造成的。

>>> r() is a                                         # 透过弱引用可以访问原对象。
True

>>> del a                                           # 原对象回收, callback 被调用。
weakref object: <weakref at 0x10f99a368; dead>
target object dead!

>>> hex(id(r))                                     # 通过对比, 可以看到 callback 参数是弱引用对象。
'0x10f99a368'                                       #      因为原对象已经死亡。

>>> r() is None                                     # 此时, 弱引用只能返回 None。这样也能判断原对象死亡。
True

```

引用计数是一种简单直接, 并且十分高效的内存回收方式。大多数时候它都能很好地工作, 除了因循环引用造成的计数故障。简单明显的循环引用, 可以用弱引用打破这种循环关系。但在实际开发中, 循环引用的形成往往很复杂, 可能由 n 个对象间接形成一个大的循环体, 此时只有等待 GC 去回收了。

垃圾回收

事实上, Python 拥有两套垃圾回收机制。除了引用计数外, 还有一个专门处理循环引用的 GC。通常我们提到垃圾回收, 都是指这个 "Reference Cycle Garbage Collection"。

能引发循环引用问题的, 都是那种 "容器" 类对象, 比如 `list`、`set`、`object` 等。对于这类对象, 虚拟机在为其分配内存时, 会额外添加一个用于追踪的 `PyGC_Head`。这些被追踪对象会被添加到一个链上, 以便 GC 进行管理。

```

typedef union _gc_head {
    struct {
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        Py_ssize_t gc_refs;
    };
};

```

```

    } gc;
    long double dummy;
} PyGC_Head;

```

当然，这类对象不一定就非得要 GC 才能回收。如果不存在循环引用，自然是积极性更高的引用计数机制抢先给咔嚓掉。也就是说，只要保证不存在循环引用，理论上是可以禁用 GC 的。当执行某些密集运算时，临时关掉 GC 可能会有较大的性能提升。

```

>>> import gc

>>> class User(object):
...     def __del__(self):
...         print hex(id(self)), "will be dead!"

>>> gc.disable()                # 关掉 GC

>>> a = User()
>>> del a                        # 对象正常回收，引用计数不会依赖 GC。
0x10fddf590 will be dead!

```

同 .NET、JAVA 一样，Python GC 同样将要回收的对象分成 3 级代龄。gen0 表示最年青的对象，也就是那些刚刚被盯上的家伙们。每级代龄都有一个最大容量阈值，当 gen0 对象数量超过其阈值时，将引发垃圾回收操作。

```

#define NUM_GENERATIONS 3

/* linked lists of container objects */
static struct gc_generation generations[NUM_GENERATIONS] = {
    /* PyGC_Head,                threshold,    count */
    {{GEN_HEAD(0), GEN_HEAD(0), 0}},    700,      0},
    {{GEN_HEAD(1), GEN_HEAD(1), 0}},    10,       0},
    {{GEN_HEAD(2), GEN_HEAD(2), 0}},    10,       0},
};

```

回收从 gen2 开始检查，如果阈值被突破，那么开始合并 gen2、gen1、gen0 几个追踪链表，将存活的对象提升代龄，而那些可回收对象则被打破循环引用，放到一个专门的列表等待回收。

```

>>> gc.get_threshold()          # 获取各级代龄阈值
(700, 10, 10)
>>> gc.get_count()              # 各级代龄链表跟踪的对象数量
(203, 0, 5)

```

需要特别注意的就是那些包含 `__del__` 的对象，因为回收前必须调用该方法。而且它们还会牵连到被其引用的对象，造成延迟释放。如果它同时存在循环引用，那么就永远不会被回收，直到进程终止。

这回我们不能偷懒用 `__del__` 监控对象回收了，改用 `weakref`。（貌似 IPython 对 GC 有些干扰，下面的测试代码建议在 Python 原生 shell 中测试）

```
>>> import gc, weakref

>>> class User(object): pass
>>> def callback(r): print r, "dead"

>>> gc.disable()                                # 停掉 GC，看看引用计数的能力。

>>> a = User(); wa = weakref.ref(a, callback)
>>> b = User(); wb = weakref.ref(b, callback)
>>> a.b = b; b.a = a                            # 形成循环引用关系。

>>> del a; del b                                # 删除名字引用。
>>> wa(), wb()                                  # 显然，计数机制对循环引用无效。
(<__main__.User object at 0x1045f4f50>, <__main__.User object at 0x1045f4f90>)

>>> gc.enable()                                # 开启 GC。
>>> gc.isenabled()                              # 可以用 isenabled 确认。
True

>>> gc.collect()                                # 因为没有达到阈值，我们手工启动回收。
<weakref at 0x1045a8cb0; dead> dead              # GC 的确有对付基友的能力。
<weakref at 0x1045a8db8; dead> dead              # 这个地址是弱引用对象的，别犯糊涂。
```

可一旦有了 `__del__`，GC 就拿循环引用没办法了。

```
>>> import gc, weakref

>>> class User(object):
...     def __del__(self): pass                  # 难道连空的 __del__ 也不行?

>>> def callback(r): print r, "dead!"

>>> gc.set_debug(gc.DEBUG_STATS | gc.DEBUG_LEAK) # 输出更详细的回收状态信息。
>>> gc.isenabled()                              # 确保 GC 在工作。
True

>>> a = User(); wa = weakref.ref(a, callback)
>>> b = User(); wb = weakref.ref(b, callback)
>>> a.b = b; b.a = a

>>> del a; del b
>>> gc.collect()                                # 从输出信息看，回收失败。
gc: collecting generation 2...
gc: objects in each generation: 520 3190 0
gc: uncollectable <User 0x10fd51fd0>            # a
```

```
gc: uncollectable <User 0x10fd57050>          # b
gc: uncollectable <dict 0x7f990ac88280>        # a.__dict__
gc: uncollectable <dict 0x7f990ac88940>        # b.__dict__
gc: done, 4 unreachable, 4 uncollectable, 0.0014s elapsed.
4

>>> wa(), wb(), hex(id(wa().__dict__)), hex(id(wb().__dict__)) # 和上面的地址对照一下。
(
    <__main__.User object at 0x10fd51fd0>,
    <__main__.User object at 0x10fd57050>,
    '0x7f990ac88280',
    '0x7f990ac88940'
)
```

关于用不用 `__del__` 的争论很多。多数情况，人们的结论是一棍子打死，诸多 "牛人" 也是这样教导新手的。可毕竟 `__del__` 承担了析构函数的角色，某些时候还是有其特定的作用的。用弱引用回调会造成逻辑分离，不便于维护。对于一些简单的脚本，我们还是能保证避免循环引用的，那不妨试试。就像前面例子中用来监测对象回收，就很方便.....

1.5 编译

Python 实现了栈式虚拟机 (Stack-based VM) 架构，通过机器无关的字节码来实现跨平台执行。这种字节码指令集没有寄存器概念，完全以栈 (抽象层面) 完全指令运算。尽管很简单，但对大多数人而言，无需关心这些细节。

要运行 Python 语言编写的程序，必须将源码编译成字节码。通常情况下，编译器会将源码转换成字节码后保存在 `pyc` 文件中。还可以用 `-O` 参数生成 `pyo` 格式，这是一种简单优化的 `pyc` 文件。

编译通常发生在模块载入那一刻。具体来看，又分为 `pyc` 和 `py` 两种情况。

载入 `pyc` 流程：

- 核对文件 Magic 标记。
- 检查时间戳和源码文件修改时间是否相同，以确定是否需要重新编译。
- 载入模块。

如果没有 `pyc`，那么就需要先完成编译：

- 对源码进行 AST 分析。
- 将分析结果编译成 `PyCodeObject`。
- 将 Magic、源码文件修改时间、`PyCodeObject` 保存到 `pyc` 文件中。
- 载入模块。

Magic 是一个特殊的数字，由 **Python** 版本号计算得来，作为 **pyc** 文件检查标记。**PyCodeObject** 则包含了成员对象的完整信息。

```
typedef struct {
    PyObject_HEAD
    int co_argcount;           // 参数个数，不包括 *args, **kwargs。
    int co_nlocals;           // 局部变量数量。
    int co_stacksize;         // 执行所需的栈空间。
    int co_flags;              // 编译标志，在创建 Frame 时用得着。
    PyObject *co_code;         // 字节码指令。
    PyObject *co_consts;       // 常量列表。
    PyObject *co_names;        // 符号列表。
    PyObject *co_varnames;     // 局部变量名列表。
    PyObject *co_freevars;     // 为闭包准备的东西...
    PyObject *co_cellvars;     // 还是闭包要的东西...。
    PyObject *co_filename;     // 源码文件名。
    PyObject *co_name;         // PyCodeObject 的名字，函数名、类名什么的。
    int co_firstlineno;        // 这个 PyCodeObject 在源码文件中的起始位置，也就是行号。
    PyObject *co_lnotab;       // 字节码指令偏移量和源码行号的对应关系，反汇编时用得着。
    void *co_zombieframe;      // 为优化准备的特殊 Frame 对象。
    PyObject *co_weakreflist;  // 为弱引用准备的...
} PyCodeObject;
```

无论是 **module** 还是其内部的 **function**，都被编译成 **PyCodeObject** 对象。内部成员都嵌套到 **co_consts** 列表中。

```
>>> pycat test.py
"""
    Hello, World!
"""

def add(a, b):
    return a + b

c = add(10, 20)

>>> code = compile(open("test.py").read(), "test.py", "exec")

>>> code.co_filename, code.co_name, code.co_names
('test.py', '<module>', ('__doc__', 'add', 'c'))

>>> code.co_consts
('\n    Hello, World!\n', <code object add at 0x105b76e30, file "test.py", line 5>, 10,
20, None)

>>> add = code.co_consts[1]
>>> add.co_varnames
('a', 'b')
```

手工编译代码，除了内置 `compile` 函数，标准库里还有 `py_compile`、`compileall` 可供选择。

```
>>> import py_compile, compileall

>>> py_compile.compile("test.py", "test.pyo")
>>> ls
main.py*      test.py      test.pyo

>>> compileall.compile_dir(".", 0)
Listing . ...
Compiling ./main.py ...
Compiling ./test.py ...
```

如果对 `pyc` 文件格式有兴趣，但又不想看 C 代码，可以到 `/usr/lib/python2.7/compiler` 目录里寻宝。又或者你对“反汇编”、“代码混淆”、“代码注入”、“破解”等话题更有兴趣，不妨看看标准库里的 `dis`，或者找本《蟒原》看看。

1.6 执行

相比 .NET、JAVA 的 CodeDOM 和 Emit，Pythoner 你就偷着乐吧。

最简单的就是 `eval()`，用来执行一个表达式。

```
>>> eval("(1 + 2) * 3")      # 假装看不懂这是啥.....
9

>>> eval("{'a': 1, 'b': 2}") # 将字符串转换为 dict。
{'a': 1, 'b': 2}
```

`eval` 默认会使用当前环境的名字空间，当然我们也可以带入自定的字典。

```
>>> x = 100
>>> eval("x + 200")          # 使用当前上下文的名字空间。
300

>>> ns = dict(x = 10, y = 20)
>>> eval("x + y", ns)        # 使用自定义名字空间。
30

>>> ns.keys()                 # 名字空间里多了 __builtins__。
['y', 'x', '__builtins__']
```

要执行一个代码片段，或者是一个 `PyCodeObject` 对象，那么需要动用 `exec`。同样可以带入自定义名字空间，以免对当前环境造成污染。

```
>>> py = """
```

```

... class User(object):
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return "<User: {0:x}; name={1}>".format(id(self), self.name)
... """

>>> ns = dict()
>>> exec py in ns                # 执行一个代码片段，使用自定义的名字空间。

>>> ns.keys()                    # 可以看到名字空间包含了新的类型：User。
['__builtins__', 'User']

>>> ns["User"]("Tom")           # 完全可用。貌似用来开发 ORM 会很简单。
<User: 10547f290; name=Tom>

```

继续看看 `exec` 执行 `PyCodeObject` 的演示。

```

>>> py = """
... def incr(x):
...     global z
...     z += x
... """

>>> code = compile(py, "test", "exec")    # 编译成 PyCodeObject。

>>> ns = dict(z = 100)                    # 自定义一个 global 名字空间。
>>> exec code in ns                       # exec 执行以后，全局名字空间多了 incr。

>>> ns.keys()                             # def 的意思是创建一个函数对象。
['__builtins__', 'incr', 'z']

>>> exec "incr(x); print z" in ns, dict(x = 50)    # 试着调用这个 incr，不过这次我们提供一个
150                                                # local 名字空间，以免污染 global。
>>> ns.keys()                             # 污染没有发生。
['__builtins__', 'incr', 'z']

```

动态执行一个 `py` 文件，可以考虑用 `execfile()`，或者 `runpy` 模块。

提示：

对 Python 基本环境有所了解，更有助于理解后续内容。实在看不明白，也没关系，等过些日子再回过头翻翻就行了。

第 2 章 内置类型

按照用途不同，Python 内置类型可分为 "数据" 和 "程序" 两大类。

数据结构：

- 空值: None
- 数字: bool, int, long, float, complex
- 序列: str, unicode, list, tuple
- 字典: dict
- 集合: set, frozenset

2.1 数字

bool

None、0、空字符串、以及没有元素的容器对象都被视为 False，反之为 True。

```
>>> map(bool, [None, 0, "", u"", list(), tuple(), dict(), set(), frozenset()])
[False, False, False, False, False, False, False, False, False]
```

虽然有点古怪，但 True、False 的确可以当数字使用。

```
>>> int(True)
1
>>> int(False)
0
>>> range(10)[True]
1
>>> x = 5
>>> range(10)[x > 3]
1
```

int

在 64 位平台上，int 最大能存储的数字是 sys.maxint (9223372036854775807)，这显然能对付绝大多数情况。整数是虚拟机特殊照顾对象：

- 从 Heap (不是 arena) 上申请名为 PyIntBlock 的内存块 (1KB)，可存储 41 个 int 对象。
- 多个 PyIntBlock 构成链，所有空闲块用链表组织起来，便于快速获取可用位置。
- 使用专门数组缓存 [-5, 257) 之间的小数字，只需计算下标就能获得指针。

- 大数字同样从 `PyIntBlock` 中获取存储空间，不足时再次申请 `PyIntBlock`。
- `PyIntBlock` 内存不用时不会返还给操作系统，直至进程结束。

看看 "小数字" 和 "大数字" 的区别：

```
>>> a = 15
>>> b = 15
>>> a is b
True
>>> sys.getrefcount(a)
47

>>> a = 257
>>> b = 257
>>> a is b
False
>>> sys.getrefcount(a)
2
```

因为 `PyIntBlock` 内存只复用不回收，试想持有大量数字对象会有什么后果？

用 `range` 创建一个巨大的数字列表，这就需要足够多的 `PyIntBlock` 为数字对象提供存储空间。就算稍后数字对象被回收，这些已经分配的 `PyIntBlock` 内存却不会归还给操作系统，于是就变相发生内存泄露了。但换成 `xrange` 就不同了，每次迭代后，前一数字对象被回收，其占用内存空闲出来以便复用，内存也就不会暴涨了。

运行下面代码前，必须先安装 `psutil` 包。

```
$ sudo easy_install -U psutil

$ cat test.py
#!/usr/bin/env python

import gc, os, psutil

def test():
    x = 0
    for i in range(10000000):    # xrange
        x += i

    return x

def main():
    print test()
    gc.collect()

    p = psutil.Process(os.getpid())
```

```
print p.get_memory_info()

if __name__ == "__main__":
    main()
```

对比 `range` 和 `xrange` 所需的 RSS 值。

```
range: meminfo(rss=93339648L, vms=2583552000L)    # 89 MB
xrange: meminfo(rss=8638464L, vms=2499342336L)    # 8 MB
```

通常情况下，大可不必担心。数字对象在回收后，其占用的内存就空闲出来，留待下次分配使用。除非有意为之，否则 `PyIntBlock` 并不会无限增长。

long

当 `int` 撑不住时，`long` 会自动替换上场。`long` 是变长对象，只要内存足够，你能创建无法想象的天文数字。

```
>>> a = sys.maxint
>>> type(a)
<type 'int'>

>>> b = a + 1
>>> type(b)
<type 'long'>

>>> 1 << 3000
12302319221611....1374723998766005827579300723253474890612250135171889174899079911291512
399773872178519018229989376L

>>> sys.getsizeof(1 << 0xFFFFFFFF)
572662332
```

`long` 出场的机会不多，`Python` 也就没有专门为其设计优化策略。

float

`float` 默认使用双精度表示，可能不能“精确”表示某些十进制的小数值。尤其是 `round` 操作结果，可能和我们预想的不同。

```
>>> 3 * 0.1 == 0.3
False

>>> round(2.675, 2)
2.67
```


可以用 `Decimal` 代替，它能精确控制运算精度、有效数位和 `round` 的结果。

```
>>> from decimal import Decimal, ROUND_UP, ROUND_DOWN

>>> float('0.1') * 3 == float('0.3')
False

>>> Decimal('0.1') * 3 == Decimal('0.3')
True

>>> round(2.675, 2)
2.67

>>> Decimal('2.675').quantize(Decimal('.01'), ROUND_UP)
Decimal('2.68')

>>> Decimal('2.675').quantize(Decimal('.01'), ROUND_DOWN)
Decimal('2.67')
```

在内存管理上，`float` 也采用 `PyFloatBlock` 模式，除了没有 "小浮点数" 一说外，和 `int` 基本相同。

2.2 字符串

与字符串相关的问题总是很多，诸如池化 (intern)、编码 (encode) 等。在 Python 中，字符串是不可变类型。其中 `str` 是 C 类型字符串，用来保存字符序列或二进制数据。

- 存储在 `arena` 区域，`str`、`unicode` 单字符都会被永久缓存。
- `str` 直接分配内存，`unicode` 则保留 1024 个宽字符长度小于 9 的复用内存块。
- 对象内部包含 `hash` 值，`str` 另有标记用来判断是否被池化。

字符串常量定义简单自由。

```
>>> "It's a book."                # 双引号里面可以用单引号。
"It's a book."

>>> 'It\'s a book.'              # 转义
"It's a book."

