Python 学习笔记

第 2 版

好好学习，天天向上

前言

最早接触 Python 是 2006 年底，算到现在也有 6 年的时间了。期间陆陆续续累积了不少资料和笔记，原本发布在个人博客里，后因种种原因博客被终止，这持续多年的习惯也暂停了很久。

可能是忘性越来越大的缘故，急切想着重新整理编写新版的笔记。一则，诸多的错误需要修正；其次，就是系统地更新到最新版本上来。现在的工作有些杂乱，要处理的事情很多，如果不做笔记，往往没多久就把辛苦得来的经验给忘得一干二净，所以下狠心在元旦假期加班完成初稿。

只所以取名为学习笔记，主要是不适合做入门课程使用，所有内容都假定读者有 "足够" 的编程基础。我原本也没打算把它写成正式读物，最主要的用途还是个人备忘罢了。

此书可以自由散播，但不能用于任何商业用途，也不能在未经许可的情况下修改其中的任何信息。
如您发现缺失错漏，请及时与我联系。谢谢！

- 书中内容以 Python 2.7 为主。
- 为阅读方便，代码输出结果被手工整理过。
- 因运行期环境不同，输出结果，尤其是内存地址会存在差异。
- 如不做特别说明，书中所指均为 CPython (www.python.org)。

不定期更新，可以到 github.com/qyuhen 获取最新版本。

代码测试环境：

- CPython 2.7.2, IPython 0.13.1
- MacBook Pro, 8GB, OS X 10.8

联系方式：

email: qyuhen@hotmail.com
QQ: 1620443

雨痕 二〇一二年冬于北京家中

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 增加附录内容。
2013-01-07 增加标准库。
2013-01-09 完成第一部分首次校对。

目录

第一部分 Python 语言

Python 2.7 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" 时将源码编译成字节码 (Byte Code)。而后，这些字节码会被虚拟机在一个 "巨大" 的核心函数里解释执行。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，这是整个进程的根对象。
- 初始化内置类型。数字、列表等类型都有专门的缓存策略需要处理。
- 创建 __builtin__ 模块，该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。
- 通过 site.py 将 site-packages 中的第三方扩展库添加到搜索路径列表。
- 执行入口 py 文件。执行前会将 __main__.__dict__ 作为名字空间传递进去。
- 程序执行结束。
- 执行清理操作，包括调用退出函数，GC 清理现场，释放所有模块等。
- 终止进程。

Python 源码是个宝库，其中有大量的编程范式和技巧可供借鉴，尤其是对内存的管理分配。个人建议有 C 基础的兄弟，在闲暇时翻看一二。

1.2 类型和对象

先有类型 (Type)，而后才能生成实例 (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减小，等于 0 时会被虚拟机回收 (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 位版本中，头长度为 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                 # 引用计数增加。
>>> sys.getrefcount(x)
3

>>> del y                 # 引用计数减小。
>>> sys.getrefcount(x)
2
```

类型指针则指向具体的类型对象，其中包含了继承关系、静态成员等信息。所有的内置类型对象都能从 types 模块中找到，至于 int、long、str 这些关键字可以看做是简短别名。

```
>>> 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 这类变长对象。其头部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。

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

我们习惯于将 x 称为变量，但在这里，更准确的词语是 "名字"。

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，它和所指向的目标对象一起在名字空间中构成一项 {name: object} 关联。

Python 有多种名字空间，比如称为 globals 的模块名字空间，称为 locals 的函数堆栈帧名字空间，还有 class、instance 名字空间。不同的名字空间决定了对象的作用域和生存周期。

```
>>> x = 123

>>> globals()                    # 获取 module 名字空间。
{'x': 123, ......}
```

可以看出，名字空间就是一个字典 (dict)。我们完全可以直接在名字空间添加项来创建名字。

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

>>> y
'Hello, World!'
```

在 Python 源码中，有这样一句话：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某个对象进行关联。其本身不包含目标对象的任何信息，只有通过对象头部的类型指针才能获知其具体类型，进而查找其相关成员数据。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对象。

```
>>> y
'Hello, World!'

>>> type(y)
<type 'str'>

>>> y = __import__("string")     # 将原本与字符串关联的名字指向模块对象。

>>> type(y)
<type 'module'>

>>> y.digits                     # 查看模块对象的成员。
'0123456789'
```

在函数外部，locals() 和 globals() 作用完全相同。而当在函数内部调用时，locals() 则是获取当前函数堆栈帧的名字空间，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)>,
}

>>> 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(0) 获取当前堆栈帧。
...     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 我们也可以函数定义模块的名字空间。

>>> 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

a = 1
def test():
    print {k:v for k, v in globals().items() if k != "__builtins__"}

>>> import test

>>> test.test()
{
       '__file__': 'test.pyc',
       '__name__': 'test',
       'a': 1,
       'test': <function test at 0x10bd85e60>,
}
```

可通过 <module>.__dict__ 访问其他模块的名字空间。

```
>>> test.__dict__                                    # test 模块的名字空间
{
       '__file__': 'test.pyc',
       '__name__': 'test',
       'a': 1,
       'test': <function test at 0x10bd85e60>,
}

>>> import sys

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

与名字空间有关的内容很多，比如作用域、LEGB 查找规则、成员查找规则等等。所有这些，都将在相关章节中给出详细说明。

使用名字空间管理上下文对象，带来无与伦比的灵活性，但也牺牲了执行性能。毕竟从字典中查找对象远比指针低效很多，各有得失。

1.4 内存管理

为提升执行性能，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，虚拟机每次从操作系统申请一块 256KB，取名为 arena 的大块内存。并按系统页大小，划分成多个 pool。每个 pool 继续分割成 n 个大小相同的 block，这是内存池最小存储单位。

block 大小是 8 的倍数，也就是说存储 13 字节大小的对象，需要找 block 大小为 16 的 pool 获取空闲块。所有这些都用头信息和链表管理起来，以便快速查找空闲区域进行分配。

大于 256 字节的对象，直接用 malloc 在堆上分配内存。程序运行中的绝大多数对象都小于这个阀值，因此内存池策略可有效提升性能。

当所有 arena 的总容量超出限制 (64MB) 时，就不再请求新的 arena 内存。而是如同 "大对象" 一样，直接在堆上为对象分配内存。另外，完全空闲的 arena 会被释放，其内存交还给操作系统。

引用传递

对象总是按引用传递，简单点说就是通过复制指针来实现多个名字指向同一对象。因为 arena 也是在堆上分配的，所以无论何种类型何种大小的对象，都存储在堆上。Python 没有值类型和引用类型一说，就算是最简单的整数也是拥有标准头的完整对象。

```
>>> 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，就需使用不可变类型，或对象复制品。

```
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 模块进行深度复制。
- 序列化对象，如 pickle、cPickle、marshal。

下面的测试建议不要用数字等不可变对象，因为其内部的缓存和复用机制可能会造成干扰。

```
>>> 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                      # 可以看到复制列表的元素依然是原对象。
False
>>> l2[0] is x
True

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

循环引用会影响 deepcopy 函数的运作，建议查阅官方标准库文档。

引用计数

Python 默认采用引用计数来管理对象的内存回收。当引用计数为 0 时，将立即回收该对象内存，要么将对应的 block 块标记为空闲，要么返还给操作系统。

为观察回收行为，我们用 __del__ 监控对象释放。

```
>>> 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，因为缓存的缘故，计数永远不会为 0，直到进程结束才由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，不妨碍对象回收的情况下间接引用对象。但不是所有类型都支持弱引用，比如 list、dict ，弱引用会引发异常。

改用弱引用回调监控对象回收。

```
>>> import sys, weakref

>>> class User(object): pass

>>> 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!"

>>> a = User()

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

>>> 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                                       #    计数 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。也可以此判断原对象死亡。
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，除了循环引用造成计数故障。简单明显的循环引用，可以用弱引用打破循环关系。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，此时只有靠 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，还有个专门处理循环引用的 GC。通常我们提到垃圾回收时，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，会额外添加用于追踪的 PyGC_Head。这些对象被添加到特殊链表里，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，这并不表示此类对象非得 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处理掉。也就是说，只要不存在循环引用，理论上可以禁用 GC。当执行某些密集运算时，临时关掉 GC 有助于提升性能。

```
>>> 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 管理新近加入的年青对象，GEN1 则是在上次回收中依然存活的对象，剩下 GEN2 存储的都是生命周期极长的家伙。每级代龄都有一个最大容量阀值，每次 GEN0 对象数量超出阀值时，都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},
};
```

GC 首先检查 GEN2，如阀值被突破，那么合并 GEN2、GEN1、GEN0 几个追踪链表。如果没有超出，则检查 GEN1。GC 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，放到专门的列表等待回收。

```
>>> gc.get_threshold()          # 获取各级代龄阀值
(700, 10, 10)

>>> gc.get_count()              # 各级代龄链表跟踪的对象数量
(203, 0, 5)
```

包含 __del__ 方法的循环引用对象，永远不会被 GC 回收，直至进程终止。

这回不能偷懒用 __del__ 监控对象回收了，改用 weakref。因 IPython 对 GC 存在干扰，下面的测试代码建议在原生 shell 中进行。

```
>>> 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()                                # 因为没有达到阀值，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。
```

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

>>> xa = wa()
>>> xa, hex(id(xa.__dict__))
<__main__.User object at 0x10fd51fd0>, '0x7f990ac88280',

>>> xb = wb()
>>> xb, hex(id(xb.__dict__))
<__main__.User object at 0x10fd57050>, '0x7f990ac88940'
```

关于用不用 __del__ 的争论很多。大多数人的结论是坚决抵制，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，通过与机器无关的字节码来实现跨平台执行能力。这种字节码指令集没有寄存器，完全以栈 (抽象层面) 进行指令运算。尽管很简单，但对普通开发人员而言，是无需关心的细节。

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。还可用 -O 参数生成 pyo 格式，这是简单优化后的 pyc 文件。

编译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 文件和 Python 版本检查标记。PyCodeObject 则包含了代码对象的完整信息。

```
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;
```

无论是模块还是其内部的函数，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')
```

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、代码注入等话题更有兴趣，不妨看看标准库里的 dis。

1.6 执行

相比 .NET、JAVA 的 CodeDOM 和 Emit，Python 天生拥有无与伦比的动态执行优势。

最简单的就是用 eval() 执行表达式。

```
>>> 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义字典。