>>> '{"name":"Tom"}'             # 单引号里面正常使用双引号。
'{"name":"Tom"}'

>>> """                          # 多行
... line 1
... line 2
... """
```

```

>>> r"abc\x"                                     # r 前缀定义非转义的 raw-string。
'abc\x'

>>> "a" "b" "c"                                   # 合并多个相邻的字符串。
'abc'

>>> "中国人"                                       # UTF-8 字符串
'\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba'

>>> type(s), len(s)
(<type 'str'>, 9)

>>> u"中国人"                                       # UNICODE 字符串
u'\u4e2d\u56fd\u4eba'

>>> type(u), len(u)
(<type 'unicode'>, 3)

```

基本操作:

```

>>> "a" + "b"
'ab'

>>> "a" * 3
'aaa'

>>> ",".join(["a", "b", "c"])                     # 合并多个字符串。
'a,b,c'

>>> "a,b,c".split(",")                             # 按指定字符分割。
['a', 'b', 'c']

>>> "a\nb\r\nc".splitlines()                       # 按行分割。
['a', 'b', 'c']

>>> "a\nb\r\nc".splitlines(True)                   # 分割后, 保留换行符。
['a\n', 'b\r\n', 'c']

>>> "abc".startswith("ab"), "abc".endswith("bc")   # 判断是否以特定子串开始或结束。
(True, True)

>>> "abc".upper(), "Abc".lower()                   # 大小写转换。
('ABC', 'abc')

>>> "abcabc".find("bc"), "abcabc".find("bc", 2)    # 可指定查找起始结束位置。
(1, 4)

>>> " abc".lstrip(), "abc ".rstrip(), " abc ".strip() # 剔除前后空格。

```

```

('abc', 'abc', 'abc')

>>> "abc".strip("ac")                                # 可删除指定的前后缀字符。
'b'

>>> "abcbabc".replace("bc", "BC")                    # 可指定替换次数。
'aBCaBC'

>>> "a\tbc".expandtabs(4)                             # 将 tab 替换成空格。
'a   bc'

>>> "123".ljust(5, '0'), "456".rjust(5, '0'), "abc".center(10, '*')    # 填充
('12300', '00456', '***abc***')

>>> "123".zfill(6), "123456".zfill(4)                  # 数字填充
('000123', '123456')

```

编码

不知什么原因，Python 2.x 的默认编码是 ASCII，而非当前操作系统的编码。因为这个莫名其妙的设置，导致麻烦迭出。为了正确完成编码转换，需将两者统一起来。

```

>>> import sys
>>> sys.getdefaultencoding()
'ascii'

>>> import locale
>>> c = locale.getdefaultlocale(); c    # 获取当前系统编码。
('zh_CN', 'UTF-8')

>>> reload(sys)                                       # setdefaultencoding 在被初始化时被 site.py 删掉了。
<module 'sys' (built-in)>

>>> sys.setdefaultencoding(c[1])                    # 重新设置默认编码。
>>> sys.getdefaultencoding()                         # OK
'UTF-8'

```

str、unicode 都提供了 encode 和 decode 转换方法。

- encode: 将默认编码转换为其他编码。
- decode: 将默认或者指定编码字符串转换为 unicode。

```

>>> s = "中国人"; s
'\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba'

>>> u = s.decode(); u                                # UTF-8 -> UNICODE
u'\u4e2d\u56fd\u4eba'

```

```

>>> gb = s.encode("gb2312"); gb          # UTF-8 -> GB2312
'\xd6\xd0\xb9\xfa\xc8\xcb'

>>> gb.encode("utf-8")                    # encode 会把 gb 当做默认 UTF-8 编码，所以出错。
UnicodeDecodeError: 'utf8' codec can't decode byte 0xd6 in position 0: invalid
continuation byte

>>> gb.decode("gb2312")                   # 可以将其转换成 UNICODE。
u'\u4e2d\u56fd\u4eba'

>>> gb.decode("gb2312").encode()          # 然后再转换成 UTF-8
'\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba'

>>> unicode(gb, "gb2312")                 # GB2312 -> UNICODE
u'\u4e2d\u56fd\u4eba'

>>> u.encode()                           # UNICODE -> UTF-8
'\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba'

>>> u.encode("gb2312")                    # UNICODE -> GB2312
'\xd6\xd0\xb9\xfa\xc8\xcb'

```

标准库另有 `codecs` 模块用来处理更复杂的编码转换，比如大小端和 BOM。

```

>>> from codecs import BOM_UTF32_LE

>>> s = "中国人"
>>> s
'\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba'

>>> s.encode("utf-32")
'\xff\xfe\x00\x00-N\x00\x00\xfdV\x00\x00\xbaN\x00\x00'

>>> BOM_UTF32_LE
'\xff\xfe\x00\x00'

>>> s.encode("utf-32").decode("utf-32")
u'\u4e2d\u56fd\u4eba'

```

格式化

Python 提供了两种字符串格式化方法，除了比较熟悉的 C 样式外，还有更强大的 `format`。

```
%[(keyname)][flags][width][.precision]typecode
```

- `keyname`: 字典 key。

- flags: - 左对齐, + 数字符号, # 进制前缀, 或者用空格、0 填充。
- width: 宽度。
- precision: 小数位。
- typecode: 类型。

```
>>> "%(key)s=%(value)d" % dict(key = "a", value = 10)      # key
'a=10'

>>> "[%-10s]" % "a"                                       # 左对齐
'[a          ]'

>>> "%+d, %+d" % (-10, 10)                                 # 数字符号
'-10, +10'

>>> "%010d" % 3                                           # 填充
'0000000003'

>>> "%.2f" % 0.1234                                       # 小数位
'0.12'

>>> "%#x, %#X" % (100, 200)                               # 十六进制、前缀、大小写。
'0x64, 0XC8'

>>> "%s, %r" % (m, m)                                     # s: str(); r: repr()
'test..., <__main__.M object at 0x103c4aa10>'
```

format 方法提供了更多的控制项, 包括对列表、字典、对象成员的支持。

```
{fieldname!conversionflag:formatspec}
    formatspec: [[fill]align][sign][#][0][width][.precision][typecode]
```

- fieldname: 序号、参数名, 键, 对象成员。
- conversionflag: r repr(), s str()。
- formatspec: 和 C 格式类似。

```
>>> "{key}={value}".format(key="a", value=10)             # 使用命名参数。
'a=10'

>>> "{0},{1},{0}".format(1, 2)                             # fieldname 可多次使用。
'1,2,1'

>>> "{0:,}".format(1234567)                                # 千分位符号
'1,234,567'

>>> "{0:,.2f}".format(12345.6789)                          # 千分位, 带小数位。
'12,345.68'
```

```
>>> "[{0:<10}], [{0:^10}], [{0:*>10}].format("a") # 左中右对齐, 可指定填充字符。
'[a          ], [   a          ], [*****a]'

>>> import sys
>>> "{0.platform}".format(sys) # 成员
'darwin'

>>> "{0[a]}".format(dict(a=10, b=20)) # 字典
'10'

>>> "{0[5]}".format(range(10)) # 列表
'5'
```

大段的文本, 可以使用 `string.Template`。 `string` 模块中还定义了各种常见的字符序列。

```
>>> from string import letters, digits, Template

>>> letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> digits
'0123456789'

>>> Template("$name, $age").substitute(name = "User1", age = 20)
'User1, 20'

>>> Template("${name}, $age").safe_substitute(name = "User1") # 没找到值, 不会抛出异常。
'User1, $age'
```

池化

在 `Python` 进程中, 无数的对象拥有一堆类似 `"__name__"`、`"__doc__"` 这样的名字, 池化有助于提升性能, 减少内存消耗。

如果想把运行期动态生成的字符串放到池中, 可以用 `intern()` 函数。

```
>>> s = "".join(["a", "b", "c"])

>>> s is "abc" # 显然动态生成的字符串 s 没有被池化。
False

>>> intern(s) is "abc" # intern 会检查内部标记, 如果未被池化, 就收纳进去当小妾。
True

>>> intern(s) is intern(s) # 以后用 intern 从池中获取字符串对象, 就可以复用了。
True
```

注意：当丢到池中的字符串不再有外部引用时，是会被回收的。

2.3 列表

列表 (list) 更接近于 **Vector**，而非数组或链表。支持插入、删除元素操作。

- 当 `len > 0` 时，单独在 **Heap** 上分配一个数组用来存储元素指针。
- 默认会缓存 80 个复用对象，但元素项数组内存会被释放。
- 根据元素数量，动态扩大或收缩数组大小，预分配内存多于实际元素数量。

基本操作：

```
>>> []                                # 空列表。
[]

>>> ['a', 'b'] * 3                     # 这个少见吧。
['a', 'b', 'a', 'b', 'a', 'b']

>>> ['a', 'b'] + ['c', 'd']           # 算是运算符重载。
['a', 'b', 'c', 'd']

>>> list(xrange(10))                   # 将迭代器转换为列表。
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list("abcd")                       # 字符串也是序列类型。
['a', 'b', 'c', 'd']

>>> l = list("abc"); l[1] = 2; l       # 按序号读写。
['a', 2, 'c']

>>> l = list(xrange(10)); l[2:-2]      # 切片。
[2, 3, 4, 5, 6, 7]

>>> l = list("abcabc"); l.count("b")  # 统计元素项。
2

>>> l = list("abcabc"); l.index("a", 2) # 从指定位置查找项，返回序号。
3

>>> l = list("abc"); l.append("d"); l  # 追加元素。
['a', 'b', 'c', 'd']

>>> l = list("abc"); l.insert(1, 100); l # 在指定位置插入元素。
['a', 100, 'b', 'c']

>>> l = list("abc"); l.extend(range(3)); l # 合并列表。
['a', 'b', 'c', 0, 1, 2]
```

```
>>> l = list("abcabc"); l.remove("b"); l          # 移除第一个指定元素。
['a', 'c', 'a', 'b', 'c']

>>> l = list("abc"); l.pop(1), l                 # 弹出指定位置的元素（默认最后项）。
('b', ['a', 'c'])
```

对于有序列表，用 `bisect` 插入元素时，可保持其有序状态。

```
>>> import bisect
>>> l = ["a", "d", "c", "e"]

>>> bisect.insort(l, "b"); l
['a', 'b', 'c', 'd', 'e']

>>> bisect.insort(l, "d"); l
['a', 'b', 'c', 'd', 'd', 'e']
```

性能

尽管预分配内存要多于实际所需，但调用 `realloc()` 调整内存大小时，依然存在性能隐患。更何况插入和删除操作，还需要循环移动后续元素。对于频繁增删的大个列表，建议使用链表代替。或者像数组那样，预分配一个足够大的列表，然后用索引号设置数据。

下面的例子测试了两种创建 `list` 对象方式的性能差异，为了获得更好的测试结果，我们关掉 GC，元素使用同一个小整数对象。

```
>>> import itertools, gc
>>> gc.disable()

>>> def test(n):
...     return len([0 for i in xrange(n)])          # 先创建 list 对象，然后 append。
...
>>> def test2(n):
...     return len(list(itertools.repeat(0, n)))    # 按照 iter 创建 list 对象。
...
>>> timeit test(10000)
1000 loops, best of 3: 810 us per loop

>>> timeit test2(10000)
10000 loops, best of 3: 89.5 us per loop
```

从测试结果来看，性能差异非常大。

某些时候，可以考虑用 `array` 代替 `list`。和 `list` 总是存储对象指针不同，`array` 像 C 那样直接内嵌数据，既省了对象头等内存开销，又提升了读写效率。


```

>>> import array

>>> a = array.array("l", range(10)); a      # 用其他序列类型初始化数组。
array('l', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a.tolist()                             # 转换为列表。
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> a = array.array("c")                   # 创建特定类型数组。

>>> a.fromstring("abc"); a                 # 从字符串添加元素。
array('c', 'abc')

>>> a.fromlist(list("def")); a             # 从列表添加元素。
array('c', 'abcdef')

>>> a.extend(array.array("c", "xyz")); a    # 合并列表或数组。
array('c', 'abcdefxyz')

```

2.4 元组

元组 (tuple) 看上去像列表的只读版本，但在底层实现上有很多不同之处。

- 因为只读，所以包括元素指针数组在内的内存是连续分配的。
- 系统会缓存 `len < 20` 的对象，以便复用。
- 缓存对象按照 `len` 大小添加到对应链表，取用非常方便。
- 每个缓存链表最多可以管理 2000 个可复用对象。

尽可能用 `tuple` 替代 `list`，除内存复用更高效外，其只读特征更利于并行开发。

基本操作：

```

>>> a = (4); type(a)                       # 少了逗号，就成了普通的括号运算符了。
<type 'int'>

>>> a = (4,); type(a)                     # 这才是元组。
<type 'tuple'>

>>> s = tuple("abcdef"); s                # 将其他序列类型转换成元组。
('a', 'b', 'c', 'a', 'd', 'e', 'f')

>>> s.count("a")                          # 元素统计。
2

>>> s.index("d")                          # 查找元素，返回序号。
4

```

标准库另提供过了 `namedtuple`，可以按名字访问元素项。

```
>>> from collections import namedtuple

>>> User = namedtuple("User", "name age")

>>> u = User("user1", 10)
>>> u.name, u.age
('user1', 10)
```

其实 `namedtuple` 并不是 `tuple`，不过是利用模板动态构建的自定义 `class`。

2.5 字典

字典 (`dict`) 是极重要的数据类型，因为名字空间就是通过它实现的。`dict` 采用开放地址法的哈希表 (`hashtable`) 实现。

- 自带元素容量为 8 的 `smalltable`，只有超出时才额外在 `Heap` 上分配内存。
- 系统缓存 80 个 `dict` 复用对象，但其在 `Heap` 分配的 `Entry` 内存会被释放。
- 按照需要，动态调整容量。扩容或收缩都将重新分配内存，重新哈希。
- 删除操作不会收缩内存。

基本操作：

```
>>> {} # 空字典
{}

>>> {"a":1, "b":2} # 普通构造方式
{'a': 1, 'b': 2}

>>> dict(a = 1, b = 2) # 构造
{'a': 1, 'b': 2}

>>> dict(["a", 1], ["b", 2]) # 用两个序列类型构造字典。
{'a': 1, 'b': 2}

>>> dict(zip("ab", range(2))) # 同上
{'a': 0, 'b': 1}

>>> dict(map(None, "abc", range(2))) # 同上
{'a': 0, 'c': None, 'b': 1}

>>> dict.fromkeys("abc", 1) # 用序列做 key，并提供默认 value。
{'a': 1, 'c': 1, 'b': 1}
```

```

>>> {k:v for k, v in zip("abc", range(3))}    # 利用生成表达式构造字典。
{'a': 0, 'c': 2, 'b': 1}

>>> d = {"a":1, "b":2}; "b" in d              # 判断是否包含 key。
True

>>> d = {"a":1, "b":2}; del d["b"]; d          # 删除 k/v。
{'a': 1}

>>> d = {"a":1}; d.update({"c": 3}); d         # 合并 dict。
{'a': 1, 'c': 3}

>>> d = {"a":1, "b":2}; d.pop("b"), d         # 弹出 value。
(2, {'a': 1})

>>> d = {"a":1, "b":2}; d.popitem()           # 弹出 k/v。
('a', 1)

```

默认返回值:

```

>>> d = {"a":1, "b":2}

>>> d.get("c"), d.get("d", 123)                # 如果没有对应 key, 返回 None 或指定值。
(None, 123)

>>> d.setdefault("a", 100)                     # key 存在, 直接返回 value。
1

>>> d.setdefault("c", 200)                     # key 不存在, 先设置, 然后返回。
200

>>> d
{'a': 1, 'c': 200, 'b': 2}

```

迭代器操作:

```

>>> d = {"a":1, "b":2}

>>> d.keys()
['a', 'b']
>>> d.values()
[1, 2]
>>> d.items()
[('a', 1), ('b', 2)]

>>> for k in d: print k, d[k]
a 1
b 2

```

```
>>> for k, v in d.items(): print k, v
a 1
b 2
```

对于大字典，调用 `keys()`、`values()`、`items()` 会构造一个同样巨大的列表。建议用迭代器替代，以减少内存开销。

```
>>> d = {"a":1, "b":2}

>>> d.iterkeys(), d.itervalues(), d.iteritems()
(
    <dictionary-keyiterator object at 0x10de82cb0>,
    <dictionary-valueiterator object at 0x10de82d08>,
    <dictionary-itemiterator object at 0x10de82d60>
)

>>> for k, v in d.iteritems():
...     print k, v
...
a 1
b 2
```

视图

要判断两个 `dict` 间的差异，需要用到 `Dictionary View Object`。当 `dict` 发生变化时，`view` 会同步变更。

```
>>> d1 = dict(a = 1, b = 2)
>>> d2 = dict(b = 2, c = 3)

>>> d1 & d2
TypeError: unsupported operand type(s) for &: 'dict' and 'dict'

>>> v1 = d1.viewitems()
>>> v2 = d2.viewitems()

>>> v1 & v2                                     # 交集
set([('b', 2)])

>>> v1 | v2                                     # 并集
set([('a', 1), ('b', 2), ('c', 3)])

>>> v1 - v2                                     # 差集
set([('a', 1)])

>>> v1 ^ v2                                     # 差集（全部差集）
set([('a', 1), ('c', 3)])
```

```
>>> ('a', 1) in v1          # 判断
True
```

扩展

当访问的 `defaultdict` key 不存在时，自动调用 `factory` 对象创建 key/value。factory 可以是任何无参数函数或 callable 对象。

```
>>> from collections import defaultdict

>>> d = defaultdict(list)

>>> d["a"].append(1)          # key "a" 不存在，直接用 list() 创建一个空列表作为 value。
>>> d["a"].append(2)
>>> d["a"]
[1, 2]
```

`dict` 是哈希表，所以默认迭代是无序的。如果希望按照添加顺序输出结果，可以用 `OrderedDict`。

```
>>> from collections import OrderedDict

>>> d = dict()
>>> d["a"] = 1
>>> d["b"] = 2
>>> d["c"] = 3

>>> for k, v in d.items(): print k, v          # 并非按添加顺序输出。
a 1
c 3
b 2

>>> od = OrderedDict()
>>> od["a"] = 1
>>> od["b"] = 2
>>> od["c"] = 3

>>> for k, v in od.items(): print k, v        # 按添加顺序输出。
a 1
b 2
c 3

>>> od.popitem()                             # 按 LIFO 顺序弹出。
('c', 3)
>>> od.popitem()
('b', 2)
>>> od.popitem()
```

```
('a', 1)
```

2.6 集合

集合 (set) 用来存储无序的不重复对象。所谓不重复对象，除了不会包含同一对象的多重引用外，还包括 hash 相同的对象。也就是说 set 只能存储 hashable 对象。frozenset 是 set 的只读版本。

判重公式: (a is b) or (hash(a) == hash(b) and eq(a, b))

在内部实现上，set 和 dict 非常相似，只不过 Entry 没有 value 字段。集合不是序列类型，不能像 list 那样按序号访问，也不能做切片操作。

```
>>> s = set("abc"); s                                # 通过序列类型初始化。
set(['a', 'c', 'b'])

>>> {v for v in "abc"}                                # 通过构造表达式创建。
set(['a', 'c', 'b'])

>>> "b" in s                                           # 判断元素是否在集合中。
True

>>> s.add("d"); s                                       # 添加元素
set(['a', 'c', 'b', 'd'])

>>> s.remove("b"); s                                   # 移除元素
set(['a', 'c', 'd'])

>>> s.discard("a"); s                                  # 如果存在，就移除。
set(['c', 'd'])

>>> s.update(set("abcd")); s                           # 合并集合
set(['a', 'c', 'b', 'd'])

>>> s.pop(), s                                         # 弹出元素
('a', set(['c', 'b', 'd']))
```

set 和 dict、list 最大的不同除了 "不重复" 外，还支持集合运算。

```
>>> "c" in set("abcd")                                # 判断集合中是否有特定元素。
True

>>> set("abc") is set("abc")
False

>>> set("abc") == set("abc")                          # 相等判断
True
```

```

>>> set("abc") != set("abc")           # 不等判断
False

>>> set("abcd") >= set("ab")            # 超集判断: issuperset
True

>>> set("bc") < set("abcd")             # 子集判断: issubset
True

>>> set("abcd") | set("cdef")           # 并集: union
set(['a', 'c', 'b', 'e', 'd', 'f'])

>>> set("abcd") & set("abx")            # 交集: intersection
set(['a', 'b'])

>>> set("abcd") - set("ab")             # 差集: difference
set(['c', 'd'])
# 仅包括左参的内容

>>> set("abx") ^ set("aby")            # 差集: symmetric_difference
set(['y', 'x'])
# 包括两者的差集

>>> set("abcd").isdisjoint("ab")        # 判断是否没有交集
False

>>> s = set("abcd"); s |= set("cdef"); s # 并集, 设置: update
set(['a', 'c', 'b', 'e', 'd', 'f'])

>>> s = set("abcd"); s &= set("cdef"); s # 交集, 设置: intersection_update
set(['c', 'd'])

>>> s = set("abx"); s -= set("abcdy"); s # 差集, 设置: difference_update
set(['x'])
# 仅左参的内容

>>> s = set("abx"); s ^= set("aby"); s   # 差集, 设置: symmetric_difference_update
set(['y', 'x'])
# 包括两者的差集

```

dict key 和 set 都需要 hashable 类型的对象, 但 list、dict、set、defaultdict、OrderedDict 都是 unhashable 的。还好 tuple、frozenset 是可以的。

```

>>> hash([])
TypeError: unhashable type: 'list'

>>> hash({})
TypeError: unhashable type: 'dict'

>>> hash(set())
TypeError: unhashable type: 'set'

```

```
>>> hash(tuple()), hash(frozenset())
(3527539, 133156838395276)
```

而如果想将自定义类型放入 `set`，需要保证 `hash` 和 `equal` 的结果都相同，因此需 `override` 两个方法： `__hash__` 和 `__eq__`。

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

>>> hash(User("tom"))      # 每次的哈希结果都不同
279218517

>>> hash(User("tom"))
279218521

>>> class User(object):
...     def __init__(self, name):
...         self.name = name
...
...     def __hash__(self):
...         return hash(self.name)
...
...     def __eq__(self, o):
...         if not o or not isinstance(o, User): return False
...         return self.name == o.name

>>> s = set()

>>> s.add(User("tom"))
>>> s.add(User("tom"))

>>> s
set([<__main__.User object at 0x10a48d150>])
```

提示：

数据结构很重要，别紧着这几个内置类型打天下。

第 3 章 表达式

3.1 句法规则

Python 源码格式有点特殊。首先，可能因为出生年代久远的缘故，编译器默认编码采用 ASCII，而非当前通行的 UTF-8。其次，就是强制缩进格式让很多人“纠结”，甚至“望而却步”。

源文件编码

下面这样的错误，初学时很常见。究其原因，还是编译器默认将文件当成 ASCII 编码的缘故。

```
$ ./main.py
SyntaxError: Non-ASCII character '\xe4' in file ./main.py on line 4, but no encoding
declared; see http://www.python.org/peps/pep-0263.html for details
```

解决方法：在文件头部添加正确的编码标识。

```
$ cat main.py
#!/usr/bin/env python
#coding=utf-8

def main():
    print "世界末日"          # 玛雅人骗人，TNND！

if __name__ == "__main__":
    main()
```

也可以写成：

```
# -*- coding:utf-8 -*-
```

强制缩进

对于强制缩进，各有所好，没法强求。只不过到了 Python 这里就成了天条，半点违不得。多数时候，我们会建议初学者用 4 个空格代替 TAB。

唯一的麻烦就是从网页拷贝代码时，缩进丢失导致源码成了乱码。解决方法是：

- 像很多 C 程序员那样，在 block 尾部添加 "# end" 注释。
- 如果嫌不好看，可自定义一个 end 伪关键字。

```
#!/usr/bin/env python
#coding=utf-8
```

```
__builtins__.end = None          # 看这里，看这里.....

def test(x):
    if x > 0:
        print "a"
    else:
        print "b"
    end
end

def main():
    print "世界末日"
end

if __name__ == "__main__":
    main()
```

只要找到 **end**，就能确定 **code block** 的缩进范围了。

注释

注释从 **#** 开始，直到行尾，不支持跨行注释。

语句

可以用 **;** 将多条语句写在一行，或者用 **** 将一条语句写成多行。

```
>>> d = {}; d["a"] = 1; d.items()
[('a', 1)]

>>> for k, v in \
...     d.items():
...     print k, v

a 1
```

某些 **()**、**[]**、**{}** 之类的表达式无需 **** 就可写成多行。

```
>>> d = {
...     "a": 1,
...     "b": 2
... }

>>> d.pop("a",
...     2)

1
```

帮助

可以非常方便地为函数、模块和类添加帮助信息。

```
>>> def test():
...     """
...     func help
...     """
...     pass

>>> test.__doc__
'\n    func help\n    '

>>> class User(object):
...     """User Model"""
...
...     def __init__(self):
...         """user.__init__"""
...         pass

>>> User.__doc__
'User Model'

>>> User.__init__.__doc__
'user.__init__'
```

在 shell 用 `help()` 查看帮助信息，它会合并所有成员内容。

3.2 命名规则

命名规则不算复杂，只不过涉及私有成员命名时有点讲究。

- 必须以字母或下划线开头，只能是下划线、字母和数字的组合。
- 不能和语言保留字相同。
- 名字区分大小写。
- 模块中以下划线开头的名字视为私有，不会被 `"from <module> import *"` 导入。
- 以双下划线开头的类成员名字视为私有，将被自动重命名。
- 同时以双下划线开头和结尾的名字，通常是特殊成员。
- 单一下划线代表最后表达式的返回值。

```
>>> s = set("abc")

>>> s.pop()
'a'

>>> _
```

```
'a'

>>> s.pop()
'c'

>>> _
'c'
```

保留字 (包括 Python 3):

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

3.3 赋值

除非在函数中使用关键字 `global`、`nonlocal` 指明外部名字，否则赋值语句总是在当前名字空间创建或修改 `name/object` 关联。

与 C 以 `block` 为隔离，能在函数中创建多个同名变量不同，Python 函数所有代码共享同一个名字空间，于是会出现下面这样的状况。

```
>>> def test():
...     while True:
...         x = 10
...         break
...     print locals()
...     print x

>>> test()
{'x': 10}
10
```

支持通过序列类型或迭代器对多个变量赋值。

```
>>> a, b = "a", "b"
>>> a, b = "ab"
>>> a, b = [1, 2]
>>> a, b = xrange(2)
```

一旦右边的值多于名字数量，会引发异常。可用切片，或者 `"_"` 补位。

```
>>> a, b = "abc"
```

```
Traceback (most recent call last):
  a, b = "abc"
ValueError: too many values to unpack

>>> a, b, _ = "abc"
>>> a, b = "abc"[:2]
```

Python 3 对此提供了更好的支持。

```
Python 3.3.0 (default, Nov  4 2012, 20:26:43)

>>> a, *b, c = "a1234c"
>>> a, b, c
('a', ['1', '2', '3', '4'], 'c')
```

3.4 表达式

if

只需记住将 "else if" 换成 "elif" 即可。

```
>>> x = 10

>>> if x > 0:
...     print "+"
... elif x < 0:
...     print "-"
... else:
...     print "0"

+
```

可以改造得简单一些。

```
>>> x = 1
>>> print "+" if x > 0 else "-" if x < 0 else "0"
+

>>> x = 0
>>> print "+" if x > 0 else "-" if x < 0 else "0"
0

>>> x = -1
>>> print "+" if x > 0 else "-" if x < 0 else "0"
-
```

或者利用 and、or 条件短路，写得更简洁点。

```

>>> x = 1
>>> print (x > 0 and "+") or (x < 0 and "-") or "0"
+

>>> x = 0
>>> print (x > 0 and "+") or (x < 0 and "-") or "0"
0

>>> x = -1
>>> print (x > 0 and "+") or (x < 0 and "-") or "0"
-

```

在 Python 中可以将两次比较合并。

```

>>> x = 10
>>> if (5 < x <= 10): print "haha!"
haha!

```

条件表达式不能包含赋值语句，习惯此种写法的要调整一下了。

```

>>> if (x = 1) > 0: pass
      File "<ipython-input-4-bc2d73931d91>", line 1
        if (x = 1) > 0: pass
            ^
SyntaxError: invalid syntax

```

while

比我们熟悉的 while 多了一个 else 分支。如果没有 break 中断循环，那么 else 就会执行。

```

>>> x = 3
>>> while x > 0:
...     x -= 1
... else:
...     print "over!"

over!

>>> while True:
...     x += 1
...     if x > 3: break
... else:
...     print "over!"

```

利用 else 分支标记循环逻辑被完整处理是个不错的主意。

for

Python 的 for 更类似 foreach，用来处理序列和迭代器对象。

```
>>> for i in xrange(3): print i
0
1
2

>>> for k, v in {"a":1, "b":2}.items(): print k, v  # 多变量赋值
a 1
b 2

>>> d = ((1, ["a", "b"]), (2, ["x", "y"]))
>>> for i, (c1, c2) in d:                                # 多层展开
...     print i, c1, c2

1 a b
2 x y
```

同样有个 else 分支。

```
>>> for x in xrange(3):
...     print x
... else:
...     print "over!"

0
1
2
over!

>>> for x in xrange(3):
...     print x
...     if x > 1: break
... else:
...     print "over!"

0
1
2
```

要实现传统的 for 循环，需要借助 enumerate() 返回序号。

```
>>> for i, c in enumerate("abc"):
...     print "s[{0}] = {1}".format(i, c)

s[0] = a
```

```
s[1] = b
s[2] = c
```

pass

占位符，用来标记空代码块。

```
>>> def test():
...     pass

>>> class User(object):
...     pass
```

break / continue

break 中断循环，**continue** 开始下一次循环。

没有 **goto**、**label**，也别想用 **break**、**continue** 跳出多层嵌套循环了。

```
>>> while True:
...     while True:
...         flag = True
...         break
...     if "flag" in locals(): break
```

如果嫌 "跳出标记" 不好看，可以试试用异常。

```
>>> class BreakException(Exception): pass

>>> try:
...     while True:
...         while True:
...             raise BreakException()
... except BreakException:
...     print "越狱成功!"
```

Q.yuhen: 也没好看到哪去，不过好歹保持内部逻辑的干净。

del

可删除名字、序列元素、字典键值，以及对象成员。

```
>>> x = 1
>>> "x" in vars()
True

>>> del x
```



```

>>> "x" in vars()
False

>>> x = range(10); del x[1]; x
[0, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x = range(10); del x[1:5]; x          # 按切片删除
[0, 5, 6, 7, 8, 9]

>>> d = {"a":1, "b":2}; del d["a"]; d      # key 不存在时, 不会抛出异常。
{'b': 2}

>>> class User(object): pass
>>> o = User(); o.name = "user1"; hasattr(o, "name")
True

>>> del o.name
>>> hasattr(o, "name")
False

```

Generator

用一种优雅的方式创建列表、字典或集合。

```

>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> {x for x in range(10)}
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> {c:ord(c) for c in "abc"}
{'a': 97, 'c': 99, 'b': 98}

```

可带上条件进行过滤。

```

>>> [x for x in range(10) if x % 2]
[1, 3, 5, 7, 9]

```

用多个 for 子句实现嵌套。

```

>>> ["{0}{1}".format(c, x) for c in "abc" for x in range(3)]
['a0', 'a1', 'a2', 'b0', 'b1', 'b2', 'c0', 'c1', 'c2']

```

这相当于：

```

>>> n = []
>>> for c in "abc":
...     for x in range(3):

```

```
...     n.append("{0}{1}".format(c, x))
```

每个子句都可有条件表达式，内层还可引用外层对象。

```
>>> [{"0}{1}".format(c, x) \
...     for c in "aBcD" if c.isupper() \
...     for x in range(5) if x % 2 \
... ]
['B1', 'B3', 'D1', 'D3']
```

甚至可以在函数调用时，直接用来生成迭代器实参。

```
>>> def test(it):
...     for i, x in enumerate(it):
...         print "{0} = {1}".format(i, x)

>>> test(hex(x) for x in range(3))
0 = 0x0
1 = 0x1
2 = 0x2
```

3.5 运算符

这东西没啥好说的，只要记得没 "++"、"--" 就行。

运算符	说明
$x + y, x - y$	加减
$x * y, x / y$	乘除
$+x, -x$	正负
$x += y, x -= y$	
$x *= y, x /= y$	
$x // y$	整除
$x ** y$	幂
$x \% y$	取模
$x \& y, x y, x \wedge y$	位运算
$\sim x$	位取反
$x \ll y, x \gg y$	位移
$x > y, x \geq y$	比较
$x < y, x \leq y$	

运算符	说明
<code>x == y, x != y</code>	相等
<code>x is y, x is not y</code>	同一对象
<code>x in y, x not in y</code>	包含 (序列、字典、迭代器)
<code>not x</code>	非
<code>x and y, x or y</code>	布尔
<code>abs</code>	绝对值
<code>pow</code>	幂
<code>len</code>	元素数量
<code>min, max</code>	最小、最大元素
<code>divmod</code>	(商, 余数)
<code>sum</code>	统计 (可以带初始值)
<code>cmp</code>	比较

切片

序列类型支持 "切片 (slice)" 操作, 可以通过两个索引序号获取一个片段。

```
>>> x = range(10)
>>> x[2:6]
[2, 3, 4, 5]
```

支持大于 1 的步进。

```
>>> x[2:6:2]
[2, 4]
```

可以忽略起始或结束序号。

```
>>> x[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x[:6]
[0, 1, 2, 3, 4, 5]

>>> x[7:]
[7, 8, 9]
```

支持倒序。

```
>>> x[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

>>> x[7:3:-2]
[7, 5]
```

可以按切片范围删除序列片段。

```
>>> x = range(10)
>>> del x[4:8]; x
[0, 1, 2, 3, 8, 9]

>>> x = range(10)
>>> del x[::2]; x
[1, 3, 5, 7, 9]
```

布尔

`and` 返回短路时的最后一个值，`or` 返回第一个真值。要是没短路的话，自然是返回最后一个值。

```
>>> 1 and 2          # True: 最后一个值
2

>>> 1 and 2 and 0    # False: 最后一个值
0

>>> 1 and 0 and 2    # False: 第一个短路值 0
0

>>> 1 or 0           # True: 第一个真值 1
1

>>> 0 or [] or 1     # True: 第一个真值 1
1

>>> 0 or 1 or ["a"]  # True: 第一个真值 1
1
```

用 `and`、`or` 实现 "三元表达式 (?)" 很方便。

```
>>> x = 5
>>> print x > 0 and "A" or "B"
A
```

或者用 `or` 提供默认值。

```
>>> x = 5
>>> y = x or 0
```

```
>>> y
5

>>> x = None
>>> y = x or 0
>>> y
0
```

相等

操作符 "==" 可以被重载，因此不适合做对象判同。

```
>>> class User(object):
...     def __init__(self, name):
...         self.name = name
...     def __eq__(self, o):
...         if not o or not isinstance(o, User): return False
...         return cmp(self.name, o.name) == 0

>>> a, b = User("tom"), User("tom")

>>> a is b
False

>>> a == b
True
```

3.6 类型转换

各种类型和字符串之间的转换。

```
>>> str(123), bin(17), oct(20), hex(22)           # int
('123', '0b10001', '024', '0x16')

>>> int('123'), int('0b10001', 2), int('024', 8), int('0x16', 16)
(123, 17, 20, 22)

>>> ord('a'), chr(97), unichr(97)                 # char
(97, 'a', u'a')

>>> str(0.97), float("0.97")                       # float
('0.97', 0.97)

>>> str([0, 1, 2]), eval("[0, 1, 2]")              # list
('[0, 1, 2]', [0, 1, 2])

>>> str((0, 1, 2)), eval("(0, 1, 2)")              # tuple
```

```
('(0, 1, 2)', (0, 1, 2))

>>> str({'a':1, 'b':2}), eval("{'a': 1, 'b': 2}")    # dict
('{a': 1, 'b': 2}', {'a': 1, 'b': 2})

>>> str({1, 2, 3}), eval("{1, 2, 3}")                # set
('set([1, 2, 3])', set([1, 2, 3]))
```

3.7 常用函数

print

Python 2.7 可直接使用 `print` 表达式，Python 3 就只能用函数了。

```
>>> import sys

>>> print >> sys.stderr, "Error!", 456
Error! 456

>>> from __future__ import print_function

>>> print("Hello", "World", sep = ",", end = "\r\n", file = sys.stdout)
Hello,World
```

还可以用标准库中的 `pprint.pprint()` 代替 `print`，能看到更漂亮的输出结果。

input

`input` 会将输入的字符串进行 `eval` 处理，`raw_input` 直接返回用户输入的原始字符串。

```
>>> input("cmd> ")
cmd> 1+2+3
6

>>> raw_input("cmd> ")
cmd> 1+2+3
'1+2+3'
```

不过在 Python 3 中已经将 `raw_input` 重命名为 `input`。

另外，要输入密码时，请用标准库 `getpass`。

```
>>> from getpass import getpass, getuser

>>> pwd = getpass("%s password: " % getuser())
```

```
yuhen password:
```

```
>>> pwd  
'123456'
```

exit

`exit([status])` 调用所有退出函数后终止进程，并返回 `ExitCode`。

- 忽略或 `status = None`，表示正常退出，`ExitCode = 0`。
- `status = <number>`，表示 `ExitCode = <number>`。
- 返回其他对象表示失败，参数会被显示，`ExitCode = 1`。

```
$ cat main.py  
#!/usr/bin/env python  
#coding=utf-8  
  
import atexit  
  
def clean():  
    print "clean..."  
  
def main():  
    atexit.register(clean)  
    exit("Failure!")  
  
if __name__ == "__main__":  
    main()  
  
$ ./main.py  
Failure!  
clean...  
  
$ echo $?  
1
```

`sys.exit()` 和 `exit()` 完全相同。`os_exit()` 直接终止进程，不调用退出函数，且退出码必须是数字。

vars

获取 `locals` 或指定对象的 `__dict__`。

```
>>> vars() is locals()  
True  
  
>>> import sys
```

```
>>> vars(sys) is sys.__dict__  
True
```

dir

获取 **locals** 名字空间中的所有名字，或者指定对象的所有可访问成员 (包括基类)。

```
>>> set(locals().keys()) == set(dir())  
True
```

提示：

犯不着记，用得多了，自然就记住了.....