```
>>> 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__']
```

要执行代码片段，或者 PyCodeObject 对象，那么就需要动用 exec 。同样可以带入自定义名字空间，以避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 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)                              # 自定义名字空间。
>>> exec code in ns                                 # exec 执行以后，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。

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。

数据类型：

- 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可视为 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 类型是 64 位整数 (sys.maxint)，这显然能应对绝大多数情况。整数是虚拟机特殊照顾对象：

- 从堆上按需申请名为 PyIntBlock 的缓存区域存储整数对象。
- 使用固定数组缓存 [-5, 257) 之间的小数字，只需计算下标就能获得指针。
- PyIntBlock 内存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 PyIntBlock 内存只复用不回收，同时持有大量整数对象将导致内存暴涨，且不会在这些对象被回收后释放内存，造成事实上的内存泄露。

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。但换成 xrange 就不同了，每次迭代后，数字对象被回收，其占用内存空闲出来并被复用，内存也就不会暴涨了。

运行下面测试代码前，必须先安装 psutil 包，用来获取内存统计数据。

```
$ 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
```

在实际开发中，很少会遇到这样的情形。就算是海量整数去重、排序，我们也可用位图等算法来节约内存使用。Python 3 已经用 xrange 替换掉了默认的 range，我们使用 2.x 时稍微注意一下即可。

long

当超出 int 限制时， 会自动转换成 long。 作为变长对象，只要有内存足够，足以存储无法想象的天文数字。

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

>>> b = a + 1                    # 超出，自动使用 long 类型。
>>> type(b)
<type 'long'>

>>> 1 << 3000
12302319221611....8906122501351718891748990799112915123997738721785190182299893 76L

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

使用 long 的机会不多，Python 也就没有必要专门为其设计优化策略。

float

使用双精度浮点数 (float)，不能 "精确" 表示某些十进制的小数值。尤其是 "四舍五入 (round)" 的结果，可能和预想不同。

```
>>> 3 / 2                 # 除法默认返回整数，在 Python 3 中返回浮点数。
1

>>> float(3) / 2
1.5
```

```
>>> 3 * 0.1 == 0.3         # 这个容易导致莫名其妙的错误。
False

>>> round(2.675, 2)        # 并没有想象中的四舍五入。
2.67
```

如果需要，可用 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')                                    # float 转型精度不同
False

>>> Decimal('0.1') * 3 == Decimal('0.3')                                # decimal 没有问题
True

>>> Decimal('2.675').quantize(Decimal('.01'), ROUND_UP)                 # 精确控制 round
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，但没有特殊的 "小浮点数"。

2.2 字符串

与字符串相关的问题总是很多，比如池化 (intern)、编码 (encode) 等。字符串是不可变类型，保存字符序列或二进制数据。

- 短字符串存储在 arena 区域， str、unicode 单字符会被永久缓存。
- str 没有缓存机制，unicode 则保留 1024 个宽字符长度小于 9 的复用对象。
- 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，可以是单引号、双引号或三引号。但我个人建议用双引号表示字符串，用单引号表示字符，和其他语言习惯保持一致。

```
>>> "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"                         # 自动合并多个相邻字符串。
'abc'

>>> "中国人"                            # UTF-8 字符串 (Linux 系统默认)。
'\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba'

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

>>> u"中国人"                           # 使用 u 前缀定义 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'

>>> "abcabc".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

Python 2.x 默认采用 ASCII 编码。为了完成编码转换，必须和操作系统字符编码统一起来。

```
>>> import sys, locale

>>> sys.getdefaultencoding()           # Python 默认编码。
'ascii'

>>> 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。
```

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

- 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，除了熟悉的 C 样式外，还有更强大的 format。

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

标记：- 左对齐，+ 数字符号，# 进制前缀，或者用空格、0 填充。

```
>>> "%(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 方法支持更多的数据类型，包括列表、字典、对象成员等。

```
{field!convertflag:formatspec}
```

格式化规范：

```
formatspec: [[fill]align][sign][#][0][width][.precision][typecode]
```

示例：

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

>>> "{0},{1},{0}".format(1, 2)                       # field 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'
```

另有 string.Template 模板可供使用。该模块还定义了各种常见的字符序列。

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

>>> letters                                                    # 字母表
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> digits                                                     # 数字表
'0123456789'

>>> Template("$name, $age").substitute(name = "User1", age = 20) # 模板替换。
'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，池化有助于减少对象数量和内存消耗， 提升性能。

用 intern() 函数可以把运行期动态生成的字符串池化。

```
>>> 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。
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
```

当池化的字符串不再有引用时，将被回收。

2.3 列表

从功能上看，列表 (list) 类似 Vector，而非数组或链表。

- 列表对象和存储元素指针的数组是分开的两块内存，后者在堆上分配。
- 虚拟机会保留 80 个列表复用对象，但其元素指针数组会被释放。
- 列表会动态调整指针数组大小，预分配内存多于实际元素数量。

创建列表：

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

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

>>> ['a', 'b'] + ['c', 'd']                     # 连接多个列表。
['a', 'b', 'c', 'd']

>>> list("abcd")                                # 将序列类型或迭代器转换为列表。
['a', 'b', 'c', 'd']

>>> [x for x in range(3)]                       # 生成器表达式。
[0, 1, 2]
```

常见操作：

```
>>> l = list("abc")
>>> l[1] = 2                                    # 按序号读写。
>>> l
['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)                                    # 在指定位置插入元素。
>>> l
['a', 100, 'b', 'c']

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

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

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

可用 bisect 向有序列表中插入元素。

```
>>> import bisect

>>> l = ["a", "d", "c", "e"]
>>> l.sort()
>>> l
['a', 'c', 'd', 'e']

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

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

性能

列表用 realloc() 调整指针数组内存大小，可能需要复制数据。插入和删除操作，还会循环移动后续元素。这些都是潜在的性能隐患。对于频繁增删元素的大型列表，应该考虑用链表等数据结构代替。

下面的例子测试了两种创建列表对象方式的性能差异。为获得更好测试结果，我们关掉 GC，元素使用同一个小整数对象，减少其他干扰因素。

```
>>> import itertools, gc

>>> gc.disable()

>>> def test(n):
...     return len([0 for i in xrange(n)])          # 先创建列表，然后 append。

>>> def test2(n):
...     return len(list(itertools.repeat(0, n)))    # 按照迭代器创建列表对象，一次分配内存。

>>> 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，可以考虑用数组代替列表。 和列表存储对象指针不同，数组直接内嵌数据，既省了创建对象的内存开销，又提升了读写效率。

```
>>> import array

>>> a = array.array("l", range(10))           # 用其他序列类型初始化数组。
>>> a
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")                       # 从字符串添加元素。
>>> a
array('c', 'abc')

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

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

- 只读对象，元组和元素指针数组内存是一次性连续分配的。
- 虚拟机缓存 n 个元素数量小于 20 的元组复用对象。

在编码中，应该尽可能用元组代替列表。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)                                 # 少了逗号，就成了普通的括号运算符了。
>>> type(a)
<type 'int'>

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

>>> s = tuple("abcadef")                    # 将其他序列类型转换成元组。
>>> s
('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
```

标准库另提供了特别的 namedtuple，可用名字访问元素项。

```
>>> from collections import namedtuple

>>> User = namedtuple("User", "name age")     # 空格分隔字段名，或使用迭代器。

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

其实 namedtuple 并不是元组，而是利用模板动态创建的自定义类型。

2.5 字典

字典 (dict) 采用开放地址法的哈希表实现。

- 自带元素容量为 8 的 smalltable，只有 "超出" 时才到堆上额外分配元素表内存。
- 虚拟机缓存 80 个字典复用对象，但在堆上分配的元素表内存会被释放。
- 按需动态调整容量。扩容或收缩操作都将重新分配内存，重新哈希。

- 删除元素操作不会立即收缩内存。

创建字典：

```
>>> {}                                          # 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))}      # 使用生成表达式构造字典。
{'a': 0, 'c': 2, 'b': 1}
```

基本操作：

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

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

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

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

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

默认返回值：

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

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

>>> d.get("d", 123)                             # 如果没有对应 key，返回缺省值。
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，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()
<dictionary-keyiterator object at 0x10de82cb0>

>>> d.itervalues()
<dictionary-valueiterator object at 0x10de82d08>

>>> d.iteritems()
<dictionary-itemiterator object at 0x10de82d60>

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

视图

要判断两个字典间的差异，使用视图是最简便的做法。

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

>>> d1 & d2                             # 字典不支持该操作。
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（仅 v1 有，v2 没有的）
set([('a', 1)])

>>> v1 ^ v2                             # 对称差集（不会同时出现在 v1 和 v2 中）
set([('a', 1), ('c', 3)])

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

视图会和字典同步变更。

```
>>> d = {"a": 1}
>>> v = d.viewitems()
```

```
>>> v
dict_items([('a', 1)])

>>> d["b"] = 2
>>> v
dict_items([('a', 1), ('b', 2)])

>>> del d["a"]
>>> v
dict_items([('b', 2)])
```

扩展

当访问的 key 不存在时， defaultdict 自动调用 factory 对象创建所需键值对。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() 函数创建一个空列表作为 value。
>>> d["a"].append(2)
>>> d["a"]
[1, 2]
```

字典是哈希表，默认迭代是无序的。如果希望按照元素添加顺序输出结果，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) 用来存储无序不重复对象。所谓不重复对象，除了不是同一对象外，还包括 "值" 不能相同。集合只能存储可哈希对象，一样有只读版本 frozenset。

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

在内部实现上，集合和字典非常相似，除了 Entry 没有 value 字段。集合不是序列类型，不能像列表那样按序号访问，也不能做切片操作。

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

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

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

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

>>> s.pop()                                     # 弹出元素
'a'
```

```
>>> s
set(['c', 'b', 'd'])
```

集合和字典、列表最大的不同除了元素不重复外，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)， 仅左边有，右边没有的。
set(['c', 'd'])

>>> set("abx") ^ set("aby")                     # 对称差集 (symmetric_difference)
set(['y', 'x'])                                 #   不会同时出现在两个集合当中的元素。

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

更新操作:

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

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

```
>>> s = set("abx")
>>> s -= set("abcdy")                          # 差集 (difference_update)
>>> s
set(['x'])

>>> s = set("abx")
>>> s ^= set("aby")                            # 对称差集 (symmetric_difference_update)
>>> s
set(['y', 'x'])
```

集合和字典主键都必须是可哈希类型对象，但常用的 list、dict、set、defaultdict、OrderedDict 都是不可哈希的，仅有 tuple、frozenset 可用。

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

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

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

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

如果想把自定义类型放入集合，需要保证 hash 和 equal 的结果都相同才能去重。

```
>>> 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，这几个内置类型并不足以完成全部工作。像 C、数据结构、常用算法这类基础是每个程序开发人员都应该掌握的。