第 4 章 函数

当编译器遇到 `def`，会生成 "MAKE_FUNCTION" 指令。也就是说 `def` 是执行指令，而不仅仅是个语法关键字。如此，我们可以在任何地方使用 `def` 动态创建函数对象。`PyCodeObject` 包含了执行指令字节码，而 `PyFunctionObject` 则为其提供了状态信息。

函数声明：

```
def name([arg,... arg = value,... *arg, **arg]):
    suite
```

结构定义：

```
typedef struct {
    PyObject_HEAD
    PyObject *func_code;           // PyCodeObject
    PyObject *func_globals;        // 所在模块的全局名字空间
    PyObject *func_defaults;       // 参数默认值列表
    PyObject *func_closure;        // 闭包列表
    PyObject *func_doc;            // __doc__
    PyObject *func_name;           // __name__
    PyObject *func_dict;           // __dict__
    PyObject *func_weakreflist;    // 弱引用链表
    PyObject *func_module;         // 所在 Module
} PyFunctionObject;
```

4.1 创建

函数是第一类对象，可作为其他函数的实参和返回值。

- 函数在同一名字空间中不能 "重载 (override)"，因为 `__dict__[name]` 是唯一的。
- 函数总是有返回值。就算没有 `return`，默认也会返回 `None`。

```
>>> def test(name):
...     if name == "a":
...         def a(): pass
...         return a
...     else:
...         def b(): pass
...         return b
>>> test("a").__name__
'a'
```

不同于用 `def` 定义复杂函数，`lambda` 只能是拥有返回值的简单的表达式。使用赋值语句会引发语法错误，可以考虑用函数代替。

```
>>> add = lambda x, y = 0: x + y

>>> add(1, 2)
3
>>> add(3)
3

>>> map(lambda x: x % 2 and None or x, range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

和 C# 等语言相比，Python 的 lambda 有点弱，但也够用。在函数式编程中，大有用武之地。

4.2 参数

Python 函数的传参方式灵活多变，可按形参顺序传参，也可不关心顺序用命名实参。

```
>>> def test(a, b):
...     print a, b

>>> test(1, "a")                # 位置参数
1 a

>>> test(b = "x", a = 100)       # 命名参数
100 x
```

参数定义支持默认值。不过要小心，默认值对象是在 def 创建函数对象时生成，以后调用函数时重复使用该对象。如果该默认值是可变类型，那么就如同 C 静态局部变量。

```
>>> def test(x, ints = []):
...     ints.append(x)
...     return ints

>>> test(1)
[1]

>>> test(2)                      # 保持了上次调用状态。
[1, 2]

>>> test(1, [])                  # 显式提供实参，不使用默认值。
[1]

>>> test(3)                      # 再次使用默认值。
[1, 2, 3]
```

默认参数后面不能有其他位置参数，除非是变参。

```
>>> def test(a, b = 0, c): pass
SyntaxError: non-default argument follows default argument

>>> def test(a, b = 0, *args, **kwargs): pass
```

用 `*args` 收集 "多余" 的位置参数, `**kwargs` 收集 "额外" 的命名参数。用这两样个名字只是惯例, 可自由命名。

```
>>> def test(a, b, *args, **kwargs):
...     print a, b
...     print args
...     print kwargs

>>> test(1, 2, "a", "b", "c", x = 100, y = 200)
1 2
('a', 'b', 'c')
{'y': 200, 'x': 100}
```

变参只能放在所有参数定义的尾部, 且 `**kwargs` 还得保证是最后一个。

```
>>> def test(*args, **kwargs):                # 可以接收任意参数的函数。
...     print args
...     print kwargs

>>> test(1, "a", x = "x", y = "y")           # 位置参数, 命名参数。
(1, 'a')
{'y': 'y', 'x': 'x'}

>>> test(1)                                   # 仅传位置参数。
(1,)
{}

>>> test(x = "x")                             # 仅传命名参数。
()
{'x': 'x'}
```

可以 "展开" 序列类型和字典, 将其元素作为多个具体实参使用。如不展开的话, 那仅能当单个实参使用了。

```
>>> def test(a, b, *args, **kwargs):
...     print a, b
...     print args
...     print kwargs

>>> test(*range(1, 5), **{"x": "Hello", "y": "World"})
1 2
(3, 4)
{'y': 'World', 'x': 'Hello'}
```

单个 "*" 展开序列类型，或者仅是字典的键列表 (keys)，"*" 展开字典键值对 (items)。但如果没有变参收集，展开后多余的参数会引发异常。

```
>>> def test(a, b):
...     print a
...     print b

>>> d = dict(a = 1, b = 2)

>>> test(*d)                                # 仅展开 keys(), test("a"、"b")。
a
b

>>> test(**d)                               # 展开 items(), test(a = 1, b = 2)。
1
2

>>> d = dict(a = 1, b = 2, c = 3)

>>> test(*d)                                # 因为没有位置变参收集多余的 "c", 导致出错。
TypeError: test() takes exactly 2 arguments (3 given)

>>> test(**d)                               # 因为没有命名变参收集多余的 "c = 3", 导致出错。
TypeError: test() got an unexpected keyword argument 'c'
```

lambda 函数同样支持默认值和变参，使用方法完全一致。

```
>>> test = lambda a, b = 0, *args, **kwargs: \
...     sum([a, b] + list(args) + kwargs.values())

>>> test(1, *[2, 3, 4], **{"x": 5, "y": 6})
21
```

4.3 作用域

函数形参和内部变量都存储在 locals 名字空间中。

```
>>> def test(a, *args, **kwargs):
...     s = "Hello, World!"
...     print locals()

>>> test(1, "a", "b", x = 10, y = "hi")
{'a': 1,
 'args': ('a', 'b'),
 'kwargs': {'y': 'hi', 'x': 10}}
```

```
    's': 'Hello, World!',  
}
```

任何时候在函数内部使用赋值语句，修改的都不是外部同名变量，而是在 **locals** 名字空间中新建了一个对象关联。注意，“赋值”是指是将外部名字指向一个新的对象，而非通过外部名字改变对象状态。

```
>>> x = 10  
  
>>> hex(id(x))  
'0x7fb8e04105e0'  
  
>>> def test():  
...     x = "hi"  
...     print hex(id(x)), x  
  
>>> test()                                # 两个 x 指向不同的对象。  
0x10af2b490 hi  
  
>>> x                                      # 外部变量没有被修改。  
10
```

如果仅仅是引用外部变量，那么按 **LEGB** 顺序在不同作用域查找该名字。

名字查找顺序: **locals** -> **enclosing function** -> **module globals** -> **__builtins__**

- **locals**: 函数内部名字空间，包括局部变量和形参。
- **enclosing function**: 外部嵌套函数的名字空间。
- **module globals**: 函数定义所在模块的名字空间。
- **__builtins__**: 所有模块载入时都持有该模块，其中包含了内置类型和函数。

想想看，如果将变量放到 **__builtins__** 名字空间中，那么就可以在任何模块中直接访问，就如同内置函数那样。不过这似乎不是个好主意，不值得推荐。

```
>>> __builtins__.b = "builtins: b"  
  
>>> g = "globals: g"  
  
>>> def enclose():  
...     e = "enclosing: e"  
...     def test():  
...         l = "locals: l"  
...         print l  
...         print e  
...         print g  
...         print b  
... 
```

```

...     return test

>>> t = enclose()

>>> t()
locals: l
enclosing: e
globals: g
builtins: b

```

现在，获取外部空间的名称没问题了，但如果想将外部名称关联到一个新对象，该如何处理呢？为此，Python 2.7 提供了 `global` 关键字，指明修改的是 `module` 名字空间。Python 3 还额外提供了 `nonlocal` 修改外部嵌套函数名字空间，可惜 2.7 没有。

```

>>> x = 100

>>> hex(id(x))
0x7f9a9264a028

>>> def test():
...     global x, y
...     x = 1000          # 这个 x 是 globals 名字空间中的。
...     y = "Hello, World!" # 因为 globals 中没有 y，那么就新建一个名字。
...     print hex(id(x))

>>> test()                # 可以看到 test.x 引用的是外部变量 x。
0x7f9a9264a028

>>> x, y                  # globals 名字空间中出现了 y。
(1000, 'Hello, World!')

```

2.7 没有 `nonlocal` 多少有点麻烦，要实现类似功能稍微有点麻烦。

```

>>> from ctypes import pythonapi, py_object
>>> from sys import _getframe

>>> def nonlocal(**kwargs):
...     f = _getframe(2)
...     ns = f.f_locals
...     ns.update(kwargs)
...     pythonapi.PyFrame_LocalsToFast(py_object(f), 0)

>>> def enclose():
...     x = 10
...
...     def test():
...         nonlocal(x = 1000)
...

```

```
...     test()
...     print x

>>> enclose()
1000
```

这种实现通过 `_getframe()` 来获取外部函数堆栈帧名字空间，存在一些限制。因为拿到可能不是调用者，而非函数创建者。

4.4 闭包

闭包的意思是说，当函数离开创建环境后，依然持有其上下文状态。比如下面的 `a` 和 `b`，在离开 `test` 函数后，依然持有 `test.x` 变量。

```
>>> def test():
...     x = [1, 2]
...     print hex(id(x))
...
...     def a():
...         x.append(3)
...         print hex(id(x))
...
...     def b():
...         print hex(id(x)), x
...
...     return a, b

>>> a, b = test()
0x109b925a8                                # test.x

>>> a()
0x109b925a8                                # 指向 test.x

>>> b()
0x109b925a8 [1, 2, 3]
```

闭包实现原理很简单，以上例来解释：

`test` 在创建 `a` 和 `b` 时，将它们所引用的外部对象 `x` 添加到 `func_closure` 列表中。因为 `x` 引用计数增加了，所以就算 `test` 堆栈帧没有了，`x` 对象也不会被回收。

```
>>> a.func_closure
(<cell at 0x109e0aef8: list object at 0x109b925a8>,)

>>> b.func_closure
(<cell at 0x109e0aef8: list object at 0x109b925a8>,)

```

为什么用 `function.func_closure`，而不是堆栈帧的名字空间呢？那是因为 `test` 仅仅返回 `a`、`b` 两个函数对象，并没有调用它们，自然也不可能会为其分配堆栈帧。这样一来，这有个问题，每次返回的 `a` 和 `b` 都必须都是新建对象，否则这个闭包状态就被覆盖了。

```
>>> def test(x):
...     def a():
...         print x
...
...     print hex(id(a))
...     return a

>>> a1 = test(100)                                # 每次创建 a 都提供不同的参数。
0x109c700c8

>>> a2 = test("hi")                                # 可以看到两次返回的函数对象并不相同。
0x109c79f50

>>> a1()                                             # a1 的状态没有被 a2 破坏。
100

>>> a2()
hi

>>> a1.func_closure                                # a1、a2 持有的闭包列表是不同的。
(<cell at 0x109e0cf30: int object at 0x7f9a92410ce0>,)

>>> a2.func_closure
(<cell at 0x109d3ead0: str object at 0x109614490>,)

>>> a1.func_code is a2.func_code                    # 这个很好理解，字节码没必要有多个。
True
```

通过 `func_code`，我们可以获知闭包所引用的外部名字。`co_cellvars` 保存了被内部函数引用的名字，而 `co_freevars` 保存的则是当前函数引用外部的名字。

```
>>> test.func_code.co_cellvars                      # 被内部函数 a 引用的名字。
('x',)

>>> a.func_code.co_freevars                          # a 引用外部函数 test 中的名字。
('x',)
```

使用闭包，一定要注意“延迟获取”现象。看下面的例子：

```
>>> def test():
...     for i in range(3):
...         def a():
...             print i
...         yield a
```



```
>>> a, b, c = test()

>>> a(), b(), c()
2
2
2
```

为啥输出的都是 2 呢？

首先，`test` 只是返回函数对象，并没有执行。其次，`test` 完成 `for` 循环时，`i` 已经等于 2，所以执行 `a`、`b`、`c` 时，它们所持有 `i` 自然也就等于 2。

4.5 堆栈帧

Python 的堆栈帧基本上就是对 x86 的模拟，用指针来模拟 BP、SP、IP 寄存器。堆栈帧对象还包括了函数执行所需的名字空间、调用堆栈、异常状态等信息。

```
typedef struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;           // 调用堆栈 (Call Stack) 链表
    PyCodeObject *f_code;           // PyCodeObject
    PyObject *f_builtins;           // builtins 名字空间
    PyObject *f_globals;            // globals 名字空间
    PyObject *f_locals;             // locals 名字空间
    PyObject **f_valuelist;          // 和 f_stacktop 共同维护运行帧空间，相当于 RBP 寄存器。
    PyObject **f_stacktop;          // 运行栈顶，相当于 RSP 寄存器的作用。
    PyObject *f_trace;              // Trace function

    PyObject *f_exc_type, *f_exc_value, *f_exc_traceback; // 记录当前栈帧的异常信息

    PyThreadState *f_tstate;        // 所在线程状态
    int f_lasti;                    // 上一条字节码指令在 f_code 中的偏移量，类似 RIP 寄存器。
    int f_lineno;                   // 与当前字节码指令对应的源码行号

    ... ..

    PyObject *f_localsplus[1];      // 动态申请的一段内存，用来模拟 x86 堆栈帧所在内存空间。
} PyFrameObject;
```

为了获取堆栈帧，可以使用 `sys.getframe()` 或者 `inspect.currentframe()`。其中 `_getframe()` 深度参数为 0 表示当前函数，1 表示上一级调用函数。除了用于调试外，日常编程中可以利用堆栈帧做很多很有意思的事情。

权限管理

通过调用堆栈检查函数 `Caller`，以实现权限管理。

```
>>> def save():
...     f = _getframe(1)
...     if not f.f_code.co_name.endswith("_logic"): # 检查 Caller 名字，限制调用者身份。
...         raise Exception("Error!")           # 还可以检查更多信息。
...     print "ok"

>>> def test(): save()
>>> def test_logic(): save()

>>> test()
Exception: Error!

>>> test_logic()
ok
```

上下文

通过调用堆栈，我们可以隐式向整个流程传递上下文对象。虽然能用 `frame.f_back` 获取上级堆栈帧，但 `inspect.stack` 获取调用堆栈列表更方便一点。

```
>>> import inspect

>>> def get_context():
...     for f in inspect.stack():                # 循环调用堆栈列表。
...         context = f[0].f_locals.get("context") # 查看该堆栈帧名字空间中是否有目标。
...         if context: return context           # 找到了就返回，并终止查找循环。

>>> def controller():
...     context = "ContextObject"                # 将 context 添加到 locals 名字空间。
...     model()

>>> def model():
...     print get_context()                      # 通过调用堆栈查找 context。

>>> controller()                                # 测试通过。
ContextObject
```

在多线程环境，可以用 `sys._current_frames` 则返回所有线程的当前堆栈帧对象。和往常一样，虚拟机还会缓存 200 个 `PyFrameObject` 复用对象，以获得更好的执行性能。整个程序跑下来，天知道有多少个 `frame` 对象。

4.6 包装

用 `functools.partial()` 可以将函数包装成更简洁的版本。

```
>>> from functools import partial

>>> def test(a, b, c):
...     print a, b, c

>>> f = partial(test, b = 2, c = 3)    # 为后续参数提供命名默认值。
>>> f(1)
1 2 3

>>> f = partial(test, 1, c = 3)       # 为前面的位置参数和后面的命名参数提供默认值。
>>> f(2)
1 2 3
```

`partial` 会按下面的规则合并参数。

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)

    return newfunc
```

提示：

与函数有关的内容很多，但都涉及底层实现。要分清函数和对象方法的差别，后面会详细说明。

第 5 章 迭代器

在 Python 中，通常将实现接口称为遵守协议。因为 "弱类型" 和 "Duck Type" 的缘故，很多静态语言中繁复的模式被悄悄抹平。

5.1 迭代器

迭代器协议，仅需要 `__iter__()` 和 `next()` 两个方法。前者返回迭代器对象，后者依次返回序列数据，直到引发 `StopIteration` 异常表示结束。

最简单的做法是内置函数 `iter()`，它可以返回常用内置类型的迭代对象。问题是，序列类型已经可以被 `for` 处理，为何还要这么做？

```
>>> class MyData(object):
...     def __init__(self):
...         self._data = []
...
...     def add(self, x):
...         self._data.append(x)
...
...     def data(self):
...         return iter(self._data)

>>> d = MyData()

>>> d.add(1)
>>> d.add(2)
>>> d.add(3)

>>> for x in d.data(): print x
1
2
3
```

上面例子中，我们返回迭代器对象代替 `self._data` 列表，以免对象状态被外部意外修改。或许你会尝试返回 `tuple`，但莫忘了这需要复制整个列表，而且浪费更多的内存。

`iter()` 函数很方便，但缺点也很明确，就是无法让迭代中途终止。现在需要我们动手实现自己的迭代器对象。在设计原则上，通常会将迭代器从数据对象中分离出去。因为迭代器需要维持状态，且可能有多个迭代器在同时操控数据，这些不该成为数据对象的负担，无端提升了复杂度。

```
>>> class MyData(object):
...     def __init__(self, *args):
...         self._data = list(args)
...
```

```

...     def __iter__(self):
...         return MyIter(self)

>>> class MyIter(object):
...     def __init__(self, data):
...         self._data = data._data
...
...     def next(self):
...         if self._index >= len(self._data): raise StopIteration()
...         d = self._data[self._index]
...         self._index += 1
...         return d

>>> d = MyData(1, 2, 3)

>>> for x in d.data(): print x
1
2
3

```

`MyData` 仅仅是数据容器，只需 `__iter__` 返回迭代器对象，而由 `MyIter` 提供 `next` 方法。

除了 `for` 循环，迭代器也可以直接用 `next()` 操控。

```

>>> d = MyData(1, 2, 3)

>>> it = iter(d)
>>> it
<__main__.MyIter object at 0x10d4fe850>

>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
StopIteration

```

5.2 生成器

我们实现了自己的迭代器类型，但基于 `index` 的代码有些丑陋。为此，`Python` 提供了 `yield` 关键字，它返回实现了迭代器协议的 `Generator Object`。

```

>>> class MyData(object):
...     def __init__(self, *args):
...         self._data = list(args)

```

```

...
...     def __iter__(self):
...         for x in self._data:
...             yield x

>>> d = MyData(1, 2, 3)

>>> for x in d: print x
1
2
3

```

编译器魔法会将包含 `yield` 的方法 (或函数) 重新打包, 使其返回 "generator object"。这样, 我们就不用废力气维护额外的迭代器类型了。

```

>>> d.__iter__()
<generator object __iter__ at 0x10db01280>

>>> it = iter(d)
>>> next(it)
1

```

那么 `yield` 为何能实现这样的魔法呢? 当 `yield` 返回数据后, 会立即阻塞函数执行, 让出执行绪, 直到下次 `next()` 调用时再从阻塞点恢复。这看上去就是协程的工作方式, 不是吗?

除了用 `next()` 获取 `yield` 返回值外, 还可用 `send()` 在获取数据的同时发送 "通知" 给 `yield`。

```

>>> def test():
...     for x in range(3):
...         d = yield x          # x 是 yield 要返回的数据, d 是从 send 接收的数据。
...         if d: print "yield: ", d

>>> it = test()

>>> it.send(None)
0

>>> it.send("a")
yield: a
1

>>> it.send("b")
yield: b
2

>>> it.send("c")
yield: c
StopIteration

```

在发送通知前，必须确保迭代器已经启动，可以用 `next()` 或者 `send(None)`。

5.3 模式

善用迭代器，总会有意外的惊喜。

生产消费模型

利用 `yield` 类协程特性，我们无需多线程就可以编写生产消费模型。

```
>>> def consumer():
...     while True:
...         d = yield
...         if not d: break
...         print "consumer:", d

>>> c = consumer()          # 创建消费者
>>> c.send(None)            # 启动消费者

>>> c.send(1)                # 生产数据，并提交给消费者。
consumer: 1

>>> c.send(2)
consumer: 2

>>> c.send(3)
consumer: 3

>>> c.send(None)            # 生产结束，通知消费者结束。
StopIteration
```

改进回调

回调函数是实现异步操作的常用手法，只不过代码规模一大，看上去就没那么舒服了。而且好好的一个逻辑被切分到两个函数里，维护也是个问题。有了 `yield`，我们完全可以用 `blocking style` 编写异步调用。

下面是 `callback` 版本的示例，其中 `Framework` 调用 `logic`，在完成某些操作或者接收到信号后，用 `callback` 返回异步结果。

```
>>> def framework(logic, callback):
...     s = logic()
...     print "[FX] logic: ", s
...     print "[FX] do something..."
```

```
...     callback("async:" + s)
```

```
>>> def logic():  
...     s = "mylogic"  
...     return s
```

```
>>> def callback(s):  
...     print s
```

```
>>> framework(logic, callback)  
[FX] logic: mylogic  
[FX] do something...  
async:mylogic
```

看看用 `yield` 改进的 `blocking style` 版本。

```
>>> def framework(logic):  
...     try:  
...         it = logic()  
...         s = next(it)  
...         print "[FX] logic: ", s  
...         print "[FX] do something"  
...         it.send("async:" + s)  
...     except StopIteration:  
...         pass
```

```
>>> def logic():  
...     s = "mylogic"  
...     r = yield s  
...     print r
```

```
>>> framework(logic)  
[FX] logic: mylogic  
[FX] do something  
async:mylogic
```

尽管 `framework` 比前面要复杂一些，但数量较多的 `logic` 却干净简洁许多。而且 `blocking style` 样式的编码给逻辑维护带来的好处也足够吸引人了。

5.4 宝藏

标准库 `itertools` 模块是每个 `Pythoner` 都不应该忽视的宝藏。

`chain`

连接多个迭代器。


```
>>> it = chain(xrange(3), "abc")
>>> list(it)
[0, 1, 2, 'a', 'b', 'c']
```

combinations

返回指定长度的元素顺序组合序列。

```
>>> it = combinations("abcd", 2)
>>> list(it)
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd')]

>>> it = combinations(xrange(4), 2)
>>> list(it)
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

`combinations_with_replacement` 会额外返回同一元素的组合。