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 码的缘故。

```
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！"                    # 玛雅人都是骗人的！

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

也可以写成：

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

强制缩进

缩进是强制性的语法规则。通常建议用 4 个空格代替 TAB，好在多数编辑器都能自动转换。

最大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！"                    # 再次鄙视玛雅人！(*_*)
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，到行尾结束，不支持跨行。大段的描述可以用 """__doc__"""。

语句

可以用 ";" 将多条语句写在同一行，或者用 "\" 将一条语句拆分成多行。

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

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

a 1
```

某些 ()、[]、{} 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，它会合并对象所有成员的帮助内容。

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，且只能是下划线、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。
- 以双下划线开头的类成员名字视为私有。
- 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'
```

保留字 (包括 Python 3)：

```
False        class        finally      is           return
None         continue     for          lambda       try
True         def          from         nolcoal      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、nolocal 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名字空间，会出现下面这样的状况。

```
>>> def test():
...     while True:
...         x = 10
...         break
...     print locals()
...     print x                 # 这个写法在 C 里面会报错。

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

支持用序列类型或迭代器对多个名字同时赋值。

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

一旦值多过名字数量，会引发异常。要么切片，要么用 "_" 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，写得更简洁些。

```
>>> 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"
-
```

可以将两次比较合并成一个表达式。

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

条件表达式不能包含赋值语句，习惯此种写法的需要调整一下了。

```
>>> 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 多了个可选的 else 分支。如果循环没有被中断，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

更名为 foreach 可能更合适一些，用来循环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
```

同样有个可选的 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，也无法用 break、continue 跳出多层嵌套循环。

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

如果嫌 "跳出标记" 不好看，可以考虑抛出异常。

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

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

其实也没好看到哪去，但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 globals()
True

>>> del x
>>> "x" in globals()
False

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

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

>>> d = {"a":1, "b":2}
>>> del d["a"]                                  # key 不存在时，不会抛出异常。
>>> d
{'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)]                      # 列表
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

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

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

>>> (x for x in range(10))
<generator object <genexpr> at 0x10328a690>
```

可带上条件进行过滤。

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

或用多个 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，内层可引用外层名字。

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

甚至可直接用做函数实参。

```
>>> 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

运算符	说明
x & y, x | y, x ^ y	位运算
~x	位取反
x << y, x >> y	位移
x > y, x >= y	比较
x < y, x <= y	
x == y, x != y	相等
x is y, x is not y	同一对象
x in y, x not in y	包含 (序列、字典、迭代器)
not x	非
x and y, x or y	布尔
abs	绝对值
all	全部
any	任意
pow	幂
len	元素数量
min, max	最小、最大元素
divmod	(商，余数)
sum	统计 (可以带初始值)
cmp	比较

切片

序列类型支持 "切片 (slice)" 操作，可通过两个索引序号获取片段。

```
>>> x = range(10)

>>> x[2:6]                # [2, 6)
[2, 3, 4, 5]

>>> x[2:-2]               # [2, len(x) - 2)
[2, 3, 4, 5, 6, 7]
```

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]
```

可按切片范围删除序列元素。

```
>>> 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]
```

甚至不等长的切片替换。

```
>>> a = [1, 2, 3]
>>> a[:1] = ["a", "b", "c"]

>>> a
['a', 'b', 'c', 2, 3]
```

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，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 (?:)"。

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

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 "==" 可被重载，不适合用来判断两个名字是否指向同一对象。

```
>>> 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                 # is 总是判断指针是否相同。
False

>>> a == b                 # 通过 __eq__ 进行判断。
True
```

3.6 类型转换

各种类型和字符串间的转换。

```
>>> str(123), int('123')                            # int
>>> bin(17),  int('0b10001', 2)
>>> oct(20),  int('024', 8)
>>> hex(22),  int('0x16', 16)

>>> str(0.9), float("0.9")                          # float

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

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

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

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

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

3.7 常用函数

print

Python 2.7 可使用 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
```

用标准库中的 pprint.pprint() 代替 print，能看到更漂亮的输出结果。要输出到 /dev/null，可以使用 open(os.devnull, "w")。

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("$ ")
$ 1+2+3
6

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

Python 3 已经将 raw_input 重命名为 input。

用标准库 getpass 输入密码。

```
>>> 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 ExiCode = <number>。
- 返回非数字对象表示失败，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 或指定对象的名字空间。

```
>>> 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，或指定对象所有可访问成员 (包括基类)。

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

第 4 章 函数

当编译器遇到 def，会生成创建函数对象指令。也就是说 def 是执行指令，而不仅仅是个语法关键字。可以在任何地方动态创建函数对象。

一个完整的函数对象由函数和代码两部分组成。其中，PyCodeObject 包含了字节码等执行数据，而 PyFunctionObject 则为其提供了状态信息。

函数声明：

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

包括函数在内的所有对象都是第一类对象，可作为其他函数的实参或返回值。

- 在名字空间中，名字是唯一主键。因此函数在同一范围内不能 "重载 (overload)"。
- 函数总是有返回值。就算没有 return，默认也会返回 None。
- 支持递归调用，但不进行尾递归优化。最大深度 sys.getrecursionlimit()。

```
>>> 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)                # 默认参数
3

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

4.2 参数

函数的传参方式灵活多变，可按位置顺序传参，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
```

支持参数默认值。不过要小心，默认值对象在创建函数时生成，所有调用都使用同一对象。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。这两个名字只是惯例，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 必须是最后一个。

```
>>> 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 "展开" 序列类型和字典，将全部元素当做多个实参使用。如不展开的话，那仅是单个实参对象。

```
>>> 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，或者仅是字典的主键列表。"**" 展开字典键值对。但如果没有变参收集，展开后多余的参数将引发异常。

```
>>> 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 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!',
}
```

除非使用 global、nonlocal 特别声明，否则在函数内部使用赋值语句，总是在 locals 名字空间中新建一个对象关联。注意："赋值" 是指名字指向新的对象，而非通过名字改变对象状态。

```
>>> 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 -> 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。
- globals: 函数定义所在模块的名字空间。
- __builtins__: 内置模块的名字空间。

想想看，如果将对象引入 __builtins__ 名字空间，那么就可以在任何模块中直接访问，如同内置函数那样。不过鉴于 __builtins__ 的特殊性，这似乎不是个好主意。

```
>>> __builtins__.b = "builtins"

>>> g = "globals"

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

```
...     return test

>>> t = enclose()

>>> t()
locals
enclosing
globals
builtins
```

通常内置模块 __builtin__ 在本地名字空间的名字是 __builtins__ (多了个 s 结尾)。但要记住这说法一点也不靠谱，某些时候它又会莫名其妙地指向 __builtin__.__dict__。如实在要操作该模块，建议显式 import __builtin__。

27.3. __builtin__ — Built-in objects

CPython implementation detail: Most modules have the name __builtins__ (note the 's') made available as part of their globals. The value of __builtins__ is normally either this module or the value of this modules's __dict__ attribute. Since this is an implementation detail, it may not be used by alternate implementations of Python.

现在，获取外部空间的名字没问题了，但如果想将外部名字关联到一个新对象，就需要使用 global 关键字，指明要修改的是 globals 名字空间。Python 3 还提供了 nonlocal 关键字，用来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             # 声明 x, y 是 globals 名字空间中的。
...     x = 1000                # globals()["x"] = 1000
...     y = "Hello, World!"     # globals()["y"] = "..."。 新建名字。
...     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!')
```

没有 nonlocal 终归有点不太方便，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。因为拿到是调用者，而不一定是函数创建者。

4.4 闭包

闭包是指：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 对象。

```
>>> 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]
```

实现方式很简单，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 仅仅返回两个函数对象，并没有调用它们，自然不可能为它们创建堆栈帧。这样一来，就导致每次返回的 a 和 b 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，可以获知闭包所引用的外部名字。

- co_cellvars: 被内部函数引用的名字列表。
- co_freevars: 当前函数引用外部的名字列表。

```
>>> 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，还需注意 "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 堆栈帧基本上就是对 x86 的模拟，用指针对应 BP、SP、IP 寄存器。堆栈帧成员包括函数执行所需的名字空间、调用堆栈链表、异常状态等。

```
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_valuestack;    // 和 f_stacktop 共同维护运行帧空间，相当于 BP 寄存器。
    PyObject **f_stacktop;      // 运行栈顶，相当于 SP 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 IP 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 堆栈帧所在内存段。
} PyFrameObject;
```

可使用 sys._getframe(0) 或 inspect.currentframe() 获取当前堆栈帧。其中 _getframe() 深度参数为 0 表示当前函数，1 表示调用堆栈的上个函数。除用于调试外，还可利用堆栈帧做些有意思的事情。

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执行流程传递上下文对象。 inspect.stack 比 frame.f_back 更方便一些。

```
>>> 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
```

sys._current_frames 返回所有线程的当前堆栈帧对象。

虚拟机会缓存 200 个堆栈帧复用对象，以获得更好的执行性能。整个程序跑下来，天知道要创建多少个这类对象。

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, *d_args, **d_kwargs):

    def wrap(*args, **kwargs):
        new_args = d_args + args               # 合并位置参数，partial 提供的默认值优先。
        new_kwargs = d_kwargs.copy()           # 合并命名参数，partial 提供的会被覆盖。
        new_kwargs.update(kwargs)

        return func(*new_args, **new_kwargs)

    return wrap
```

提示：

与函数相关内容很多，涉及虚拟机底层实现。还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 文档中，实现接口通常被称为遵守协议。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，后者依次返回数据，直到引发 StopIteration 异常结束。

最简单的做法是用内置函数 iter()，它返回常用类型的迭代器包装对象。问题是，序列类型已经可以被 for 处理，为何还要这么做?

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

>>> d = Data()

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

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

返回迭代器对象代替 self._data 列表，可避免对象状态被外部修改。或许你会尝试返回 tuple，但这需要复制整个列表，浪费更多的内存。

iter() 很方便，但无法让迭代中途停止，这需要自己动手实现迭代器对象。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 Data(object):
...     def __init__(self, *args):
...         self._data = list(args)
...
```

```
...     def __iter__(self):
...         return DataIter(self)

>>> class DataIter(object):
...     def __init__(self, data):
...         self._index = 0
...         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 = Data(1, 2, 3)

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

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

除了 for 循环，迭代器也可以直接用 next() 操控。

```
>>> d = Data(1, 2, 3)

>>> it = iter(d)
>>> it
<__main__.DataIter object at 0x10dafe850>

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

>>> next(it)
StopIteration
```

5.2 生成器

基于索引实现的迭代器有些丑陋，更合理的做法是用 yield 返回实现了迭代器协议的 Generator 对象。

```
>>> class Data(object):
```

```
...     def __init__(self, *args):
...         self._data = list(args)
...
...     def __iter__(self):
...         for x in self._data:
...             yield x

>>> d = Data(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 对象。这样一来，就无须废力气维护额外的迭代器类型了。

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

>>> iter(d).next()
1
```

协程

yield 为何能实现这样的魔法？这涉及到协程 (coroutine) 的工作原理。先看下面的例子。

```
>>> def coroutine():
...     print "coroutine start..."
...     result = None
...     while True:
...         s = yield result
...         result = s.split(",")

>>> c = coroutine()                    # 函数返回协程对象。

>>> c.send(None)                       # 使用 send(None) 或 next() 启动协程。
coroutine start...