```
>>> it = combinations_with_replacement("abcd", 2)
>>> list(it)
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'b'), ('b', 'c'), ('b', 'd'),
('c', 'c'), ('c', 'd'), ('d', 'd')]
```

compress

按条件表过滤迭代器元素。

```
>>> it = compress("abcde", [1, 0, 1, 1, 0])
>>> list(it)
['a', 'c', 'd']
```

条件列表可以是任何布尔列表。

count

从起点开始，"无限"循环下去。

```
>>> for x in count(10, step = 2):
...     print x
...     if x > 17: break

10
12
14
16
```

cycle

当迭代器结束时，就从头来过。

```
>>> for i, x in enumerate(cycle("abc")):
...     print x
...     if i > 7: break

a
b
c
a
b
c
a
b
c
```

dropwhile

跳过头部符合条件的元素。

```
>>> it = dropwhile(lambda i: i < 4, [2, 1, 4, 1, 3])
>>> list(it)
[4, 1, 3]
```

takewhile 则仅保留头部符合条件的元素。

```
>>> it = takewhile(lambda i: i < 4, [2, 1, 4, 1, 3])
>>> list(it)
[2, 1]
```

groupby

将连续出现的相同元素进行分组。

```
>>> [list(k) for k, g in groupby('AAAABBBCCDAABCCDD')]
[['A'], ['B'], ['C'], ['D'], ['A'], ['B'], ['C'], ['D']]

>>> [list(g) for k, g in groupby('AAAABBBCCDAABCCDD')]
[['A', 'A', 'A', 'A'], ['B', 'B', 'B'], ['C', 'C'], ['D'], ['A', 'A'], ['B', 'B'], ['C', 'C'], ['D', 'D']]
```

ifilter

与内置函数 `filter()` 类似，仅保留符合条件的元素。

```
>>> it = ifilter(lambda x: x % 2, xrange(10))
>>> list(it)
[1, 3, 5, 7, 9]
```

`ifilterfalse` 正好相反，保留不符合条件的元素。

```
>>> it = ifilterfalse(lambda x: x % 2, xrange(10))
>>> list(it)
[0, 2, 4, 6, 8]
```

imap

与内置函数 `map()` 类似。

```
>>> it = imap(lambda x, y: x + y, (2,3,10), (5,2,3))
>>> list(it)
[7, 5, 13]
```

islice

以切片的方式从迭代器获取元素。

```
>>> it = islice(xrange(10), 3)
>>> list(it)
[0, 1, 2]

>>> it = islice(xrange(10), 3, 5)
>>> list(it)
[3, 4]

>>> it = islice(xrange(10), 3, 9, 2)
>>> list(it)
[3, 5, 7]
```

izip

与内置函数 `zip()` 类似，多余元素会被抛弃。

```
>>> it = izip("abc", [1, 2])
>>> list(it)
[('a', 1), ('b', 2)]
```

要保留多余元素可以用 `izip_longest`，它提供了一个补缺参数。

```
>>> it = izip_longest("abc", [1, 2], fillvalue = 0)
>>> list(it)
[('a', 1), ('b', 2), ('c', 0)]
```

permutations

与 `combinations` 顺序组合不同，`permutations` 让每个元素都从头组合一遍。

```
>>> it = permutations("abc", 2)
>>> list(it)
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]

>>> it = combinations("abc", 2)
>>> list(it)
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

product

让每个元素都和后面的迭代器完整组合一遍。(笛卡尔积元组)

```
>>> it = product("abc", [0, 1])
>>> list(it)
[('a', 0), ('a', 1), ('b', 0), ('b', 1), ('c', 0), ('c', 1)]
```

repeat

将一个对象重复 `n` 次。

```
>>> it = repeat("a", 3)
>>> list(it)
['a', 'a', 'a']
```

starmap

按顺序处理每组元素。

```
>>> it = starmap(lambda x, y: x + y, [(1, 2), (10, 20)])
>>> list(it)
[3, 30]
```

tee

复制迭代器。

```
>>> for it in tee(xrange(5), 3):  
...     print list(it)  
  
[0, 1, 2, 3, 4]  
[0, 1, 2, 3, 4]  
[0, 1, 2, 3, 4]
```

提示：

乱花迷人眼..... Python 有许多奇怪的类库，比如一个简单随机数生成都能有 n 种闻所未闻的方法。Pythoner 远比其他语言使用者幸福，庞大而专业的类库资源，让我们可以更懒一些。

第 6 章 模块

在 Python 中，模块 (module) 对应同名的 py 源码文件。将多个模块文件放到独立的子目录中，并提供初始化文件 `__init__.py`，就形成了包 (package)。

无论是导入包，还是导入包中的任何模块或成员，都会优先执行初始化文件，且仅执行一次。我们可以用其初始化包环境，存储帮助、版本等信息。

6.1 模块对象

模块对象有几个重要的特别属性：

- `__name__`: 模块名 `<package>.<module>`，在 `sys.modules` 中以此为 key。
- `__file__`: 模块完整文件名。
- `__dict__`: 模块 `globals` 名字空间。

进程入口模块名是 `"__main__"`。

除了使用 py 文件外，还可以动态创建空模块对象。

```
>>> import sys, types

>>> m = types.ModuleType("sample", "sample module.")      # 用 type 创建对象。
>>> m
<module 'sample' (built-in)>

>>> m.__dict__
{'__name__': 'sample', '__doc__': 'sample module.'}

>>> sys.modules.get("sample")                               # 并没有添加到 sys.modules。

>>> def test(): print "test..."
>>> m.test = test                                           # 动态添加模块成员。
>>> m.test()
test...
```

动态添加模块成员时，须注意函数所引用的是其定义模块的名字空间。

```
>>> def test(): print "test:", __name__
>>> test()
test: __main__

>>> m.test = test
>>> m.test()
test: __main__
```

还有 `imp.new_module()` 也可以动态创建模块对象，同样不会添加到 `sys.modules`。

```
>>> import imp

>>> m = imp.new_module("test")
>>> m
<module 'test' (built-in)>

>>> m.__dict__
{'__name__': 'test', '__doc__': None, '__package__': None}
```

reload

如果已导入模块的源文件发生变更，可使用内置函数 `reload()` 重新载入。重新载入的模块依旧使用原内存地址，只不过被其他模块引用的原内部成员不会被刷新。

测试一下，为避免本地名字引用造成干扰，我们直接从 `sys.modules` 获取模块。

```
>>> import sys

>>> hex(id(sys.modules["string"]))
'0x10b4fc6e0'

>>> reload(sys.modules["string"])
<module 'string'>

>>> hex(id(sys.modules["string"]))          # reload 后的模块地址未曾改变，所以其他地方对
'0x10b4fc6e0'                                # 该模块的引用就不会失效，而且被 "刷新"。
```

但如果采用手动方法重新载入，那么就会出现两个不同的模块对象了。

```
>>> del sys.modules["string"]
>>> sys.modules["string"] = __import__("string")

>>> hex(id(sys.modules["string"]))          # 地址变了。
'0x10bc17a98'
```

6.2 搜索路径

虚拟机按以下顺序搜索包和模块：

- 进程根目录。
- 用 `PYTHONPATH` 环境变量指定的路径。
- Python 标准库目录。
- 路径文件 (.pth) 保存的路径 (通常放在 `site-packages` 目录)。

进程启动后，所有这些路径都被组织到 `sys.path` 列表中 (顺序可能会被修改)。任何 `import` 操作都按照 `sys.path` 列表查找目标模块。当然，可以用代码往 `sys.path` 添加自定义路径。

虚拟机按以下顺序匹配目标模块：

- `py` 源码文件。
- `pyc` 字节码文件。
- `egg` 包文件或包目录。
- `so`、`dll`、`pyd` 等扩展文件。
- Python 内置模块。
- 其他各类模块。

要执行程序，源文件不是必须的。实际上，很多软件发布时都会删掉 `py` 文件，仅保留二进制 `pyc` 字节码文件。但要注意，字节码很容易被反编译，不能奢求它能带来安全。

find_module

可用 `imp.find_module()` 获取模块的具体文件信息。

```
>>> import imp

>>> imp.find_module("os")
(
    <open file '/System/.../2.7/lib/python2.7/os.py', mode 'U' at 0x1013aa420>,
    '/System/.../2.7/lib/python2.7/os.py',
    ('.py', 'U', 1)
)
```

6.3 导入模块

通常进程中的模块对象总是单例。在第一次导入后，模块会被添加到 `sys.modules`，以后总是检查模块对象是否已经存在。可以用 `sys.modules[__name__]` 获取当前模块对象。

关键字 `import` 将包、模块或成员对象导入到当前名字空间中，可以是 `globals`，也可以是函数内部的 `locals` 名字空间。

```
>>> import pymongo, redis
>>> import pymongo.connection, pymongo.database
>>> import pymongo.connection as mgoconn, pymongo.database as mgodb

>>> from pymongo import connection
>>> from pymongo import connection, database
>>> from pymongo import connection as mgoconn, database as mgodb
```



```
>>> from pymongo import *
>>> from pymongo.connection import *
```

如果待导入对象和当前名字空间中已有的名字冲突，可以用 `as` 换个别名。需要注意，`"import *"` 不会导入模块私有成员（以下划线开头的名字）和 `__all__` 未指定的名字。

因为 `import` 实际导入的是目标模块 `globals` 名字空间中的成员，那么就有一个问题：目标模块也会导入其他模块，这些模块同样在目标模块的名字空间中。`import * 目标模块` 时，所有这些一并被带入到当前模块中，造成一定程度的污染。因此建议在模块中用 `__all__` 指定可以被批量导出的成员。

```
__all__ = ["add", "x"]
```

私有成员和 `__all__` 不会影响显式导出目标模块成员。`Python` 并没有严格的私有权限控制，仅以特定的语法规则来提醒调用人员。

在函数中使用 `"import *"` 会引发警告信息，但并不影响使用。

```
def main():
    import test
    from test import add, _x
    from sys import *          # SyntaxWarning: import * only allowed at module level
```

如果将包中要公开的模块和成员导入初始化文件 `__init__.py` 名字空间中，那么只需导入包，就能访问所需的目标成员，根本无需导入具体模块。这么做有助于隐藏包的实现细节，降低使用复杂度，减少外部对包文件组织结构的依赖。

`__import__`

和 `import` 关键字不同，内置函数 `__import__()` 以字符串为参数载入模块。载入的模块会被添加到 `sys.modules` 列表，但不会在当前名字空间中创建引用。

```
>>> import sys

>>> sys.modules.get("zlib")      # 没有 zlib。

>>> __import__("zlib")          # 导入 zlib，返回模块对象。
<module 'zlib'>

>>> sys.modules.get("zlib")      # zlib 添加到 sys.modules。
<module 'zlib'>

>>> globals().get("zlib")        # 名字空间中没有 zlib，除非将 __import__ 结果关联到某个名字。
```

用 `__import__` 导入 `package.module` 时，返回的是 `package` 而非 `module`。看下面的例子：

```
test <dir>
|_ __init__.py
|_ add.py

>>> m = __import__("test.add")

>>> m                                     # 返回的并不是 test.add 模块。
<module 'test' from 'test/__init__.pyc'>

>>> m.__dict__.keys()                     # 还好 add 在 test 的名字空间中。
['__builtins__', '__file__', '__package__', '__path__', 'add', '__name__', '__doc__']

>>> m.add                                 # 得这样才能访问 add 模块。
<module 'test.add' from 'test/add.pyc'>
```

只有 `fromlist` 参数不为空时，才会返回目标模块。

```
>>> m = __import__("test.add", fromlist = ["*"])

>>> m
<module 'test.add' from 'test/add.pyc'>

>>> m.__dict__.keys()
['__builtins__', '__file__', '__package__', 'hi', 'x', '__name__', '__doc__']
```

`__import__` 使用有点麻烦，建议用 `importlib.import_module()` 代替。

```
>>> import sys, importlib

>>> m = importlib.import_module("test.add")
>>> m                                     # 返回的是目标模块，而非 package。
<module 'test.add' from 'test/add.pyc'>

>>> sys.modules.get("test.add")           # 模块自然要添加到 sys.modules。
<module 'test.add' from 'test/add.pyc'>

>>> "test.add" in globals().keys()        # 没有添加到当前名字空间中。
False

>>> importlib.import_module(".add", "test") # 使用 "." 或 ".." 指定模块在多层
<module 'test.add' from 'test/add.pyc'>    # package 中位置。(必须)
```

load_source

标准库 `imp` 模块提供了 `load_source()`、`load_compiled()` 等几个函数，可用来载入指定路径下的模块文件。该路径可以不在 `sys.path` 列表中，且优先使用已编译的字节码文件。

需要小心的是，这些函数操作类似 `reload()`，也就是说每次都会新建模块对象。

```
>>> imp.load_source("add", "./test/add.py")
<module 'add' from './test/add.pyc'>
```

6.4 构建包

包支持多级嵌套，包文件路径存放在 `__path__` 字段中。如果要获取包里面的所有模块列表，不应该用 `os.listdir()`，而是 `pkgutil` 模块。

```
test <dir>
|_ __init__.py
|_ add.py
|_ user.py
|
|_ a <dir>
.   |_ __init__.py
.   |_ sub.py
|
|_ b <dir>
    |_ __init__.py
    |_ sub.py
```

```
>>> import pkgutil, test

>>> for _, name, ispkg in pkgutil.iter_modules(test.__path__, test.__name__ + "."):
...     print "name: {0:12}, is_sub_package: {1}".format(name, ispkg)
...
name: test.a          , is_sub_package: True
name: test.add        , is_sub_package: False
name: test.b          , is_sub_package: True
name: test.user       , is_sub_package: False

>>> for _, name, ispkg in pkgutil.walk_packages(test.__path__, test.__name__ + "."):
...     print "name: {0:12}, is_sub_package: {1}".format(name, ispkg)
...
name: test.a          , is_sub_package: True
name: test.a.sub      , is_sub_package: False
name: test.add        , is_sub_package: False
name: test.b          , is_sub_package: True
name: test.b.sub      , is_sub_package: False
name: test.user       , is_sub_package: False
```

函数 `iter_modules()` 和 `walk_packages()` 的区别在于：后者会迭代所有深度的子包。

`pkgutil.get_data()` 可以读取包内任何资源文件内容。比如获取 `test/add.py` 源码字符串。

```
>>> pkgutil.get_data("test", "add.py")
'#coding=utf-8\n\nx = 1\n\ndef hi():\n    pass\n\n\nprint "add init"\n'
```

Python Egg

Python 支持将包压缩成一个独立文件，以便于发布。类似 Java jar 的做法。

1. 首先，安装 `setuptools`。

```
$ sudo easy_install setuptools
```

2. 创建一个空目录，将包目录完整拷贝到该目录下。

3. 创建 `setup.py` 文件。(<http://docs.python.org/2/distutils/setupscript.html>)

```
from setuptools import setup, find_packages

setup (
    name = "test",
    version = "0.0.9",
    keywords = ("test", ),
    description = "test package",

    url = "http://github.com/qyuhen",
    author = 'Q.yuhen',
    author_email = "qyuhen@hotmail.com",

    packages = find_packages(),
)
```

4. 创建 egg 压缩文件。

```
$ python setup.py bdist_egg

running bdist_egg
running egg_info
creating test.egg-info
... ..
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist/test-0.0.9-py2.7.egg' and adding 'build/.../egg' to it
removing 'build/bdist.macosx-10.8-intel/egg' (and everything under it)
```

看看现在的目录里有什么。

```
$ ls -lh
total 8
```

```

drwxr-xr-x  4 yuhen  staff  136B 12 30 00:40 build
drwxr-xr-x  3 yuhen  staff  102B 12 30 00:40 dist
-rw-r--r--  1 yuhen  staff  294B 12 30 00:39 setup.py
drwxr-xr-x 10 yuhen  staff  340B 12 30 00:22 test
drwxr-xr-x  6 yuhen  staff  204B 12 30 00:40 test.egg-info

$ ls -lh dist
total 8
-rw-r--r--  1 yuhen  staff   3.2K 12 30 00:40 test-0.0.9-py2.7.egg

$ tar tvf dist/test-0.0.9-py2.7.egg
-rwxrwxrwx  0 0      0          1 12 30 00:40 EGG-INFO/dependency_links.txt
-rwxrwxrwx  0 0      0         226 12 30 00:40 EGG-INFO/PKG-INFO
-rwxrwxrwx  0 0      0         228 12 30 00:40 EGG-INFO/SOURCES.txt
-rwxrwxrwx  0 0      0          5 12 30 00:40 EGG-INFO/top_level.txt
-rwxrwxrwx  0 0      0          1 12 30 00:40 EGG-INFO/zip-safe
-rwxrwxrwx  0 0      0         21 12 30 00:15 test/__init__.py
-rwxrwxrwx  0 0      0        137 12 30 00:40 test/__init__.pyc
-rwxrwxrwx  0 0      0         60 12 30 00:15 test/add.py
-rwxrwxrwx  0 0      0        305 12 30 00:40 test/add.pyc
-rwxrwxrwx  0 0      0          0 12 30 00:15 test/user.py
-rwxrwxrwx  0 0      0        133 12 30 00:40 test/user.pyc
-rwxrwxrwx  0 0      0          0 12 30 00:15 test/a/__init__.py
-rwxrwxrwx  0 0      0        139 12 30 00:40 test/a/__init__.pyc
-rwxrwxrwx  0 0      0          8 12 30 00:15 test/a/sub.py
-rwxrwxrwx  0 0      0        151 12 30 00:40 test/a/sub.pyc
-rwxrwxrwx  0 0      0          0 12 30 00:15 test/b/__init__.py
-rwxrwxrwx  0 0      0        139 12 30 00:40 test/b/__init__.pyc
-rwxrwxrwx  0 0      0          8 12 30 00:15 test/b/sub.py
-rwxrwxrwx  0 0      0        151 12 30 00:40 test/b/sub.pyc

```

只需将 `test-0.0.9-py2.7.egg` 文件路径添加到路径文件 (`.pth`) 或 `PYTHONPATH` 环境变量，就可以使用了。而最常见的做法是用 `easy_install` 将其安装到 `site_packages` 目录。

```

$ sudo easy_install dist/test-0.0.9-py2.7.egg

Processing test-0.0.9-py2.7.egg
Copying test-0.0.9-py2.7.egg to /Library/Python/2.7/site-packages
Adding test 0.0.9 to easy-install.pth file

Installed /Library/Python/2.7/site-packages/test-0.0.9-py2.7.egg
Processing dependencies for test==0.0.9
Finished processing dependencies for test==0.0.9

```

安装后的搜索路径被自动添加到 `site-packages/easy-install.pth` 文件。

第 7 章 类

诸如“万事万物皆对象”这类宗教宣传语，听得耳朵都出茧子了。由于历史原因，Python 2.x 同时存在两种类模型，算是个不大不小的坑。面向对象思想的演变也在影响着语言的进化，单根继承在 Python 中对应的是 New-Style Class，而非那个半跛的 Classic Class。

Python 3 终于甩掉包袱，默认全部使用 New-Style Class。所以呢，就算还在用 2.x 开发，也别再折腾 Classic Class，踏踏实实从 object 继承，或在源文件设置默认元类。

```
>>> class User: pass

>>> type(User)                                # 默认是 Classic Class。
<type 'classobj'>

>>> issubclass(User, object)                  # 显然不是从 object 继承。
False

>>> __metaclass__ = type                      # 指定默认元类。

>>> class Manager: pass                       # 还是没有显式从 object 继承。

>>> type(Manager)                             # 但已经是 New-Style Class。
<type 'type'>

>>> issubclass(Manager, object)               # 确定了！
True
```