>>> c.send("a,b")                      # 向协程发送消息，使其恢复执行。
['a', 'b']

>>> c.send("c,d")
['c', 'd']

>>> c.close()                          # 关闭协程，使其退出。或用 c.throw() 使其引发异常。
```

```
>>> c.send("e,f")                          # 无法向已关闭的协程发送消息。
StopIteration
```

协程执行流程：

- 创建协程后对象，必须使用 send(None) 或 next() 启动。
- 协程在执行 yield result 后让出执行绪，等待消息。
- 调用方发送 send("a,b") 消息，协程恢复执行，将接收到的数据保存到 s，执行后续流程。
- 再次循环到 yeild，协程返回前面的处理结果，并再次让出执行绪。
- 直到关闭或被引发异常。

close() 引发协程 GeneratorExit 异常，使其正常退出。而 throw() 可以引发任何类型的异常，这需要在协程内部捕获。

虽然生成器 yield 能轻松实现协程机制，但离真正意义上的高并发还有不小的距离。可以考虑使用成熟的第三方库，比如 gevent/eventlet，或直接用 greenlet。

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 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，看上去就不那么舒服了。好好的逻辑被切分到两个函数里，维护也是个问题。有了 yield，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 变得复杂了一些，但却保持了 logic 的完整性。blocking style 样式的编码给逻辑维护带来的好处无需言说。

5.4 宝藏

标准库 itertools 模块是不应该忽视的宝藏。

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
18
```

cycle

迭代结束，再从头来过。

```
>>> 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('AAAABBBCCDAABBCCDD')]
[['A'], ['B'], ['C'], ['D'], ['A'], ['B'], ['C'], ['D']]

>>> [list(g) for k, g in groupby('AAAABBBCCDAABBCCDD')]
[['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。

```
>>> 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]
```

第 6 章 模块

不同于 C++、Java、C# namespace 仅作为符号隔离前缀，Python 模块是运行期对象。模块对应同名源码文件，为成员提供全局名字空间。

6.1 模块对象

模块对象有几个重要属性：

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

除使用 py 文件外，还可动态创建模块对象。

```
>>> 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.'}

>>> "sample" in sys.modules                                # 并没有添加到 sys.modules。
False

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

为模块动态添加函数成员时，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__
```

imp.new_module() 也可用来动态创建模块对象，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

当模块源文件发生变更时，可使用内置函数 reload() 重新导入模块。新建模块对象依旧使用原内存地址，只是原先被引用的内部成员对象不会被同步刷新。

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，且被 "刷新"。
```

如果改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模块 (包)：

- 当前进程根目录。
- PYTHONPATH 环境变量指定的路径列表。
- Python 标准库目录列表。
- 路径文件 (.pth) 保存的目录 (通常放在 site-packages 目录下)。

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 包文件或目录。
- so、dll、pyd 等扩展文件。
- 内置模块。
- 其他。

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

进程中的模块对象通常是唯一的。在首次成功导入后，模块对象被添加到 sys.modules，以后导入操作总是先检查模块对象是否已经存在。可用 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名字冲突，可用 as 更换别名。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__ 列表中未指定的对象。

在函数中使用 "import *" 会引发警告，虽然不影响使用，但应该避免引入用不到的名字。(Python 3 已经禁止该用法了)

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

__all__

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 *" 操作时，所有这些一并被带入到当前模块中，造成一定程度的污染。建议在模块中用 __all__ 指定可被批量导出的成员名单。

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

私有成员和 __all__ 都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，仅以特定的命名规则来提醒调用人员。

__import__

和 import 关键字不同，内置函数 __import__() 以字符串为参数导入模块。导入的模块会被添加到 sys.modules，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'>

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

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。
['__file__', ..., '__path__', 'add']

>>> 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__ 太麻烦，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()                            # 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)
```

注意：关键字 import 总是优先查找当前模块所在目录，而 __import__、import_module 则是优先查找进程根目录。所以用 __import__、import_module 导入包模块时，必须带上包前缀。

load_source

imp 另提供了 load_source()、load_compiled() 等几个函数，可用来载入不在 sys.path 搜索路径列表中的模块文件。优先使用已编译的字节码文件，模块对象会被添加到 sys.modules。

需要小心，这些函数类似 reload()，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

将多个模块文件放到独立目录，并提供初始化文件 __init__.py，就形成了包 (package)。

无论是导入包，还是导入包中任何模块或成员，都会执行初始化文件，且仅执行一次。可用来初始化包环境，存储帮助、版本等信息。

__all__

"from <package> import *" 仅导入 __init__.py 的名字空间，而该文件通常又只是个空文件，这意味着没有任何模块被导入。此时就需要用 __all__ 指定可以被导入的模块名字列表，该定义无需将模块显式引入到 __init__.py 名字空间。

```
$ cat test/__init__.py
__all__ = ["add"]
```

有太多理由不建议使用 "import *"，比如引入不需要的模块，意外 "覆盖" 当前空间同名对象等等。

换种做法，将要公开的模块和模块成员显式导入到 __init__.py 名字空间中，调用者只需 "import <package>"，然后用 "<package>.<member>" 就可访问所需的目标对象。如此可规避上述问题，还有助于隐藏包的实现细节，减少外部对包文件组织结构的依赖。

__path__

某些时候，包内的文件太多，需要分类存放到多个目录中，但又不想拆分成新的包或子包。这么做是允许的，只要在 __init__.py 中用 __path__ 指定所有子目录的全路径即可 (子目录可放在包外)。

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

```
$ cat test/__init__.py
__path__ = ["/home/yuhen/py/test/a", "/home/yuhen/py/test/b"]
```

稍微改进一下。还可以用 os.listdir() 扫描全部子目录，自动形成路径列表。

```
from os.path import abspath, join
subdirs = lambda *dirs: [abspath(join(__path__[0], sub)) for sub in dirs]

__path__ = subdirs("a", "b")
```

pkgutil

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() 可读取包内任何文件内容。

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

egg

将包压缩成单个文件，以便于分发和安装。类似 Java JAR 那样。

1. 安装 setuptools。

```
$ sudo easy_install setuptools
```

2. 创建空目录，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)
```

生成的 egg 文件存放在 dist 目录。

```
$ 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
```

将 test-0.0.9-py2.7.egg 全路径添加到路径文件 (.pth) 或 PYTHONPATH 环境变量就可使用。更最常见的做法是将其安装到 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

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 Classic Class。

Python 3 终于甩掉包袱，仅保留 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)                          # 2.x 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，关键字 class 的作用是创建类型对象。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) 存储了所有的静态字段和方法 (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。官方文档将所有成员统称为 Attribute。

```
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自己的名字空间。

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

>>> u.__dict__
{}
```

访问对象成员时，就从这几个名字空间中查找，而非以往的 globals、locals。

```
成员查找顺序: instance.__dict__ -> class.__dict__ -> baseclass.__dict__
```

注意分清对象成员和普通名字的差别。就算在对象方法中，普通名字依然遵循 LEGB 规则。

7.2 字段

字段 (Field) 和 属性 (Property) 是不同的。

- 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。
- 必须通过类型和实例对象才能访问字段。
- 以双下划线开头的 class 和 instance 成员视为私有，会被重命名。(module 成员不变)

```
>>> 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
```

可以在任何时候添加实例字段，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 class.<name> 外，也可以用 instance.<name>。按照成员查找规则，只要没有同名的实例成员，那么就继续查找 class.__dict__。

```
>>> 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'
```

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，可避免非正常逻辑修改。

私有字段以双下划线开头，无论是静态还是实例成员，都会被重命名: _<class>__<name>。

```
>>> 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,                    # 编码时无需关心重命名。
...                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，我们既想使用私有字段，又不想放弃外部访问权限。

- 用重命名后的格式访问。
- 只用一个下划线，仅提醒，不重命名。

不必过于纠结 "权限" 这个词，从底层来看，本就没有私有一说。

7.3 属性

属性 (Property) 是由 getter、setter、deleter 几个方法构成的逻辑。属性可能直接返回字段值，也可能是动态逻辑运算的结果。

属性以装饰器或描述符实现，原理以后再说。实现规则很简单，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"

>>> u.__dict__                  # 从 instance.__dict__ 可以看出属性和字段的差异。
{'_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__
__dict__     = <attribute '__dict__' of 'User' objects>
name         = <property object at 0x106ed6100>
```

从 class.__dict__ 可以看出，几个属性方法最终变成了 property object。这也解释了几个同名方法为何没有引发错误。既然如此，我们可以直接用 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__
__dict__     = <attribute '__dict__' 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__
{}
```

区别不大，只是 class.__dict__ 中保留了几个方法。

属性方法多半都很简单，用 lambda 实现会更加简洁。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 o: o._uid)                          # 只读属性。
...
...     name = property(lambda o: o._name, \                      # 可读写属性。
...                     lambda o, v: setattr(o, "_name", v))

>>> u = User(1)

>>> u.uid
1
>>> u.uid = 100
AttributeError: can't set attribute

>>> u.name = "Tom"
>>> u.name
'Tom'
```

不同于前面提过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"] = "xxxxxxxx"

>>> u.__dict__
```

```
{'_uid': 1, 'uid': 1000000, 'name': 'xxxxxxxx', '_name': 'Tom'}

>>> u.uid                                       # 访问的依旧是属性。
1

>>> u.name
'Tom'
```

尽可能使用属性，而不是直接暴露内部字段。

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
<bound method User.print_id of <__main__.User object at 0x10cf58b50>>

>>> u.print_id()
0x10cf58b50

>>> User.print_id
<unbound method User.print_id>

>>> User.print_id(u)
0x10cf58b50
```

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。

那么静态方法呢？为什么必须用 staticmethod、classmethod 装饰器？

```
>>> 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'>>
```

不使用装饰器的方法 a，将被当做了实例方法，自然不能以静态方法调用。

```
>>> User.a()
TypeError: unbound method a() must be called with User instance as first argument (got
nothing instead)
```

装饰器 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，可以有默认值、变参。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，总会有几个特殊的可选方法：

- __new__: 创建对象实例。
- __init__: 初始化对象状态。
- __del__: 对象回收前被调用。

```
>>> 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__ 可返回任意类型，但不同的类型会导致 __init__ 方法不被调用。

```
>>> 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
```

在方法里访问对象成员时，必须使用对象实例引用。否则会当做普通名字，依照 LEGB 规则查找。

```
>>> 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__，不可能出现同名主键，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, "CXO")

>>> m.__dict__                                      # 实例包含了所有基类的字段。
{'_age': 40, '_title': 'CXO', '_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用实例引用基类所有成员，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，那么首先被找到的是该静态成员，而不是基类的实例方法了。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
```

同样因为优先级的缘故，只需在派生类创建一个同名实例方法，就可实现 "覆盖 (override)"，签名可完全不同。

```
>>> 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。至于多重继承好不好，估计要打很久的口水仗。

```
>>> 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)。mro 和我们前面提及的成员查找规则是有区别的，__mro__ 列表中并没有 instance。所以在表述时，需要注意区别。

```
>>> 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

super() 起到其他语言 base 关键字的作用，它依照 mro 顺序搜索基类成员。

```
>>> 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)      # 可以考虑放在 __init__。
...         base.a()                   # A.a(self)
...         base.b()                   # B.b(self)

>>> C().test()
a
b
```

super 的类型参数决定了在 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。单继承时总是搜索该参数的基类型。

```
>>> 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):                             # 以 c instance 调用，那么
...         super(self.__class__, self).test()      #  self.__class__ 就是 C 类型对象。
...         print "b"                               #  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状况。

```
>>> 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()                      # 对输出结果很意外?
A                                #   super 按照 mro 列表顺序查找后续类型。
B                                #   那么在 A.__init__ 中的 super(A, self) 实际返回 B，
B                                #   super(A, self).__init__() 实际是 B.__init__()。

>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type 'object'>)
```

多重继承将很多问题复杂化，建议改用组合模式实现类似的功能。

__bases__

类型对象有两个相似的成员：

- __base__: 只读，总是返回 __bases__[0]。
- __bases__: 基类列表，可直接修改来更换基类，影响 mro 顺序。

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

>>> C.__bases__                  # 直接基类型元组
(<class '__main__.B'>,)

>>> C.__base__                   # __bases__[0]
<class '__main__.B'>

>>> C.__mro__                    # mro
(<class '__main__.C'>, <class '__main__.B'>, <type 'object'>)

>>> C.__bases__ = (A,)           # 通过 __bases__ 修改基类

>>> C.__base__                   # __base__ 变化
<class '__main__.A'>

>>> C.__mro__                    # mro 变化
(<class '__main__.C'>, <class '__main__.A'>, <type 'object'>)
```

对多继承一样有效，比如调整基类顺序。

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

>>> C.__bases__
(<class '__main__.A'>, <class '__main__.B'>)

>>> C.__base__
<class '__main__.A'>

>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type 'object'>)

>>> C.__bases__ = (B, A)         # 交换基类型顺序

>>> C.__base__                   # __base__ 总是返回 __bases__[0]
<class '__main__.B'>

>>> C.__mro__                    # mro 顺序也发生变化
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <type 'object'>)
```

通过更换基类，我们可实现代码注入 (Code Inject)，影响既有类型的行为。

抽象类

抽象类 (Abstract Class) 无法实例化，且派生类必须 "完整" 实现所有抽象成员才可创建实例。

```
>>> 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 o: o._uid)
...     name = property(lambda o: o._name, lambda o, v: setattr(o, "_name", v))
...     title = abstractproperty()

>>> class CXO(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: "CXO")

>>> c = CXO(1, "Tom")
>>> c.print_id()
1 Tom CXO
```

派生类 Manager 也是抽象类，它实现了部分基类的抽象成员，又增加了新的抽象成员。这种做法在面向对象模式里很常见，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，也是精华所在。即便是运行期，我们也可以随意改动对象，增加或删除成员。

增加成员时，要明确知道放到哪儿，比如将实例方法放到 instance.__dict__ 是没效果的。

```
>>> 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)                            # 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()                                # 测试通过。
0x10c91c090
```

静态方法必须用装饰器 staticmethod、classmethod 包装一下，否则会被当做实例方法。

```
>>> 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):                         # 注意 classmethod 和 staticmethod 的区别。
...     print "class method:", cls

>>> User.cstatic = classmethod(cstatic)

>>> 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'>
```

在运行期调整对象成员，时常要用到几个以字符串为参数的内置函数。其中 hasattr、getattr 依照成员查找规则搜索对象成员，而 setattr、delattr 则直接操作实例和类型的名字空间。

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

>>> setattr(u, "name", "tom")                 # u.name = "tom"

>>> u.__dict__
{'name': 'tom'}

>>> setattr(User, "table", "t_user")          # User.table = "t_user"

>>> User.table
't_user'

>>> u.table
't_user'
```

```
>>> hasattr(u, "table")                          # mro: User.__dict__["table"]
True

>>> getattr(u, "table", None)
't_user'

>>> delattr(u, "table")                          # Error: "table" not in u.__dict__
AttributeError: table

>>> delattr(User, "table")

>>> delattr(u, "name")                           # del u.__dict__["name"]
>>> u.__dict__
{}
```

__slots__

__slots__ 属性会阻止虚拟机创建实例 __dict__，仅为名单中的指定成员分配内存空间。这有助于减少内存占用，提升执行性能，尤其是在需要大量此类对象的时候。

```
>>> 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 = "CXO"                              # 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 = "CXO"
AttributeError: 'User' object has no attribute '_title'

>>> vars(u)                                      # 因为没有 __dict__，vars 失败。
TypeError: vars() argument must have __dict__ attribute
```

虽然没有了 __dict__，但依然可以用 dir() 和 inspect.getmember() 获取实例成员信息。

```
>>> 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'}
```

其派生类同样必须用 __slots__ 为新增字段分配存储空间 (即便是空 __slots__ = [])，否则依然会创建 __dict__，反而导致更慢的执行效率。

```
>>> class Manager(User):
...     __slots__ = ("_title")
...
...     def __init__(self, name, age, title):
...         User.__init__(self, name, age)
...         self._title = title
```

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__

先看看这几个方法的触发时机。

- __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__。

不要在这几个方法里直接访问对象成员，也不要用 hasattr/getattr/setattr/delattr 函数，因为它们会被再次拦截，形成无限循环。正确的做法是直接操作 __dict__。

而 __getattribute__ 连 __dict__ 都会拦截，只能用基类的 __getattribute__ 返回结果。

```
>>> 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__ 仅用于相等判断。

```
>>> 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
```

提示：

面向对象理论很复杂，涉及到的内容十分繁复，应该找本经典的大部头好好啃啃。

第 8 章 异常

异常不仅仅是错误，还是一种正常的跳转逻辑。

8.1 异常

除多了个可选的 else 分支外，与其他语言并无多大差别。

```
>>> 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，else 分支只在没有异常发生时执行。可无论如何，finally 总会被执行。

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!
```

下面这种写法已经被 Python 3 抛弃，不建议使用。

```
>>> def test():
...     try:
...         raise KeyError, "message"         # 相当于 KeyError("Message")
...     except (IndexError, KeyError), ex:    # 相当于 as 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() 获取调用堆栈上的最后异常信息。

```
>>> def test():
...     try:
...         raise KeyError("key error!")
...     except:
...         exc_type, exc_value, traceback = sys.exc_info()
...         sys.excepthook(exc_type, exc_value, traceback)       # 显示异常信息

>>> test()
Traceback (most recent call last):
```

```
    raise KeyError("key error!")
KeyError: 'key error!'
```

自定义异常通常继承自 Exception。应该用具体异常类型表示不同的错误行为，而不是 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 输出调试好得多。

```
>>> 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

上下文管理协议 (Context Management Protocol) 为代码块提供了包含初始化和清理操作的安全上下文环境。即便代码块发生异常，清理操作也会被执行。

- __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 = args
...
...     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 装饰器，用来简化上下文类型开发。

```
>>> 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 clsoing 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 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。contextmanager 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。

上下文管理协议的用途很广，比如：

- Synchronized: 为代码块提供 lock/unlock 线程同步。
- DBContext: 为代码块中的逻辑提供共享的数据库连接，并负责关闭连接。
- 等等……

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，但不仅仅是添加元数据。

```
>>> @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。

```
>>> 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：

- 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调用 wrap。

你完全可以把 "@" 当做语法糖，也可以直接使用函数式写法。只不过那样不便于代码维护，毕竟 AOP 极力避免代码侵入。

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)
```

用类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增加一些额外成员，那么可直接返回原类型。

```
>>> def action(cls):
...     cls.mvc = staticmethod(lambda: "Action")
...     return cls

>>> @action
... class Login(object): pass

>>> Login.mvc()
'Action'
```

这就是典型的 metaprogramming 做法了。

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：先传参数，后送类型。

```
>>> 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。
- 等等……

第 10 章 描述符

很少有人会去刻意关注描述符 (Descriptor)，尽管它时时刻刻以属性、方法的身份出现。

描述符协议：

```
__get__(self, instance, owner) --> return value
__set__(self, instance, value)
__delete__(self, instance)
```

描述符对象以类型 (owner class) 成员的方式出现，且最少要实现一个协议方法。最常见的描述符有 property、staticmethod、classsmethod。访问描述符类型成员时，解释器会自动调用与行为相对应的协议方法。

- 实现 __get__ 和 __set__ 方法，称为 data descriptor。
- 仅有 __get__ 方法的，称为 non-data descriptor。
- __get__ 对 owner_class、owner_instance 访问有效。
- __set__、__delete__ 仅对 owner_instance 访问有效。

```
>>> 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，不会有默认 __get__ 实现。

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
```

non-data

non-data descriptor 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__
```

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__                                # 类型对象，也就是隐式 cls 参数。
<class '__main__.Data'>

>>> m.__call__()
<class '__main__.Data'>
```

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)。那么类型对象又由谁来创建呢？—— 元类 (metaclass)，也就是类型的类型。

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>
```

现在可以理清几者的关系，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过程进行干预。

```
>>> 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                                # 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                                  # cls = Data
...         type.__init__(cls, name, bases, attrs)

>>> class Data(object):
...     __metaclass__ = InjectMeta                               # 自定义元类
...     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。

```
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 (想必你对他的 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，通常作为工具类 (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!
```