本书不再浪费口水折腾 Classic Class，所有内容均使用 New-Style Class。

7.1 名字空间

类型是类型，实例是实例。如同 def 那样，关键字 class 的作用是创建一个类型对象。前面章节也曾提到过，类型对象很特殊，在整个进程中是单例的，是不被回收的。

```
typedef struct
{
    PyObject_HEAD
    PyObject *cl_bases;           /* A tuple of class objects */
    PyObject *cl_dict;           /* A dictionary */
    PyObject *cl_name;           /* A string */

    PyObject *cl_getattr;
    PyObject *cl_setattr;
    PyObject *cl_delattr;
} PyClassObject;
```

因为 New-Style Class，Class 和 Type 总算是一回事了。

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

>>> u = User()

>>> type(u)
<class '__main__.User'>

>>> u.__class__
<class '__main__.User'>
```

类型 (class) 存储了所有的静态成员和方法 (包括实例方法)，而实例 (instance) 仅存储实例字段，从基类 object 开始，所有继承层次上的实例字段。

```
typedef struct
{
    PyObject_HEAD
    PyClassObject *in_class;      /* The class object */
    PyObject      *in_dict;      /* A dictionary */
    PyObject      *in_weakreflist; /* List of weak references */
} PyInstanceObject;
```

类型和实例各自拥有独立的名字空间。

```
>>> User.__dict__
<dictproxy object at 0x106eaa718>

>>> u.__dict__
{}
```

访问对象成员时，就从这个两个名字空间中查找，而非以往熟悉的 globals、locals。

成员查找顺序：instance.__dict__ -> class.__dict__ -> baseclass.__dict__

当然，要分清对象成员和普通名字的差别，就算在方法中，普通名字依然遵循 LEGB 规则。

7.2 字段

字段 Field 和 Property 是不同的，尽管 Python 将对象成员统称为 Attribute。

- 实例字段存储在 instance.__dict__，代表单个对象实体的状态。
- 静态字段存储在 class.__dict__，为所有同类型实例共享。
- 访问实例成员必须使用 self 或对象实例前缀。
- 访问静态成员必须使用类型或 self 前缀。
- 所有以双下划线开头的静态和实例字段都视为私有，会自动重命名。

```

>>> class User(object):
...     table = "t_user"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age

>>> u1 = User("user1", 20)                                # 实例字段存储在 instance.__dict__。
>>> u1.__dict__
{'age': 20, 'name': 'user1'}

>>> u2 = User("user2", 30)                                # 每个实例的状态都是相互隔离的。
>>> u2.__dict__
{'age': 30, 'name': 'user2'}

>>> for k, v in User.__dict__.items():                    # 静态字段存储在 class.__dict__。
...     print "{0:12} = {1}".format(k, v)

__module__      = __main__
__dict__         = <attribute '__dict__' of 'User' objects>
__init__         = <function __init__ at 0x106eb4398>
table           = t_user

```

可以在任何时候创建实例字段，但这仅影响该实例名字空间，与其他同类型实例无关。

```

>>> u1.x = 100

>>> u1.__dict__
{'x': 100, 'age': 20, 'name': 'user1'}

>>> u2.__dict__
{'age': 30, 'name': 'user2'}

```

要访问静态字段，除了用类型前缀外，也可以用 `self`。按照对象成员的查找规则，只要没有同名的实例成员，那么就继续查找 `class.__dict__`，这自然就是静态成员了。

```

>>> User.table                                           # 使用 <class>.<name> 查找静态成员。
't_user'

>>> u1.table                                             # 使用 <instance>.<name> 查找静态成员。
't_user'

>>> u2.table                                             # 静态成员为所有实例对象共享。
't_user'

>>> u1.table = "xxx"                                    # 在 instance.__dict__ 创建一个同名成员。

>>> u1.table                                             # 这回按照查找顺序，命中的就是实例成员了。

```



```
'xxx'

>>> u2.table                                     # 当然，这不会影响其他实例对象。
't_user'
```

为了自由和魔法，我们总有直接面对名字空间的时候。只是 `__dict__` 看上去并不那么美观，是该内置函数 `hasattr`、`getattr`、`setattr`、`delattr` 几兄弟上场了。

```
>>> getattr(u1, "x")                             # 使用字符串参数，有更多的灵活性。
100

>>> getattr(u1, "y", None)                       # 如果字段不存在，还可以提供一个默认值。

>>> hasattr(u1, "y")                             # 判断某个字段是否存在。
False

>>> setattr(u1, "y", "naonao...")                # 添加字段，同 u1.y = "naonao..."。
>>> u1.y
'naonao...'

>>> setattr(u1, "y", "xiaorong...")              # 或者修改已有的字段。
>>> u1.y
'xiaorong...'

>>> delattr(u1, "y")                              # 删除字段，同 del u1.y

>>> delattr(u1, "y")                              # 如果字段不存在，抛出异常。
AttributeError: y
```

这几个函数对所有 `Attribute` 都有效，无论是实例还是静态成员。

```
>>> setattr(User, "id_key", "uid")

>>> User.id_key
'uid'
```

面向对象一个很重要的特征就是封装，它隐藏对象内部实现细节，仅暴露用户所需的接口。因此私有字段是极重要的，可以避免意外的非逻辑修改。

在 `Python` 里，私有字段以双下划线开头。无论是静态还是实例私有字段，都会被编译器重命名成特殊格式：`__<classname>__<fieldname>`。

```
>>> class User(object):
...     __table = "t_user"
...
...     def __init__(self, name, age):
...         self.__name = name
...         self.__age = age
```

```

...
...     def __str__(self):
...         return "{0}: {1}, {2}".format(
...             self.__table,                # 重命名是编译期的事，编码时无需关心。
...             self.__name,
...             self.__age)

>>> u = User("tom", 20)

>>> u.__dict__                                # 可以看到私有实例字段被重命名了。
{'_User__name': 'tom', '_User__age': 20}

>>> str(u)
't_user: tom, 20'

>>> User.__dict__.keys()                      # 私有静态字段也被重命名。
['_User__table', ...]

```

某些时候，我们既想拥有私有字段，又不想放弃外部访问权限。毕竟在许诸多的语言里，也有反射的生存空间。

- 用重命名后的格式访问。
- 只用一个下划线，仅提醒访问者，但不重命名。

不必太纠结 "权限" 这个词，从底层来说，本来也没有私有一说。所有要做的，都只是遵循某种既定的规则而已。

7.3 属性

属性 (property) 不是字段，它是由 `getter`、`setter`、`deleter` 几个方法构成的逻辑。可直接返回字段值，也可能是完全动态的计算结果。

属性可以通过 `Decorator`、`Descriptor` 实现，原理以后再说。总之很简单，也很好理解。

```

>>> class User(object):
...     @property
...     def name(self): return self.__name        # 注意几个方法是同名的。
...
...     @name.setter
...     def name(self, value): self.__name = value
...
...     @name.deleter
...     def name(self): del self.__name

>>> u = User()

>>> u.name = "Tom"                            # 从 instance.__dict__ 可以看出属性和字段的差异。

```

```

>>> u.__dict__
{'_User__name': 'Tom'}

>>> u.name                                # instance.__dict__ 中并没有 name, 显然是 getter 起作用了。
'Tom'

>>> del u.name                             # 好吧, 这是 deleter。
>>> u.__dict__
{}

>>> for k, v in User.__dict__.items():
    print "{0:12} = {1}".format(k, v)
...
__module__    = __main__
__doc__       = None
__dict__      = <attribute '__dict__' of 'User' objects>
__weakref__   = <attribute '__weakref__' of 'User' objects>
name          = <property object at 0x106ed6100>

```

从 `class.__dict__` 可以看出, 几个 Decorator 方法最终合并成了 Property Descriptor。这也解释了, 几个同名方法是如何共存的。既然如此, 我们可以直接用内置函数 `property()` 实现属性。

```

>>> class User(object):
...     def get_name(self): return self.__name
...     def set_name(self, value): self.__name = value
...     def del_name(self): del self.__name
...     name = property(get_name, set_name, del_name, "help...")

>>> for k, v in User.__dict__.items():
...     print "{0:12} = {1}".format(k, v)

__module__    = __main__
__doc__       = None
__dict__      = <attribute '__dict__' of 'User' objects>
__weakref__   = <attribute '__weakref__' of 'User' objects>
set_name      = <function set_name at 0x106eb4b18>
del_name      = <function del_name at 0x106eb4b90>
get_name      = <function get_name at 0x106eb4aa0>
name          = <property object at 0x106ec8db8>

>>> u = User()

>>> u.name = "Tom"
>>> u.__dict__
{'_User__name': 'Tom'}

>>> u.name
'Tom'

```

```
>>> del u.name
>>> u.__dict__
{}
```

区别不是太大，只是 `class.__dict__` 中依然保留几个函数，因为它们并不同名。

属性函数多半都很简单，可以考虑改用 `lambda` 实现。鉴于 `lambda` 函数不能使用赋值语句，故改用 `setattr`。还得注意别用会被重命名的私有字段名做参数。

```
>>> class User(object):
...     def __init__(self, uid):
...         self._uid = uid
...
...     uid = property(lambda self: self._uid)           # 只读属性。
...
...     name = property(lambda self: self._name, \       # 可读写属性。
...                      lambda self, value: setattr(self, "_name", value))

>>> u = User(1)

>>> u.uid
1
>>> u.uid = 100
AttributeError: can't set attribute

>>> u.name = "Tom"
>>> u.name
'Tom'
```

不同于前面提到的对象成员查找规则，属性总是比同名实例字段优先。

```
>>> u = User(1)

>>> u.name = "Tom"
>>> u.__dict__
{'_uid': 1, '_name': 'Tom'}

>>> u.__dict__["uid"] = 1000000                        # 显式在 instance.__dict__ 创建同名实例字段。
>>> u.__dict__["name"] = "xxxxxxx"

>>> u.__dict__
{'_uid': 1, 'uid': 1000000, 'name': 'xxxxxxx', '_name': 'Tom'}

>>> u.uid                                              # 访问的依旧是属性。
1

>>> u.name
```

```
'Tom'
```

既然使用了 OOP，那么就尽可能用属性，而不是直接暴露字段。

7.4 方法

实例方法和函数的最大区别是 `self` 这个隐式参数。

```
>>> class User(object):
...     def print_id(self):
...         print hex(id(self))

>>> u = User()

>>> u.print_id()
0x10cf58b50

>>> u.print_id
<bound method User.print_id of <__main__.User object at 0x10cf58b50>>

>>> User.print_id
<unbound method User.print_id>
```

从上面的代码可以看出实例方法的特殊性。当用实例调用时，它是个 `bound method`，动态绑定到对象实例。而当用类型调用时，是 unbound method，必须显式传递 `self` 参数。

```
>>> User.print_id(u)
0x10cf58b50
```

那么静态方法呢？为什么必须用 `staticmethod`、`classmethod Decorator`？

```
>>> class User(object):
...     def a(): pass
...
...     @staticmethod
...     def b(): pass
...
...     @classmethod
...     def c(cls): pass

>>> User.a
<unbound method User.a>

>>> User.b
<function b at 0x10c8ef320>

>>> User.c
```

```
<bound method type.c of <class '__main__.User'>>
```

没用 Decorator 的方法 a，被当做了实例方法，自然不能当静态方法使用。

```
>>> User.a()
TypeError: unbound method a() must be called with User instance as first argument (got nothing instead)
```

而用了装饰器的静态方法 b、c，和实例隔离开来。classmethod 绑定了类型对象作为隐式参数。

```
>>> User.b()

>>> User.c()
<class '__main__.User'>
```

除了上面说的这些特点外，方法的使用和普通函数类似，可以有默认参数、变参等等。还有，实例方法隐式参数 self 只是习惯性命名，可以用你喜欢的任何名字。

说到对象，总会有几个特殊的方法：构造、析构。

方法 __new__ 创建对象实例，通常由默认元类提供。__init__ 初始化对象状态，__del__ 在对象回收前被调用，但因为已知原因，很少用而已。

```
>>> class User(object):
...     def __new__(cls, *args, **kwargs):
...         print "__new__", cls, args, kwargs
...         return object.__new__(cls)
...
...     def __init__(self, name, age):
...         print "__init__", name, age
...
...     def __del__(self):
...         print "__del__"

>>> u = User("Tom", 23)
__new__ <class '__main__.User'> ('Tom', 23) {}
__init__ Tom 23

>>> del u
__del__
```

构造方法 __new__ 用来创建对象实例，理论上可以返回任意类型。可问题是，返回不同的类型将导致 __init__ 方法不会被调用。

```
>>> class User(object):
...     def __new__(cls, *args, **kwargs):
...         print "__new__"
...         return 123
```

```

...
...     def __init__(self):
...         print "__init__"

>>> u = User()
__new__

>>> type(u)
<type 'int'>

>>> u
123

```

在方法里调用其他对象成员时，必须使用 **self** 对象实例引用。否则会当做普通的局部名字，依照 LEGB 规则从 **locals**、**globals** 中查找。

```

>>> table = "TABLE"

>>> class User(object):
...     table = "t_user"
...
...     def __init__(self, name, age):
...         self.__name = name
...         self.__age = age
...
...     def tostr(self):
...         return "{0}, {1}".format(
...             self.__name, self.__age) # 使用 self 引用实例字段。
...
...     def test(self):
...         print self.tostr()           # 使用 self 调用其他实例方法。
...         print self.table             # 使用 self 引用静态字段。
...         print table                  # 按 LEGB 查找外部名字空间。

>>> User("Tom", 23).test()
Tom, 23
t_user
TABLE

```

因为所有方法都存储在 **class.__dict__**，不可能出现同名 **key**，所以不支持方法重载 (**overload**)。

7.5 继承

除了所有基类的实例字段都存储在 **instance.__dict__** 外，其他成员依然是各归各家。

```

>>> class User(object):
...     table = "t_user"

```

```

...
...     def __init__(self, name, age):
...         self._name = name
...         self._age = age
...
...     def test(self):
...         print self._name, self._age

>>> class Manager(User):
...     table = "t_manager"
...
...     def __init__(self, name, age, title):
...         User.__init__(self, name, age)          # 必须显式调用基类初始化方法。
...         self._title = title
...
...     def kill(self):
...         print "213..."

>>> m = Manager("Tom", 40, "CX0")

>>> m.__dict__                                     # 实例包含了所有基类的字段。
{'_age': 40, '_title': 'CX0', '_name': 'Tom'}

>>> for k, v in Manager.__dict__.items():          # 派生类名字空间里没有任何基类成员。
...     print "{0:5} = {1}".format(k, v)

table = t_manager
kill  = <function kill at 0x10c9032a8>

>>> for k, v in User.__dict__.items():
...     print "{0:5} = {1}".format(k, v)

table = t_user
test  = <function test at 0x10c903140>

```

如果派生类不提供初始化方法，则默认会查找并使用基类的方法。

基类引用存储在 `__base__`，直接派生类存储在 `__subclasses__`。

```

>>> Manager.__base__
<class '__main__.User'>

>>> User.__subclasses__()
[<class '__main__.Manager'>]

```

可以用 `issubclass()` 判断是否继承自某个类型，或用 `isinstance()` 判断实例对象的基类。

```

>>> issubclass(Manager, User)

```



```

True

>>> issubclass(Manager, object)      # 可以是任何层级的基类。
True

>>> isinstance(m, Manager)
True

>>> isinstance(m, object)
True

```

成员查找规则允许我们直接用 `self` 引用基类成员，包括实例方法、静态方法、静态字段。但这里有个坑：如果派生类有一个与基类实例方法同名的静态成员（方法或字段），那么首先命中的是静态成员，因为派生类的名字空间优先于基类。

```

>>> class User(object):
...     def abc(self):
...         print "User.abc"

>>> class Manager(User):
...     @staticmethod
...     def abc():
...         print "Manager.static.abc"
...
...     def test(self):
...         self.abc()           # 按照查找顺序，首先找到的是 static abc()。
...         User.abc(self)      # 只好显式调用基类方法。

>>> Manager().test()
Manager.static.abc
User.abc

```

只需在派生类创建一个同名实例方法，即可实现 "覆盖 (override)"，参数列表都可以不同。

```

>>> class User(object):
...     def test(self):
...         print "User.test"

>>> class Manager(User):
...     def test(self, s):      # 依然是因为派生类名字空间优先于基类。
...         print "Manager.test:", s
...         User.test(self)    # 显式调用基类方法。

>>> Manager().test("hi!")
Manager.test: hi!
User.test

```

多重继承

Python 诞生的时候，单继承还不是主流思想。至于多重继承好与不好，估计要打很久的口水仗，全凭自己选择。

```
>>> class A(object):
...     def __init__(self, a):
...         self._a = a

>>> class B(object):
...     def __init__(self, b):
...         self._b = b

>>> class C(A, B):                                # 多重继承。基类顺序影响成员搜索顺序。
...     def __init__(self, a, b):
...         A.__init__(self, a)                    # 依次调用所有基类初始化方法。
...         B.__init__(self, b)

>>> C.__bases__
(<class '__main__.A'>, <class '__main__.B'>)

>>> c = C(1, 2)

>>> c.__dict__                                     # 包含所有基类实例字段。
{'_b': 2, '_a': 1}

>>> issubclass(C, A), isinstance(c, A)
(True, True)

>>> issubclass(C, B), isinstance(c, B)
(True, True)
```

多重继承成员搜索顺序，也就是 mro (method resolution order) 要稍微复杂一点。归纳一下就是：从下到上 (深度优先，从派生类到基类)，从左到右 (基类声明顺序)。

```
>>> C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type 'object'>]

>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type 'object'>)
```

super

Python 提供了类似 base 关键字的内置函数 super (其实是个 class)，它依照 __mro__ 顺序搜索基类成员。如此可避免将基类型名字写得到处都是，更有利于代码维护。

```
>>> class A(object):
...     def a(self): print "a"

>>> class B(object):
...     def b(self): print "b"

>>> class C(A, B):
...     def test(self):
...         base = super(C, self)      # 可以写到 __init__。
...         base.a()                  # A.a(self)
...         base.b()                  # B.b(self)

>>> C().test()
a
b
```

`super` 的类型参数指定了在 `__mro__` 列表中的搜索起始位置，总是返回该参数后续类型的成员。

```
>>> class A(object):
...     def test(self): print "a"

>>> class B(A):
...     def test(self): print "b"

>>> class C(B):
...     def __init__(self):
...         super(C, self).test()      # 从 mro 中 C 的后续类型，也就是 B 开始查找。
...         super(B, self).test()      # 从 B 的后续类型 A 开始查找。

>>> C.__mro__
[<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <type 'object'>]

>>> C()
b
a
<__main__.C object at 0x101498f90>
```

不建议用 `self.__class__` 代替当前类型名，因为这可能会引发混乱。

```
>>> class A(object):
...     def test(self):
...         print "a"

>>> class B(A):
...     def test(self):
...         super(self.__class__, self).test()    # 以 c instance 调用，那么
...         print "b"                             # self.__class__ 就是 C 类型对象。
...                                                # super(C, self) 总是查找其基类 B。
```

```

# 于是死循环发生了。
>>> class C(B):
...     pass

>>> C().test()
RuntimeError: maximum recursion depth exceeded while calling a Python object

```

在多重继承初始化方法中使用 `super` 可能会引发一些奇怪的问题。

```

>>> class A(object):
...     def __init__(self):
...         print "A"
...         super(A, self).__init__()      # 找到的是 B.__init__

>>> class B(object):
...     def __init__(self):
...         print "B"
...         super(B, self).__init__()      # object.__init__

>>> class C(A, B):
...     def __init__(self):
...         A.__init__(self)
...         B.__init__(self)

>>> o = C()
A
B
B
# 对输出结果很意外?
# super 按照 mro 列表顺序查找后续类型。那么在 A.__init__ 中
# super(A, self) 实际返回的是 B, super(A, self).__init__
# 自然导致一次 print "B" 输出。