第二部分 标准库

本部分内容尚未校对……

第 12 章 字符串

12.1 re

正则表达式是处理字符串最重要的一种手段。

转义: . ^ $ * + ? { } [ ] \ | ( )	
定义:	
\d	数字，相当于 [0-9]。
\D	非数字字符，相当于 [^0-9]。
\s	空白字符，相当于 [\t\r\n\f\v]。
\S	非空白字符。
\w	字母或数字，相当于 [0-9a-zA-Z]。
\W	非字母或数字。
.	任意字符。
|	或。
^	非，或者开始位置标记。
$	结束位置标记。
\b	单词边界。
\B	非单词边界。
重复:	
*	0 或任意多个字符。添加 ? 后缀避免贪婪匹配。
?	0 或一个字符。
+	1 或多个字符。
{n}	n 个字符。
{n,}	最少 n 个字符。
{,m}	最多 m 个字符。
{n, m}	n 到 m 个字符。

编译: 可以直接在表达式前部添加 "(?iLmsux)" 标志	
s	单行。
i	忽略大小写。
L	让 \w 匹配本地字符，对中文支持不好。
m	多行。
x	忽略多余的空白字符。
u	Unicode。

正则函数

re 有几个重要的函数：

- match(): 匹配字符串开始位置。
- search(): 扫描字符串，找到第一个位置。
- findall(): 找到全部匹配，以列表返回。
- finditer(): 找到全部匹配，以迭代器返回。

match 和 search 仅匹配一次，匹配不到返回 None。

```
>>> import re

>>> s = "12abc345ab"

>>> m = re.match(r"\d+", s)
>>> m.group(), m.span()
('12', (0, 2))

>>> m = re.match(r"\d{3,}", s)
>>> m is None
True

>>> m = re.search(r"\d{3,}", s)
>>> m.group(), m.span()
('345', (5, 8))

>>> m = re.search(r"\d+", s)
>>> m.group(), m.span()
('12', (0, 2))
```

findall 返回列表 (或空列表)，finditer 和 match、search 一样返回 MatchObject 对象。

```
>>> ms = re.findall(r"\d+", s)
>>> ms
['12', '345']

>>> ms = re.findall(r"\d{5}", s)
>>> ms
[]

>>> for m in re.finditer(r"\d+", s): print m.group(), m.span()
...
12 (0, 2)
345 (5, 8)

>>> for m in re.finditer(r"\d{5}", s): print m.group(), m.span() # 返回空列表
...
>>>
```

MatchObject

match、search、finditer 返回的对象 —— MatchObject。

- group(): 返回匹配的完整字符串。
- start(): 匹配的开始位置。
- end(): 匹配的结束位置。
- span(): 包含起始、结束位置的元组。
- groups(): 返回分组信息。
- groupdict(): 返回命名分组信息。

```
>>> m = re.match(r"(\d+)(?P<letter>[abc]+)", s)

>>> m.group()
'12abc'

>>> m.start()
0

>>> m.end()
5

>>> m.span()
(0, 5)

>>> m.groups()
('12', 'abc')

>>> m.groupdict()
```

```
{'letter': 'abc'}
```

group() 可以接收多个参数，用于返回指定序号的分组。

```
>>> m.group(0)
'12abc'

>>> m.group(1)
'12'

>>> m.group(2)
'abc'

>>> m.group(1,2)
('12', 'abc')

>>> m.group(0,1,2)
('12abc', '12', 'abc')
```

start()、end() 和 span() 同样能接收分组序号。和 group() 一样，序号 0 表示整体匹配结果。

```
>>> m.start(0), m.end(0)
(0, 5)

>>> m.start(1), m.end(1)
(0, 2)

>>> m.start(2), m.end(2)
(2, 5)

>>> m.span(0)
(0, 5)

>>> m.span(1)
(0, 2)

>>> m.span(2)
(2, 5)
```

编译标志

可以用 re.I、re.M 等参数，也可以直接在表达式中添加 "(?iLmsux)" 标志。

- s: 单行。"." 匹配包括换行符在内的所有字符。
- i: 忽略大小写。
- L: 让 "\w" 能匹配当地字符，貌似对中文支持不好。
- m: 多行。

- x: 忽略多余的空白字符，让表达式更易阅读。
- u: Unicode。

试试看。

```
>>> re.findall(r"[a-z]+", "%123Abc%45xyz&")
['bc', 'xyz']

>>> re.findall(r"[a-z]+", "%123Abc%45xyz&", re.I)
['Abc', 'xyz']

>>> re.findall(r"(?i)[a-z]+", "%123Abc%45xyz&")
['Abc', 'xyz']
```

下面这么写好看多了吧?

```
>>> pattern = r"""
...     (\d+)     # number
...     ([a-z]+)  # letter
... """

>>> re.findall(pattern, "%123Abc\n%45xyz&", re.I | re.S | re.X)
[('123', 'Abc'), ('45', 'xyz')]
```

组操作

命名组：(?P<name>...)

```
>>> for m in re.finditer(r"(?P<number>\d+)(?P<letter>[a-z]+)", "%123Abc%45xyz&", re.I):
...     print m.groupdict()
...
{'number': '123', 'letter': 'Abc'}
{'number': '45', 'letter': 'xyz'}
```

无捕获组：(?:...)，作为匹配条件，但不返回。

```
>>> for m in re.finditer(r"(?:\d+)([a-z]+)", "%123Abc%45xyz&", re.I):
...     print m.groups()
...
('Abc',)
('xyz',)
```

反向引用：\<number> 或 (?P=name)，引用前面的组。

```
>>> for m in re.finditer(r"<a>\w+</a>", "%<a>123Abc</a>%<b>45xyz</b>&"):
...     print m.group()
...
```

```
<a>123Abc</a>

>>> for m in re.finditer(r"<(\w)>\w+</(\1)>", "%<a>123Abc</a>%<b>45xyz</b>&"):
...     print m.group()
...
<a>123Abc</a>
<b>45xyz</b>

>>> for m in re.finditer(r"<(?P<tag>\w)>\w+</(?P=tag)>", "%<a>123Abc</a>%<b>45xyz</
b>&"):
...     print m.group()
...
<a>123Abc</a>
<b>45xyz</b>
```

正声明 (?=...)：组内容必须出现在右侧，不返回。
负声明 (?!...)：组内容不能出现在右侧，不返回。
反向正声明 (?<=)：组内容必须出现在左侧，不返回。
反向负声明 (?<!)：组内容不能出现在左侧，不返回。

```
>>> for m in re.finditer(r"\d+(?=[ab])", "%123Abc%45xyz%780b&", re.I):
...     print m.group()
...
123
780

>>> for m in re.finditer(r"(?<!\d)[a-z]{3,}", "%123Abc%45xyz%byse&", re.I):
...     print m.group()
...
byse
```

更多信息请阅读官方文档或更专业的书籍。

修改

split: 用 pattern 做分隔符切割字符串。如果用 "(pattern)"，那么分隔符也会返回。

```
>>> re.split(r"\W", "abc,123,x")
['abc', '123', 'x']

>>> re.split(r"(\W)", "abc,123,x")
['abc', ',', '123', ',', 'x']
```

sub: 替换子串。可指定替换次数。

```
>>> re.sub(r"[a-z]+", "*", "abc,123,x")
'*,123,*'
```

```
>>> re.sub(r"[a-z]+", "*", "abc,123,x", 1)
'*,123,x'
```

subn() 和 sub() 差不多，不过返回 "(新字符串，替换次数)"。

```
>>> re.subn(r"[a-z]+", "*", "abc,123,x")
('*,123,*', 2)
```

还可以将替换字符串改成函数，以便替换成不同的结果。

```
>>> def repl(m):
...     print m.group()
...     return "*" * len(m.group())
...

>>> re.subn(r"[a-z]+", repl, "abc,123,x")
abc
x
('***,123,*', 2)
```

12.2 StringIO

提供类文件接口的字符串缓冲区，可选用性能更好的 cStringIO 版本。

```
>>> from contextlib import closing
>>> from cStringIO import StringIO

>>> with closing(StringIO("ab")) as f:
...     print >> f, "cd"
...     f.write("1234")
...     print f.getvalue()

abcd
1234
```

建议用 with 上下文确保调用 close() 方法释放所占用内存。用 getvalue() 返回字符串前，必须确保是打开状态 (closed = False)。

12.3 struct

struct 看上去有点像 format，区别是它输出的是二进制字节序列。可以通过格式化参数，指定类型、长度、字节序(大小端)、内存对齐等。

```
>>> from struct import *
```

```
>>> hexstr = lambda s: map(lambda c: hex(ord(c)), s)

>>> s = pack("i", 0x1234)

>>> hexstr(s)                                          # 4 字节整数小端排列
['0x34', '0x12', '0x0', '0x0']

>>> unpack("i", s)                                     # 还原。4660 = 0x1234
(4660,)

>>> s = pack(">i", 0x1234)                             # 大端

>>> hexstr(s)
['0x0', '0x0', '0x12', '0x34']

>>> s = pack("2i2s", 0x12, 0x34, "ab")                 # 多值。注意指定字符串长度。

>>> hexstr(s)
['0x12', '0x0', '0x0', '0x0', '0x34', '0x0', '0x0', '0x0', '0x61', '0x62']

>>> unpack("2i2s", s)
(18, 52, 'ab')
```

还可以将结果输出到 bytearray、array、ctypes.create_str_buffer() 等缓冲对象中。

```
>>> fmt = "3bi2s"
>>> size = calcsize(fmt)                # 计算指定格式转换所需的字节长度。

>>> buffer = bytearray(size)

>>> pack_into(fmt, buffer, 0, 0x1, 0x2, 0x3, 0x1FFFFF, "ab")

>>> buffer
bytearray(b'\x01\x02\x03\x00\xff\xff\x1f\x00ab')

>>> unpack_from(fmt, str(buffer), 0)
(1, 2, 3, 2097151, 'ab')
```

第 13 章 数据类型

13.1 bisect

bisect 使用二分法在一个 "已排序 (sorted) 序列" 中查找合适的插入位置。

```
>>> import bisect

>>> b = [ 20, 34, 35, 65, 78 ]

>>> bisect.bisect(b, 25)         # 查找 25 在列表中的合适插入位置。
1

>>> bisect.bisect(b, 40)         # 查找 40 在列表中的合适插入位置。
3

>>> bisect.bisect_left(b, 35)    # 如果待查找元素在列表中存在，则返回左侧插入位置。
2

>>> bisect.bisect_right(b, 35)   # 如果待查找元素在列表中存在，则返回右侧插入位置。
3
```

还可以直接用 insort_left() 直接插入元素而非查找。

```
>>> bisect.insort_left(b, 25)

>>> b
[20, 25, 34, 35, 65, 78]

>>> bisect.insort_left(b, 40)

>>> b
[20, 25, 34, 35, 40, 65, 78]
```

用 bisect 实现一个 SortedList 非常简单。

```
>>> def SortedList(list, *elements):
...     for e in elements:
...         bisect.insort_right(list, e)
...
...     return list

>>> SortedList([], 3, 7, 4, 1)
[1, 3, 4, 7]

>>> o = SortedList([], 3, 7, 4, 1)
```

```
>>> o
[1, 3, 4, 7]

>>> SortedList(o, 8, 2, 6, 0)
[0, 1, 2, 3, 4, 6, 7, 8]
```

可以考虑用 bisect 来实现 Consistent Hashing 算法，只要找到 Key 在 Ring 上的插入位置，其下一个有效元素就是我们的目标服务器配置。

13.2 heapq

最小堆: 完全平衡二叉树，所有节点都小于其子节点。

堆的意义：最快找到最大/最小值。在堆结构中插入或删除最小(最大)元素时进行重新构造时间复杂度为O(logN)，而其他方法最少为O(N)。堆在实际开发中的更倾向于算法调度而非排序。比如优先级调度时，每次取优先级最高的；时间驱动调度时，取时间最小或等待最长的等等。

```
>>> from heapq import *
>>> from random import *

>>> rand = sample(xrange(1000), 10)                          # 生成随机数序列。
>>> rand
[572, 758, 737, 738, 412, 755, 507, 734, 479, 374]

>>> heap = []
>>> for x in rand: heappush(heap, x)                         # 将随机数压入堆。
>>> heap                                                     # 堆是树，并非排序列表。
[374, 412, 507, 572, 479, 755, 737, 758, 734, 738]

>>> while heap: print heappop(heap)                          # 总是弹出最小元素。
374
412
479
507
572
734
737
738
755
758
```

其他相关函数。

```
>>> d = sample(xrange(10), 10)
>>> d
[9, 7, 3, 4, 0, 2, 5, 1, 8, 6]
```

```
>>> heapify(d)                          # 将列表转换为堆。
>>> d
[0, 1, 2, 4, 6, 3, 5, 9, 8, 7]

>>> heappushpop(d, -1)                  # 先 push(item)，后 pop。弹出值肯定小于或等于 item。
-1

>>> heapreplace(d, -1)                  # 先 pop，后 push(item)。弹出值可能大于 item。
0

... ...

>>> a = range(1, 10, 2)
>>> b = range(2, 10, 2)
>>> [x for x in merge(a, b)]            # 合并有序序列。
[1, 2, 3, 4, 5, 6, 7, 8, 9]

... ...

>>> d = sample(range(10), 10)
>>> d
[9, 0, 3, 4, 5, 6, 1, 2, 8, 7]

>>> nlargest(5, list)                   # 从列表(不一定是堆)有序返回最大的 n 个元素。
[9, 8, 7, 6, 5]

>>> nsmallest(5, list)                  # 有序返回最小的 n 个元素。
[0, 1, 2, 3, 4]
```

利用元组 __cmp__，用数字表示对象优先级，实现优先级队列。

```
>>> from string import *

>>> data = map(None, sample(xrange(100), 10), sample(letters, 10))
>>> data
[(31, 'Z'),
 (71, 'S'),
 (94, 'r'),
 (65, 's'),
 (98, 'B'),
 (10, 'U'),
 (8, 'u'),
 (25, 'p'),
 (11, 'v'),
 (29, 'i')]

>>> for item in data: heappush(heap, item)
>>> heap
```

```
[(8, 'u'),
 (11, 'v'),
 (10, 'U'),
 (25, 'p'),
 (29, 'i'),
 (94, 'r'),
 (31, 'Z'),
 (71, 'S'),
 (65, 's'),
 (98, 'B')]

>>> while heap: print heappop(heap)
(8, 'u')
(10, 'U')
(11, 'v')
(25, 'p')
(29, 'i')
(31, 'Z')
(65, 's')
(71, 'S')
(94, 'r')
(98, 'B')
```

或者重载自定义类型的 __cmp__ 操作符。

第 14 章 数学运算

14.1 random

伪随机数生成模块。如果不提供 seed，默认使用系统时间。

使用相同 seed，可获得相同的随机数序列，常用于测试。

```
>>> from random import *

>>> a = Random(); a.seed(1)

>>> [a.randint(1, 100) for i in range(20)]
[14, 85, 77, 26, 50, 45, 66, 79, 10, 3, 84, 44, 77, 1, 45, 73, 23, 95, 91, 4]

>>> b = Random(); b.seed(1)

>>> [b.randint(1, 100) for i in range(20)]
[14, 85, 77, 26, 50, 45, 66, 79, 10, 3, 84, 44, 77, 1, 45, 73, 23, 95, 91, 4]
```

使用示例

生成最大 N 个二进制位的长整数。

```
>>> getrandbits(5)
29L

>>> bin(getrandbits(5))
'0b11101'
```

生成 start <= N < stop 范围内的随机整数。

```
>>> randrange(1, 10)
2

>>> randrange(1, 10, 3)          # 支持步进
4

>>> randrange(1, 10, 3)
1

>>> randrange(1, 10, 3)
7
```

生成 a <= N <= b 范围内的整数。

```
>>> randint(1, 10)
5
```

从序列中随机返回元素。

```
>>> import string

>>> string.digits
'0123456789'

>>> choice(string.digits)
'6'

>>> choice(string.digits)
'1'

>>> choice(string.digits)
'3'
```

打乱序列，随机洗牌。

```
>>> a = range(10)

>>> shuffle(a)

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

从序列中随机挑选 n 个不同元素组合成列表。

```
>>> string.letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> sample(string.letters, 10)
['I', 'F', 'W', 'O', 'r', 'o', 'A', 'K', 'i', 'h']

>>> "".join(sample(string.letters, 10))       # 生成指定长度的随机字符串很容易
'kMmSgPVWIi'

>>> "".join(sample(string.letters, 10))
'feCTyRZrHv'
```

生成 0.0 <= N < 1 的随机浮点数。

```
>>> random()
0.39559451765020448

>>> random()
0.62378508101496177
```

生成 min <= N <= max 范围内的随机浮点数。

```
>>> uniform(1, 10)
7.6889886379206587

>>> uniform(10, 1)
5.1617099528426609
```

该模块还支持三角、β分布、指数分布、伽马分布、高斯分布等非常专业的随机算法。

第 15 章 文件与目录

15.1 file

通常建议用内置函数 open() 打开文件，file 用于类型判断。

```
>>> with open("test.txt", "w") as f:
...     print isinstance(f, file)                    // 类型判断
...     f.writelines(map(str, range(10)))

True
```

File Object 实现了上下文协议，可确保文件被及时关闭。实际上，文件对象被回收时总是会调用 close 方法，所以可以写下面这样的代码。

```
>>> open("test.txt", "r").read()
'0123456789'
```

如果要把数据写到磁盘上，除调用 flush() 外，还得用 sync()，以确保数据从系统缓冲区同步到磁盘。close() 总是会调用这两个方法。

打开模式：

- r: 只读。
- w: 只写。已存在文件将被清除 (truncate)。
- a: 添加。总是添加到文件尾部。
- b: 二进制模式。
- r+: 更新文件，可读写，不会截短文件。
- w+: 更新文件，可读写，清除原有内容。
- a+: 更新文件，可读写，总是在尾部添加。

文件对象还实现了迭代器协议，可直接循环获取其内容。

```
>>> with open("main.py", "r") as f:
...     for line in f: print line
...
```

读方法总能判断不同平台的换行标记，但写方法不会添加任何换行字符，包括 writelines。

```
>>> with open("test.txt", "w") as f:
...     f.write("a")
...     f.writelines("bc")
```

```
>>> cat test.txt
abc
```

如必须按不同平台写入换行标记，可使用 os.linesep。

```
>>> os.linesep
'\n'
```

字符串本身就是序列类型，可以直接用 writelines(str)。readline() 会返回包括换行符在内的整个行数据。通常建议用迭代器或 xreadlines() 代替 readlines()，后者默认一次性读取整个文件。

15.2 binary

用 struct 将其他类型构建成二进制字节数组，然后写入文件即可。

```
>>> import struct

>>> data = struct.pack("2i2s", 0x1234, 0xFF56, "ab")
>>> open("test.dat", "w").write(data)

>>> !xxd -g 1 test.dat
0000000: 34 12 00 00 56 ff 00 00 61 62                    4...V...ab

>>> struct.unpack("2i2s", open("test.dat").read())
(4660, 65366, 'ab')

>>> with open("test.dat") as f:               // 结构化读取
...     def xread(fmt):
...         n = struct.calcsize(fmt)          // 计算长度
...         s = f.read(n)
...         return struct.unpack(fmt, s)
...     print xread("i")
...     print xread("i")
...     print xread("2s")

(4660,)
(65366,)
('ab',)
```

对于相同类型的数据，可考虑用 array，以获得更好的性能。

```
>>> import array

>>> datas = array.array("i")
>>> datas.append(0x1234)
>>> datas.append(0xFF56)
```

```
>>> datas.tofile(open("test.dat", "w"))

>>> !xxd -g 1 test.dat
0000000: 34 12 00 00 56 ff 00 00                          4...V...

>>> d2 = array.array("i")
>>> d2.fromfile(open("test.dat"), 2)
>>> d2
array('i', [4660, 65366])
```

类似的还有 bytearray，可作 Buffer 用，详情参见 struct 章节。

15.3 encoding

标准库 codecs 提供了一个包装版的 open()，可自动完成编码转换工作。

```
>>> import sys
>>> reload(sys)
>>> sys.setdefaultencoding("utf-8")

>>> with codecs.open("test.txt", "w", "gbk") as f:
...     f.write("中国")

>>> !xxd -g 1 test.txt
0000000: d6 d0 b9 fa                                      ....