>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type 'object'>)

```

多重继承会导致很多问题复杂化，建议改用组合模式实现类似的功能。

抽象类

抽象类 (Abstract Class) 无法创建对象实例，且派生类必须 "完整" 实现所有抽象成员。

```

>>> from abc import ABCMeta, abstractmethod, abstractproperty

>>> class User(object):
...     __metaclass__ = ABCMeta      # 通过元类来控制抽象类行为。
...
...     def __init__(self, uid):
...         self._uid = uid
...
...     @abstractmethod
...     def print_id(self): pass      # 抽象方法

```

```

...
...     name = abstractproperty()          # 抽象属性

>>> class Manager(User):
...     def __init__(self, uid):
...         User.__init__(self, uid)
...
...     def print_id(self):
...         print self._uid, self._name
...
...     name = property(lambda s: s._name, lambda s, v: setattr(s, "_name", v))

>>> u = User(1)                                # 抽象类无法实例化。
TypeError: Can't instantiate abstract class User with abstract methods name, print_id

>>> m = Manager(1)
>>> m.name = "Tom"
>>> m.print_id()
1 Tom

```

如果派生类也是抽象类型，那么可以部分实现或完全不实现基类抽象成员。

```

>>> class Manager(User):
...     __metaclass__ = ABCMeta
...
...     def __init__(self, uid, name):
...         User.__init__(self, uid)
...         self.name = name
...
...     uid = property(lambda s: s._uid)
...     name = property(lambda s: s._name, lambda s, v: setattr(s, "_name", v))
...     title = abstractproperty()

>>> class CX0(Manager):
...     def __init__(self, uid, name):
...         Manager.__init__(self, uid, name)
...
...     def print_id(self):
...         print self.uid, self.name, self.title
...
...     title = property(lambda s: "CX0")

>>> c = CX0(1, "Tom")
>>> c.print_id()
1 Tom CX0

```

派生类 **Manager** 也是一个抽象类，它实现了部分基类的抽象成员，又创建了新的抽象成员。这种做法在 **OOP** 里很常见，只须保证整个继承体系走下来，所有层次的抽象成员都被实现即可。

7.6 开放类

Open Class 几乎是所有动态语言的标配，也是精华之所在。即便是运行期，我们也可以恣意雕琢对象，增加或去掉些什么。

增加成员时，要明确知道放在哪儿，比如将实例方法放到 `instance.__dict__` 是没有效果的。

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

>>> def print_id(self): print hex(id(self))

>>> u = User()

>>> u.print_id = print_id                                # 添加到 instance.__dict__
>>> u.__dict__
{'print_id': <function print_id at 0x10c88e320>}

>>> u.print_id()                                         # 失败，不是 bound method。
TypeError: print_id() takes exactly 1 argument (0 given)

>>> u.print_id(u)                                         # 这仅仅当做一个普通函数字段来用。
0x10c91c0d0
```

因为不是 `bound method`，所以必须显式传递对象引用。正确的做法是放到 `class.__dict__`。

```
>>> User.__dict__["print_id"] = print_id                # dictproxy 显然是只读的。
TypeError: 'dictproxy' object does not support item assignment

>>> User.print_id = print_id                             # 同 setattr(User, "print_id", print_id)

>>> User.__dict__["print_id"]
<function print_id at 0x10c88e320>

>>> u = User()

>>> u.print_id                                           # 总算是 bound method 了。
<bound method User.print_id of <__main__.User object at 0x10c91c090>>

>>> u.print_id()                                         # 测试正常。
0x10c91c090
```

静态方法必须用 `staticmethod`、`classmethod` 包装一下，否则会被当成实例方法。

```
>>> def mstatic(): print "static method"

>>> User.mstatic = staticmethod(mstatic)                # 使用装饰器包装。
```

```

>>> User.mstatic                                     # 正常的静态方法。
<function mstatic at 0x10c88e398>

>>> User.mstatic()                                   # 调用正常。
static method

>>> def cstatic(cls): print "class method:", cls

>>> User.cstatic = classmethod(cstatic)               # 注意 classmethod 和 staticmethod 的区别。

>>> User.cstatic                                     # classmethod 绑定到类型对象。
<bound method type.cstatic of <class '__main__.User'>>

>>> User.cstatic()                                   # 调用成功。
class method: <class '__main__.User'>

```

还有个很特别的 `__slots__`，这家伙会阻止实例对象创建 `__dict__`，仅为指定的成员分配空间。

```

>>> class User(object):
...     __slots__ = ("_name", "_age")
...
...     def __init__(self, name, age):
...         self._name = name
...         self._age = age

>>> u = User("Tom", 34)

>>> hasattr(u, "__dict__")
False

>>> u.title = "CX0"                                   # 动态增加字段失败。
AttributeError: 'User' object has no attribute 'title'

>>> del u._age                                         # 已有字段可被删除。
>>> u._age = 18                                        # 将坑补回是允许的。
>>> u._age
18

>>> del u._age                                         # 该谁的就是谁的，换个主是不行滴。
>>> u._title = "CX0"
AttributeError: 'User' object has no attribute '_title'

>>> vars(u)                                            # 因为没有 __dict__，vars 失败。
TypeError: vars() argument must have __dict__ attribute

```

没 `__dict__` 也没关系，`dir()` 和 `inspect.getmember()` 一样好用。

```
>>> import inspect

>>> u = User("Tom", 34)

>>> {k:getattr(u, k) for k in dir(u) if not k.startswith("__")}
{'_age': 34, '_name': 'Tom'}

>>> {k:v for k, v in inspect.getmembers(u) if not k.startswith("__")}
{'_age': 34, '_name': 'Tom'}
```

7.7 操作符重载

__setitem__

又称索引器，像序列或字典类型那样操作对象。

```
>>> class A(object):
...     def __init__(self, **kwargs):
...         self._data = kwargs
...
...     def __getitem__(self, key):
...         return self._data.get(key)
...
...     def __setitem__(self, key, value):
...         self._data[key] = value
...
...     def __delitem__(self, key):
...         self._data.pop(key, None)
...
...     def __contains__(self, key):
...         return key in self._data.keys()

>>> a = A(x = 1, y = 2)

>>> a["x"]
1

>>> a["z"] = 3

>>> "z" in a
True

>>> del a["y"]

>>> a._data
{'x': 1, 'z': 3}
```


`__call__`

像函数那样调用对象，也就是传说中的 callable。

```
>>> class A(object):
...     def __call__(self, *args, **kwargs):
...         print hex(id(self)), args, kwargs

>>> a = A()

>>> a(1, 2, s = "hi")                # 完全可以把对象实例伪装成函数接口。
0x10c8957d0 (1, 2) {'s': 'hi'}
```

`__dir__`

配合 `__slots__` 隐藏内部成员。

```
>>> class A(object):
...     __slots__ = ("x", "y")
...
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
...     def __dir__(self):              # 必须返回 list, 而不是 tuple。
...         return ["x"]

>>> a = A(1, 2)

>>> dir(a)                            # y 不见了。
['x']
```

`__getattr__`

先看看这几个家伙的触发时机。

- `__getattr__`: 访问不存在的成员。
- `__setattr__`: 对任何成员的赋值操作。
- `__delattr__`: 删除成员操作。
- `__getattribute__`: 访问任何存在或不存在的成员，包括 `__dict__` 等特殊成员。

不要在这几个方法里直接访问对象成员，也不要使用 `hasattr/getattr/setattr/delattr` 函数，因为它们会被上面这个家伙拦截，那样就形成无限循环了。正确的做法是直接操作 `__dict__`。

可怕的 `__getattribute__` 会拦截 `__dict__` 在内的所有成员，因此只能用基类的 `__getattribute__` 返回结果。

如果需要返回错误，可以引发 `AttributeError` 异常。

```
>>> class A(object):
...     def __init__(self, x):
...         self.x = x                                # 会被 __setattr__ 捕获。
...
...     def __getattr__(self, name):
...         print "get:", name
...         return self.__dict__.get(name)
...
...     def __setattr__(self, name, value):
...         print "set:", name, value
...         self.__dict__[name] = value
...
...     def __delattr__(self, name):
...         print "del:", name
...         self.__dict__.pop(name, None)
...
...     def __getattribute__(self, name):
...         print "attribute:", name
...         return object.__getattribute__(self, name)

>>> a = A(10)                                         # __init__ 里面的 self.x = x 被 __setattr__ 捕获。
set: x 10
attribute: __dict__

>>> a.x                                               # 访问已存在字段，仅被 __getattribute__ 捕获。
attribute: x
10

>>> a.y = 20                                          # 创建新的字段，被 __setattr__ 捕获。
set: y 20
attribute: __dict__

>>> a.z                                               # 访问不存在的字段，被 __getattr__ 捕获。
attribute: z
get: z
attribute: __dict__

>>> del a.y                                           # 删除字段被 __delattr__ 捕获。
del: y
attribute: __dict__
```

`__cmp__`

`__cmp__` 通过返回数字来判断大小，而 `__eq__` 仅用于判断相等。

```
>>> class A(object):
...     def __init__(self, x):
...         self.x = x
...
...     def __eq__(self, o):
...         if not o or not isinstance(o, A): return False
...         return o.x == self.x
...
...     def __cmp__(self, o):
...         if not o or not isinstance(o, A): raise Exception()
...         return cmp(self.x, o.x)

>>> A(1) == A(1)
True

>>> A(1) == A(2)
False

>>> A(1) < A(2)
True

>>> A(1) <= A(2)
True
```

`__repr__`

`__repr__`、`__str__` 分别对应 `repr()` 和 `str()`。

```
>>> class A(object):
...     def __repr__(self):
...         return "<A id={0:#x}>".format(id(self))
...
...     def __str__(self):
...         return "Hello, World!"

>>> a = A()

>>> repr(a)
'<A id=0x10c916410>'

>>> str(a)
'Hello, World!'
```

提示：

面向对象涉及到的理论有几箩筐，显然不是本书要倒腾的内容，应该找几本专业大部头来啃。

第 8 章 异常

异常不仅仅是错误，还有一种正常的跳转逻辑。

8.1 异常

除多了个 `else` 分支外，和其他语言并没多少差别。

```
>>> def test(n):
...     try:
...         if n % 2:
...             raise Exception("Error Message!")
...     except Exception as ex:
...         print "Exception:", ex.message
...     else:
...         print "Else..."
...     finally:
...         print "Finally..."

>>> test(1)                                     # 引发异常，else 分支未执行，finally 总是在最后执行。
Exception: Error Message!
Finally...

>>> test(2)                                     # 未引发异常，else 分支执行。
Else...
Finally...
```

关键字 `raise` 抛出异常对象实例，`else` 分支只在没有异常发生时执行。可无论如何，`finally` 总是要走一趟的。

可以有多个 `except` 分支捕获不同类型的异常。

```
>>> def test(n):
...     try:
...         if n == 0:
...             raise NameError()
...         elif n == 1:
...             raise KeyError()
...         elif n == 2:
...             raise IndexError()
...         else:
...             raise Exception()
...     except (IndexError, KeyError) as ex: # 可以同时捕获不同类型的异常
...         print type(ex)
...     except NameError:
...         # 捕获具体异常类型，但对异常对象没兴趣。
...         print "NameError"
```

```

...     except:                                # 捕获任意类型异常。
...         print "Exception!"

>>> test(0)
NameError

>>> test(1)
<type 'exceptions.KeyError'>

>>> test(2)
<type 'exceptions.IndexError'>

>>> test(3)
Exception!

```

获取异常对象实例，也能写成下面这样，只是觉得和 "raise Exception, 'error message'" 一样不舒服，缺乏统一风格。

```

>>> def test():
...     try:
...         raise KeyError()
...     except (IndexError, KeyError), ex:
...         print type(ex)

```

支持在 `except` 中重新抛出异常。

```

>>> def test():
...     try:
...         raise Exception("error!")
...     except:
...         print "catch exception!"
...         raise                                # 原样抛出异常，不会修改 traceback 信息。

>>> test()
catch exception!
Traceback (most recent call last):
  raise Exception("error!")
Exception: error!

```

如果需要，可用 `sys.exc_info()` 获取异常信息，用 `excepthook()` 输出到 `stderr`。

```

>>> def test():
...     try:
...         raise KeyError("key error!")
...     except:
...         exc_type, exc_value, traceback = sys.exc_info()
...         sys.excepthook(exc_type, exc_value, traceback)

```

```
>>> test()
Traceback (most recent call last):
  raise KeyError("key error!")
KeyError: 'key error!'
```

自定义异常类型通常继承自 **Exception**。建议用具体的异常类型表示不同的错误行为，而不是用 **message** 这样的状态值。

除了异常，还可以显示警告信息。**warnings** 模块另有函数用来控制警告的具体行为。

```
>>> import warnings

>>> def test():
...     warnings.warn("hi!")      # 默认仅显式警告信息，不会中断执行。
...     print "test..."

>>> test()
UserWarning: hi!
test...
```

8.2 断言

断言 (**assert**) 虽然简单，但远比用 "print 调试大法" 好得多。

```
>>> def test(n):
...     assert n > 0, "n 必须大于 0"      # 错误信息是可选的。
...     print n

>>> test(1)
1

>>> test(0)
Traceback (most recent call last):
  assert n > 0, "n 必须大于 0"
AssertionError: n 必须大于 0
```

很简单，当条件不符时，抛出 **AssertionError** 异常。**assert** 受只读参数 **__debug__** 控制，可以在启动时添加 **"-O"** 参数使其失效。

```
$ python -O main.py
```

8.3 上下文

Context Management Protocol 为代码块提供了包含进入和离开两个方法的上下文环境，可用来提供上下文对象，拦截异常，自动清理等操作。

- `__enter__`: 初始化环境，返回上下文对象。
- `__exit__`: 执行清理操作。返回 `True` 时，将阻止异常向外传递。

```
>>> class MyContext(object):
...     def __init__(self, *args):
...         self._data = data
...
...     def __enter__(self):
...         print "__enter__"
...         return self._data                # 不一定要返回上下文对象自身。
...
...     def __exit__(self, exc_type, exc_value, traceback):
...         if exc_type: print "Exception:", exc_value
...         print "__exit__"
...         return True                      # 阻止异常向外传递。

>>> with MyContext(1, 2, 3) as data:        # 将 __enter__ 返回的对象赋值给 data。
...     print data

__enter__
(1, 2, 3)
__exit__

>>> with MyContext(1, 2, 3):                # 发生异常，显示并拦截。
...     raise Exception("data error!")

__enter__
Exception: data error!
__exit__
```

可以在一个 `with` 语句中使用多个上下文对象，依次按照 **FILO** 顺序调用。

```
>>> class MyContext(object):
...     def __init__(self, name):
...         self._name = name
...
...     def __enter__(self):
...         print self._name, "__enter__"
...         return self
...
...     def __exit__(self, exc_type, exc_value, traceback):
...         print self._name, "__exit__"
...         return True

>>> with MyContext("a"), MyContext("b"):
...     print "exec code..."
```



```
a __enter__
b __enter__
exec code...
b __exit__
a __exit__
```

contextlib

标准库 `contextlib` 提供了一个 `contextmanager` Decorator，用来简化上下文类型编码。

```
>>> from contextlib import contextmanager

>>> @contextmanager
... def closing(o):
...     print "__enter__"
...     yield o
...     print "__exit__"
...     o.close()          # 正常情况下要检查很多条件，比如 None，是否有 close 方法等。

>>> with closing(open("README.md", "r")) as f:
...     print f.readline()

__enter__
#学习笔记
__exit__
```

原理很简单，`contextmanager` 替我们创建 `Context` 对象，并利用 `yield` 切换执行过程。

- 通过 `__enter__` 调用 `closing` 函数，将 `yield` 结果作为 `__enter__` 返回值。
- `yield` 让出了 `closing` 执行权限，转而执行 `Context with` 代码块。
- 执行完毕，`__exit__` 发送消息，通知 `yield` 恢复执行 `closing` 后续代码。

和第 5 章提到的用 `yield` 改进回调的做法差不多。话说回来，`contextmanager` 的确让我们少写了很多代码。但也有个麻烦，因为不是自己写 `__exit__`，所以得额外处理异常。

```
>>> @contextmanager
... def closing(o):
...     try:
...         yield o
...     except:
...         pass          # 忽略，或抛出。
...     finally:          # 确保 close 被执行。
...         o.close()
```

`contextlib` 已有现成的 `closing` 可用，不用费心完善上面的例子。倒是可以试着自己动手写一个 `synchronized context`。`__enter__` 请求锁定 (lock)，`__exit__` 解除锁定 (unlock)。

C# 有 `with(IDisposable)`，Go 有 `defer`，Python 有 `Context ...` 善加利用，充分发挥想象力。

提示：

如果你从没抛出过自定义异常，那么得好好想想了.....

第 9 章 装饰器

装饰器 (Decorator) 在 Python 编程中极为常见, 可以轻松实现 Metadata、Proxy、AOP 等模式。简单点说, 装饰器通过返回包装对象实现间接调用, 以此来插入额外逻辑。

语法看上去和 Java Annotation、C# Attribute 类似, 但因为动态执行的缘故, 更简单, 更强大。

```
>>> @check_args
... def test(*args):
...     print args
```

还原成容易理解的方式:

```
>>> test = check_args(test)
```

类似的做法, 我们在使用 `staticmethod`、`classmethod` 时就已见过。那么 `check_args` 具体内容是什么?

```
>>> def check_args(func):
...     def wrap(*args):
...         args = filter(bool, args)
...         func(*args)
...
...     return wrap                                # 返回 wrap 函数对象

>>> @check_args                                    # 解释器执行 test = check_args(test)
... def test(*args):
...     print args

>>> test                                           # 现在 test 名字与 wrap 关联。
<function wrap at 0x108affde8>

>>> test(1, 0, 2, "", [], 3)                     # 通过 wrap(test(args)) 完成调用。
(1, 2, 3)
```

整个过程非常简单:

- 解释器将目标函数对象 `test` 作为参数传递给装饰器 `check_args`。
- 装饰器返回包装函数 `wrap` 实现对 `test` 的间接调用。
- 原函数名字 `test` 被重新关联到 `wrap`, 所有对该名字的调用实际都是 `call wrap`。

你完全可以把装饰器当做语法糖, 也可以放弃 "@" 写法。只不过那样不便于代码维护, 毕竟 AOP 的理念是横向切面编程。

装饰器不一定非得是个函数返回包装对象, 也可以是个类, 通过 `__call__` 完成目标调用。

```

>>> class CheckArgs(object):
...     def __init__(self, func):
...         self._func = func
...
...     def __call__(self, *args):
...         args = filter(bool, args)
...         self._func(*args)

>>> @CheckArgs                                     # 生成 CheckArgs 实例。
... def test(*args):
...     print args

>>> test                                           # 名字指向该实例。
<__main__.CheckArgs object at 0x107a237d0>

>>> test(1, 0, 2, "", [], 3)                     # 每次都是通过该实例的 __call__ 调用。
(1, 2, 3)

```

用一个装饰器对象实例替代原函数，以后的每次调用的都是该实例的 `__call__` 方法。这种写法有点啰嗦，还得注意避免在装饰器对象上保留状态。

Class

为 `Class` 提供装饰器同样简单，无非是将类型对象做为参数而已。

```

>>> def singleton(cls):
...     def wrap(*args, **kwargs):
...         o = getattr(cls, "__instance__", None)
...         if not o:
...             o = cls(*args, **kwargs)
...             cls.__instance__ = o
...         return o
...
...     return wrap                                # 返回 wrap 函数，可以看做原 class 的工厂方法。

>>> @singleton
... class A(object):
...     def __init__(self, x):
...         self.x = x

>>> A
<function wrap at 0x108afff50>

>>> a, b = A(1), A(2)
>>> a is b
True

```

将 class A 替换成 func wrap 可能有些不好看，修改一下，返回 class wrap。

```
>>> def singleton(cls):
...     class wrap(cls):
...         def __new__(cls, *args, **kwargs):
...             o = getattr(cls, "__instance__", None)
...             if not o:
...                 o = object.__new__(cls)
...                 cls.__instance__ = o
...             return o
...     return wrap
...

>>> @singleton
... class A(object):
...     def test(self): print hex(id(self))

>>> a, b = A(), A()

>>> a is b
True

>>> a.test()
0x1091e9990
```

创建继承自原类型的 class wrap，然后在 __new__ 里面做手脚就行了。

大多数时候，我们仅用装饰器为原类型（包括函数）增加一些额外属性，那么可直接返回原类型，无需创建 wrap 什么的。

```
>>> def action(cls):
...     cls.mvc = staticmethod(lambda: "ACTION")
...     return cls

>>> @action
... class Login(object): pass

>>> Login.mvc()
'ACTION'
```

这种做法就是 metaprogramming 了。

参数

参数让装饰器拥有变化，也更加灵活。只是需要两步才能完成：先传递参数，后传递类型。

```

>>> def table(name):
...     def _table(cls):
...         cls.__table__ = name
...         return cls
...     return _table

>>> @table("t_user")
... class User(object): pass

>>> @table("t_blog")
... class Blog(object): pass

>>> User.__table__
't_user'

>>> Blog.__table__
't_blog'

```

只比无参数版本多了最外层传递参数的调用，其他完全相同。

```
User = table("t_user")(User)
```

嵌套

可以在同一目标上使用多个装饰器。

```

>>> def A(func):
...     print "A"
...     return func

>>> def B(func):
...     print "B"
...     return func

>>> @A
... @B
... def test(): print "test"

B
A

```

分解一下，无非是函数嵌套调用。

```
test = A(B(test))
```

functools.wraps

如果装饰器返回的是包装对象，那么有些东西必然是不同的。

```
>>> def check_args(func):
...     def wrap(*args):
...         return func(*filter(bool, args))
...
...     return wrap

>>> @check_args
def test(*args):
...     """test function"""
...     print args

>>> test.__name__                # 冒牌货!
'wrap'

>>> test.__doc__                 # 山寨货连个说明书都木有!
```

一旦 `test` 的调用者要检查某些特殊属性，那么这个 `wrap` 就会暴露了。幸好有 `functools.wraps`。

```
>>> def check_args(func):
...     @functools.wraps(func)
...     def wrap(*args):
...         return func(*filter(bool, args))
...
...     return wrap

>>> @check_args
def test(*args):
...     """test function"""
...     print args

>>> test
<function test at 0x108b026e0>

>>> test.__name__
'test'

>>> test.__doc__
'test function'

>>> test(1, 0, 2, "", 3)
(1, 2, 3)
```

`functools.wraps` 是装饰器的装饰器，它的作用是将原函数对象的指定属性复制给包装函数对象，默认有 `__module__`、`__name__`、`__doc__`，或者通过参数选择。

提示：

想想看装饰器都能干嘛？

- AOP: 身份验证、参数检查、异常日志等等。
- Proxy: 对目标函数注入权限管理等。
- Context: 提供函数级别的上下文环境，比如 `Synchronized(func)` 同步。
- Caching: 先检查缓存是否过期，然后再决定是否调用目标函数。
- Metaprogramming: 这个自不必多说了。

还有谁？

第 10 章 描述符

其实很少有人会去刻意关注描述符 (Descriptor)，尽管它时时刻刻以属性、方法的身份出来晃荡。

描述符协议：

```
__get__(self, instance, owner)
__set__(self, instance, value)
__delete__(self, instance)
```

描述符对象总是依赖某个类型 (owner class) 而存在，就像属性总是某个类型的一部分。

- 自定义描述符类型至少要实现协议中的一个方法。
- 同时提供 `__get__` 和 `__set__` 方法，称为 **data descriptor**。
- 仅有 `__get__` 方法的，称为 **non-data descriptor**。
- 必须是 **owner class** 成员，可用 `owner_class` 或 `owner_instance` 访问。
- 解释器自动调用对应方法。但 `__set__`、`__delete__` 仅在 `owner_instance` 访问时有效。

```
>>> class MyDescriptor(object):
...     def __get__(self, instance, owner):          # 本例中 owner 是 class Data。
...         print "get:", instance, owner
...         return hex(id(instance))
...
...     def __set__(self, instance, value):
...         print "set:", instance, value
...
...     def __delete__(self, instance):
...         print "del:", instance

>>> class Data(object):
...     x = MyDescriptor()

>>> d = Data()

>>> d.x                                              # __get__ 的返回值。
get: <__main__.Data object at 0x107a23790> <class '__main__.Data'>
'0x107a23790'

>>> d.x = 100                                       # d 被当做 instance 实参。
set: <__main__.Data object at 0x107a23790> 100

>>> del d.x                                         # d 被当做 instance 实参。
del: <__main__.Data object at 0x107a23790>

>>> Data.x                                          # 以 owner 类型访问时，__get__ 有效。
get: None <class '__main__.Data'>                 # instance = None
'0x106a96148'
```

```
>>> Data.x = 1                                     # __set__ 对 class 调用无效。
                                                    # 因此 Data.x 被重新赋值。
>>> type(Data.x)
<type 'int'>
```

如果没有定义 `__get__` 方法，那么直接返回描述符对象，不会有什么默认 `__get__` 实现。

non-data descriptor

`data descriptor` 总是比同名的实例字段优先级高，这点在前面讲属性时已测试过。而 `non-data descriptor` 则悲剧了点，会被同名实例字段抢先。

```
>>> class Descriptor(object):
...     def __get__(self, instance, owner):
...         print "__get__"

>>> class Data(object):
...     x = Descriptor()

>>> d = Data()

>>> d.x                                             # 描述符有效。
__get__

>>> d.__dict__                                     # instance.__dict__ 没有同名字段。
{}

>>> d.x = 123                                     # 没有 __set__，创建同名实例字段。

>>> d.__dict__
{'x': 123}

>>> d.x                                             # 依据成员查找规则，实例字段被优先命中。
123

>>> Data.x                                         # 描述符在 owner_class.__dict__。
__get__
```

property

属性总是 `data descriptor`，这和是否提供 `setter` 无关。其优先级总是高过同名实例字段，如果没有提供 `setter`，`__set__` 方法会阻止赋值操作。

```
>>> class Data(object):
...     oid = property(lambda s: hex(id(s)))
```

```
>>> hasattr(Data.oid, "__set__")
True

>>> d = Data()

>>> d.oid
'0x107a23a90'

>>> d.oid = 123
AttributeError: can't set attribute
```

bound method

通过描述符，我们可以了解实例方法 `self` 参数是如何隐式传递的。

```
>>> class Data(object):
...     def test(self): print "test"

>>> d = Data()

>>> d.test                                     # 只有 bound method 才会隐式传递 self。
<bound method Data.test of <__main__.Data object at 0x10740b050>>

>>> Data.test.__get__(d, Data)                 # 向 __get__ 传递 instance 参数。
<bound method Data.test of <__main__.Data object at 0x10740b050>>

>>> Data.test                                 # unbound method 需显式传递 self。
<unbound method Data.test>

>>> Data.test.__get__(None, Data)               # instance 为 None。
<unbound method Data.test>
```

现在可以看出，`bound/unbound` 是 `__get__` 造成的，关键就是 `instance` 参数。那么 `self` 参数存在哪？由谁替我们自动传递 `self` 参数呢？

```
>>> bm = Data.test.__get__(d, Data)

>>> bm.__func__                               # 实际的目标函数 test。
<function test at 0x107404488>

>>> bm.__self__                               # __get__ instance 参数，也就是 self。
<__main__.Data object at 0x10740b050>

>>> bm.__call__()                             # __call__ 内部替我们传递 self !
test
```

```

>>> unbm = Data.test.__get__(None, Data)      # unbound method

>>> unbm.__func__
<function test at 0x107404488>

>>> unbm.__self__ is None                     # instance == None, self == None.
True

>>> unbm.__call__()                           # __call__ 会检查 __self__。
TypeError: unbound method test() must be called with Data instance as first argument
(got nothing instead)

>>> unbm.__call__(d)                          # 只好给 __call__ 有效的 instance。
test

```

classmethod

不同于 `staticmethod`，`classmethod` 会 `bound` 类型对象。

```

>>> class Data(object):
...     @classmethod
...     def test(cls): print cls

>>> Data.test.__get__(None, Data)
<bound method type.test of <class '__main__.Data'>>

>>> m = Data.test.__get__(None, Data)

>>> m.__self__                                # 参数名是什么并不重要。
<class '__main__.Data'>

>>> m.__call__()
<class '__main__.Data'>

```

提示：

描述符不同于装饰器，从某些角度来说，它提升了复杂度。了解其内部运行机制是值得鼓励的，但未必就非得把它当做一种可以炫耀的技巧写到代码里。**Python** 追求的是简单、简洁，最了不起的是你的同事能轻松阅读你的代码，但无须知道这些细节。

可用可不用，那便不用，除非能大幅度减少代码数量。

第 11 章 元类

类型对象地位超然，负责创建对象实例，控制对象行为 (方法)，星际母虫版的存在。那么作为女王陛下的类型对象又是谁生呢？——元类 (metaclass)，也就是类型的类型，真正的终极 BOSS。(虚拟机、解释器神马的属于上帝模式，就不扯了)

New-Style Class 的默认元类是 `type`。

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

>>> Data.__class__
<type 'type'>

>>> type.__class__          # 最终的类型就是 type，包括 type 自己。
<type 'type'>
```

关键字 `class` 会被编译成元类创建类型对象指令。

```
>>> Data = type("Data", (object,), {"x": 1})          # class 的实际行为。

>>> Data.__class__
<type 'type'>

>>> Data.__base__
<type 'object'>

>>> Data.x
1
```

正因为 `class` 和 `def` 一样是指令，我们可以在任何地方创建类型对象。

```
>>> def test():
...     class Data(object): pass
...     return Data

>>> Data = test()

>>> Data.__name__
'Data'

>>> type(Data)
<type 'type'>

>>> Data()
<__main__.Data object at 0x10659f4d0>
```

好了，现在应该理清这几人的关系，以及创建顺序了。

```

class = metaclass(...)          # 元类创建类型
instance = class(...)           # 类型创建实例

instance.__class__ is class      # 实例的类型
class.__class__ is metaclass    # 类型的类型

```

__metaclass__

除了使用默认元类 `type` 以外，还可以用 `__metaclass__` 属性指定自定义元类，以便对类型对象创建进行干预。

```

>>> class InjectMeta(type):
...     def __new__(cls, name, bases, attrs):
...         t = type.__new__(cls, name, bases, attrs)
...
...         def print_id(self): print hex(id(self))
...         t.print_id = print_id          # 为类型对象添加实例方法。
...         t.s = "Hello, World!"        # 添加静态字段。
...
...     return t

>>> class Data(object):
...     __metaclass__ = InjectMeta        # 显式指定元类。

>>> Data.__metaclass__
<class '__main__.InjectMeta'>

>>> Data.__class__
<class '__main__.InjectMeta'>

>>> dir(Data)
['__class__', ... 'print_id', 's']

>>> Data.s
'Hello, World!'

>>> Data().print_id()
0x10659d850

```

自定义元类通常都从 `type` 继承，习惯以 `Meta` 结尾，就像前面章节用过的抽象元类 `abc.ABCMeta` 那样。代码很简单，只需注意 `__new__` 和 `__init__` 方法参数的区别就行了。

```

>>> class InjectMeta(type):
...     def __new__(cls, name, bases, attrs):
...         print "class:", cls          # InjectMeta
...         print "name:", name

```

```

...     print "bases:", bases
...     print "attrs:", attrs
...     return type.__new__(cls, name, bases, attrs)
...
...     def __init__(cls, name, bases, attrs):
...         print "class:", cls                                # Data
...         type.__init__(cls, name, bases, attrs)

>>> class Data(object):
...     __metaclass__ = InjectMeta
...     x = 1
...     def test(self): pass

class: <class '__main__.InjectMeta'>
name: Data
bases: (<type 'object'>,)
attrs: {
    'test': <function test at 0x1065370c8>,
    'x': 1,
    '__module__': '__main__',
    '__metaclass__': <class '__main__.InjectMeta'>
}

class: <class '__main__.Data'>

```

当解释器创建类型对象时，会按以下顺序查找 `__metaclass__` 属性，以便获得有效的创建方法。

```
class.__metaclass__ -> bases.__metaclass__ -> module.__metaclass__ -> type
```

这也是为什么在模块中可以用 `__metaclass__` 为所有类型指定默认元类的缘故。

虽然惯例将元类写成 `type` 的派生类，但也可以用函数代替。

```

>>> def inject_meta(name, bases, attrs):
...     t = type(name, bases, attrs)
...     t.s = "Hello, World!"
...     return t

>>> class Data(object):
...     __metaclass__ = inject_meta

>>> Data.__metaclass__
<unbound method Data.inject_meta>

>>> Data.s
'Hello, World!'

```

magic

对象行为由类型决定，实例不过存储了状态数据。那么，当我们控制了类型对象的创建，也就意味着可以让对象的实际行为和代码存在极大的差异。这是魔法的力量，也是 **Python** 核心开发人员 **Tim Peters** 说出下面这番话的原因 (相比你对他的 **import this** 很熟悉)。

Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why). Tim Peters (c.l.p post 2002-12-22)

试着写两个简单的例子练练手。

静态类 (**static class**): 不允许创建实例，通常作为 **utility** 存在。

```
>>> class StaticClassMeta(type):
...     def __new__(cls, name, bases, attr):
...         t = type.__new__(cls, name, bases, attr)
...
...         def ctor(cls, *args, **kwargs):
...             raise RuntimeError("Cannot create a instance of the static class!")
...         t.__new__ = staticmethod(ctor)
...
...     return t

>>> class Data(object):
...     __metaclass__ = StaticClassMeta

>>> Data()
RuntimeError: Cannot create a instance of the static class!
```

密封类 (**sealed class**): 禁止被继承。

```
>>> class SealedClassMeta(type):
...     _types = set()
...
...     def __init__(cls, name, bases, attrs):
...         if cls._types & set(bases):
...             # 判断当前类型基类是否是 sealed class。
...             raise SyntaxError("Cannot inherit from a sealed class!")
...         cls._types.add(cls)
...         # 将当前类型加入到禁止继承集合。

>>> class A(object):
...     __metaclass__ = SealedClassMeta

>>> class B(A): pass
SyntaxError: Cannot inherit from a sealed class!
```


小结：

第一部分写了二十多天，掰手指头一数，删的比留的还多。写着写着就歪楼了，要么内容过深，失了笔记的本意。要么发散得厉害，收不住手。着实有些苦恼，远不如写博客或者挖坟来得痛快。

后面的标准库、扩展库就不日更了，改不定期补上，但可以保证不是月刊。(^_^)

如果你觉得这笔记还行，给俺发封 email 吧，算是鼓励。当然，发现错误，还是要扔砖头的，我一并接着，并积极改正错误，争取得到宽大处理。谢谢！（我替你说：卖萌可耻）

email: qyuheng@hotmail.com

weibo: <http://weibo.com/qyuheng>

QQ: 1620443

雨痕 2013-01-05 19:34

第二部分 标准库

第三部分 扩展库

附录

CPython

参数：

- -b: 不生成 pyc/pyo 字节码文件。
- -E: 忽略 PYTHONPATH 环境变量。
- -i: 执行完成后，进入交互模式。(通常用 pdb.pm() 进入异常现场)
- -O: 优化字节码，并设置 `__debug__ = False`。
- -OO: 优化字节码，并移除 `__doc__` 信息。
- -S: 不执行 `site.py`，不添加所有第三方库搜索路径。
- -v: 显示模块初始化和回收信息。
- -c: 直接执行 Python 代码。如: `python -c "print 1+2"`。
- -m: 执行模块。如: `python -m pdb main.py`

IPython

远超 Python shell 的增强版本，可以当做 "IDE" 使用。

命令	说明
<code>%quickref</code>	快速导引。
<code>%magic</code>	Magic Functions 详细说明。 <code>%fun?</code> 获取具体帮助。
<code>%lsmagic</code>	列出所有可用 Magic Functions。
<code>obj?, obj??</code>	获取对象信息，?? 返回更详细的信息，比如源码。
<code>?obj.*abc*</code>	返回对象匹配的成员。比如: <code>str.is*</code>
<code>!, !!</code>	执行系统命令，捕获输出结果为字符串或列表。
<code>%bookmark</code>	目录书签。
<code>%cd, %pwd, _dh</code>	工作目录。
<code>%dirs, %popd, %pushd</code>	目录栈。
<code>%doctest_mode</code>	切换 shell 提示样式，包括提示符、输出等设置。
<code>%ed, %edit</code>	使用编辑器打开文件。
<code>%env</code>	输出环境变量。
<code>%hist</code>	历史命令列表。
<code>%pdb</code>	pdb 开关。 <code>%debug</code> 进入最后一次异常场景， <code>pdb.pm()</code> 。
<code>%pdoc</code>	查看对象 <code>__doc__</code> 信息。
<code>%pfile, %pycat</code>	查看包含指定对象的文件内容。
<code>%pinfo, %pinfo2</code>	查看对象的详细信息。
<code>%pprint</code>	Pretty-Print 开关。
<code>%prun, %time, %timeit</code>	性能测试。
<code>%psearch</code>	在当前名字空间按通配符搜索名字。
<code>%psource</code>	显示对象源码。
<code>%reset</code>	重置环境，移除所有名字。
<code>%run</code>	执行指定文件。
<code>%who, %whos</code>	查看所有变量。

备注

- 在 shell command 中可以用 `$name` 引用 Python 名字, `$$name` 引用环境变量。
- 系统命名捕获可以直接赋值给某个名字, 如 `name = !uname`。
- `%ed`: `-n` 跳转到指定行; `-x` 退出编辑器时不执行。
- `%run`: `-n` 设定 `__name__` 为非 `"__main__"`; `-i` 引入交互环境名字空间; `-d` 进入调试模式; `-t` `timeit`; `-p` `profile`。

演示

```
In [1]: prun sum(range(1000))
         4 function calls in 0.000 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000    0.000    0.000    0.000 {range}
      1   0.000    0.000    0.000    0.000 {sum}
      1   0.000    0.000    0.000    0.000 <string>:1(<module>)
```

```
In [2]: time sum(range(1000))
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00 s
Out[2]: 499500
```

```
In [3]: timeit -n 10 -r 3 sum(range(1000))
10 loops, best of 3: 25.8 us per loop
```

PDB

习惯用 `ipdb`，代码高亮，更好的异常调试支持。

命令	说明
a, args	显示函数参数。
b, break	设置或显示断点列表。
bt, w, where	显示 stack trace。
c, cont, continue	继续执行，直到下一个有效断点。
cl, clear	清除断点。
d, down, u, up	在 call stack frame 列表间移动。
disable, enable	禁用或启用断点。
exit, q, quit	退出 pdb。
h, help	查看命令列表或详细使用方法。
ignore	暂时忽略某个断点，可设置忽略次数。
l, list	显示源码。
n, next	继续执行下一行代码。
p, pp	显示变量或表达式的结果。pp = pretty-print。
r, return	继续执行，直到当前函数结束。
run, restart	运行，重新运行。
s, step	继续执行下一行代码，继续函数内部。
tbreak	创建临时断点，命中后失效。
whatis	查看对象类型。

启动方式

- 在源码中插入 `import ipdb; ipdb.set_trace()`。
- 命令行 `python -m ipdb main.py`。
- IPython 执行 `%run -d main.py, %debug` 等命令。