>>> "中国".encode("gbk")
'\xd6\xd0\xb9\xfa'

>>> s = codecs.open("test.txt", encoding = "gbk").read()
>>> s
u'\u4e2d\u56fd'
>>> print s
中国
```

15.4 descriptor

除使用文件对象外，某些时候还可能需要直接操控文件描述符。

```
>>> import os

>>> fd = os.open("test.txt", os.O_CREAT | os.O_RDWR, 0644)        // 注意是八进制。

>>> ls -l test.txt
-rw-r--r--  1 yuhen  staff  6  3 25 10:45 test.txt
```

```
>>> os.write(fd, "abc")
3

>>> f = os.fdopen(fd, "r+")             // 通过描述符创建文件对象。
>>> f.seek(0, os.SEEK_SET)              // 注意调整位置。
>>> f.read()
'abc'
>>> f.write("123")
>>> f.flush()                           // os 库提供的函数是系统调用，因此需要把数据从用户缓存
                                        //     刷新到系统缓存。

>>> os.lseek(fd, 0, os.SEEK_SET)
0

>>> os.read(fd, 100)
'abc123'

>>> os.close(fd)                        // 通常建议用和打开对应的方式关闭。
```

文件对象 fileno() 方法返回其对应的文件描述符。

15.5 tempfile

Python 对临时文件的支持算是我所见过语言中最丰富的。通常建议使用 NamedTemporaryFile，其他可以忽略。

- TemporaryFile: 创建临时文件对象，关闭时自动删除。
- NamedTemporaryFile: 创建临时文件对象，可获取文件名，参数决定是否自动删除。
- SpooledTemporaryFile: 和 TemporaryFile 类似，只有在数据超过阀值时，才写入硬盘。

```
>>> import tempfile, os.path

>>> tmp = tempfile.NamedTemporaryFile()
>>> tmp.name
'/var/folders/r2/4vkjhz6s6lz02hk6nh2qb99c0000gn/T/tmpYYB6p3'

>>> os.path.exists(tmp.name)
True

>>> tmp.close()
>>> os.path.exists(tmp.name)
False
```

默认使用系统临时目录和前缀，当然也可以指定不同的配置。

```
>>> with tempfile.NamedTemporaryFile(prefix = "xxx_", suffix = ".tmp", dir = ".") as f:
...     print f.name
```

```
...
/Users/yuhen/test/xxx_SL3apY.tmp
```

与临时文件有关的函数还有：

- tempfile.gettempdir: 返回系统临时文件存放路径。
- tempfile.gettempprefix: 返回默认的临时文件名前缀。
- tempfile.mkdtemp: 创建临时目录。
- tempfile.mkstemp: 创建临时文件，返回描述符和文件名，需手工删除。
- os.tempnam: 仅返回有效的临时文件名，并不创建文件。
- os.tmpfile(): 创建临时文件对象，关闭后自动删除。

```
>>> tempfile.gettempdir()
'/var/folders/r2/4vkjhz6s6lz02hk6nh2qb99c0000gn/T'

>>> tempfile.gettempprefix()
'tmp'
```

```
>>> d = tempfile.mkdtemp(); d
'/var/folders/r2/4vkjhz6s6lz02hk6nh2qb99c0000gn/T/tmpE_bRWd'

>>> os.path.exists(d)
True

>>> os.removedirs(d)
```

```
>>> fd, name = tempfile.mkstemp()

>>> os.write(fd, "123\n")
4

>>> os.close(fd)

>>> os.path.exists(name)
True

>>> os.remove(name)
```

15.6 os.path

常用函数列表：

函数	说明
abspath	绝对路径。

函数	说明
relpath	相对路径。
realpath	符号链接的真实路径。
normpath	将拼接的路径还原成正常样式。
basename	文件名。以 / 结尾的路径，返回空。
dirname	目录名。
commonprefix	多个路径的共有父目录。
exists	判断路径是否存在，失效符号链接返回 False。
lexists	判断符号连接文件自身是否存在。
expanduser	展开用户根目录 ~ 开始的路径。
expandvars	展开路径中的环境变量。如: ${name} 或 Windows %name%。
getatime	最后访问时间。通常指读操作。
getmtime	最后修改时间。
getctime	Windows 返回创建时间，Linux 返回属性更新时间。
getsize	文件大小。
isabs	判断是否绝对路径。
isdir	判断是否目录。
isfile	判断是否文件。
islink	判断是否符号链接。
ismount	判断是否载入点。
split	将路径分解成 (目录, 文件名)。
splitext	将路径分解成 (目录/主文件名，扩展名)。
join	拼接路径。

拼接的目录看上乱糟糟让人烦心。

```
>>> os.path.normpath("./../a/b/../c")
'../a/c'
```

展开用户根路径，或者包含系统环境变量的路径。

```
>>> os.path.expanduser("~/.vimrc")
'/Users/yuhen/.vimrc'
```

```
>>> os.path.expandvars("$HOME/.vimrc")
'/Users/yuhen/.vimrc'
```

除非只要扩展名，否则还是先用 basename 将路径去掉。

```
>>> os.path.splitext(os.path.basename("/usr/local/lib/libevent.a"))
('libevent', '.a')
```

15.7 os

常用函数列表：

函数	说明
chdir	修改工作目录。
getcwd	获取工作目录。
listdir	获取目录下所有成员，不支持递归和通配符。
walk	深度遍历所有子目录成员。
mkdir	创建子目录。如目标已存在，异常。
makedirs	递归创建多级子目录。
rmdir	删除空目录。
removedirs	递归删除目录中所有深度子目录。如非空则抛出异常。
remove	删除文件，如果是目录则抛出异常。
rename	重命名。如目标目录已存在，抛出异常。如果目标是文件，覆盖。
renames	重命名，必要的话会创建目标路径。
chmod	修改权限。lchmod 修改符号文件本身。
chown	修改拥有人。lchown 修改符号文件本身。
access	权限测试。
link	创建硬链接。
symlink	创建符号链接。
readlink	获取符号链接指向的目标。
unlink	解除链接。
stat	获取文件属性。lstat 返回符号文件本身属性。

函数	说明
utime	修改时间，参数 None 表示当前系统时间，相当于 touch 命令。

迭代 walk，返回 "(路径，子目录列表，文件列表)"，可配合 fnmatch 做通配符过滤。

```
>>> for path, dirs, files in os.walk("."):
...     for f in files:
...         if fnmatch.fnmatch(f, "*.py"):
...             print os.path.join(path, f)

./main.py
./bak/amqplib_test.py
./bak/eventlet_test.py
./bak/extract_text.py
./bak/fabric_test.py
```

如果仅操作当前目录，可以用 glob 代替 listdir，前者支持通配符。

```
>>> glob.glob("./bak/[rs]*.py")                      # 迭代器版本: iglob
['./bak/redis_test.py', './bak/socket_test.py']
```

如目录中还有文件存在，removedirs 会抛出异常。建议用 shutil.rmtree() 代替，注意参数区别。

```
>>> os.makedirs("./a/b/c")
>>> open("./a/b/c/test.txt", "w").write("abc")

>>> os.removedirs("./a/b/c")
OSError: [Errno 66] Directory not empty: './a/b/c'

>>> import shutil
>>> shutil.rmtree("./a")
```

某些时候，需要先测试文件是否拥有某些权限。

```
>>> os.access("a.txt", os.W_OK)
True
```

都是哪些人需要修改文件时间?

```
>>> !stat -x a.txt
  File: "a.txt"
  Size: 0            FileType: Regular File
  Mode: (0644/-rw-r--r--)         Uid: (  501/   yuhen)  Gid: (   20/   staff)
Device: 1,2   Inode: 5111644    Links: 1
Access: Mon Mar 25 17:43:01 2013
Modify: Mon Mar 25 17:43:01 2013
Change: Mon Mar 25 17:43:01 2013
```

```
>>> atime = time.mktime(datetime.datetime(2010, 10, 1).utctimetuple())
>>> mtime = time.mktime(datetime.datetime(2010, 11, 2).utctimetuple())
>>> os.utime("a.txt", (atime, mtime))

>>> os.stat("a.txt").st_atime == atime
True
```

获取文件权限信息时，别忘了转换成八进制。

```
>>> oct(os.stat("a.txt").st_mode)
'0100644'
```

15.8 shutil

常用函数列表：

函数	说明
copyfile	拷贝文件内容。不包括权限等属性，且目标必须是包含文件名的路径。
copymode	仅拷贝权限，不包括拥有人和文件内容。
copystat	拷贝权限、时间等属性，不包括拥有人和内容。
copy	拷贝文件，包括权限属性。覆盖已有文件，目标可以是目录。
copy2	拷贝文件，然后调用 copystat。
copytree	拷贝目录树，包括权限等属性。
rmtree	删除目录树。
move	递归移动文件或目录树。支持跨文件系统操作。

copytree 可以指定多个忽略通配符，且必须确保目标路径不存在。

```
>>> shutil.copytree("./bak", "./b/bak", ignore = shutil.ignore_patterns("*.pyc",
"*.bak"))
```

第 16 章 数据存储

16.1 serialization

marshal

Python 专用的序列化算法，PyCodeObject 就是用该算法序列化后保存到 pyc 二进制文件。与具体的机器架构无关，但可能随 Python 版本发生变化。通常不建议用来存储自定义数据。

支持：None, bool, int, long, float, complex, str, unicode, tuple, list, set, frozenset, dict, code objects, StopIteration。容器元素必须是所支持类型，不能是递归引用。

```
>>> from marshal import dump, load, dumps, loads

>>> s = dumps(range(10))
>>> s
'[\n\x00\x00\x00i\x00\...\x00\x00'

>>> loads(s)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

保存序列化结果到文件。

```
>>> with file("test.dat", "w") as f:
...     dump(range(10), f)

>>> with file("test.dat", "r") as f:
...     print load(f)

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

pickle

应该用 cPickle 代替 pickle，按官方文档的说法有千倍的提升，且可相互替换。支持用户自定义类型，支持三种协议版本：

- 0: 使用可显示的 ASCII 字符编码，便于阅读和手工编辑。(默认)
- 1: 兼容早期 Python 版本的二进制格式。
- 2: 最有效的二进制编码格式。

```
>>> import pickle, cPickle

>>> s = "Hello, World!"
```

```
>>> d = cPickle.dumps(s, 2)
>>> d
'\x80\x02U\rHello, World!q\x01.'

>>> cPickle.loads(d)
'Hello, World!'

>>> pickle.loads(d)                    # 和 pickle 格式完全相同。
'Hello, World!'
```

同样有读写文件的 dump、load 函数。看看支持的数据类型：

- None, True, False
- int, long, float, complex
- str, unicode
- tuple, list, set, and dict (元素必须是支持类型)
- function (模块级别的函数)
- classe (模块级别的自定义类，非嵌套)
- instance (有 __dict__ 属性，或者实现 pickle protocol 协议)

看看对自定义类型的测试。

```
>>> class Data(object):
...     def __init__(self, x, y):
...         print "__init__"
...         self._x = x
...         self._y = y

>>> d = Data(100, 200)
__init__

>>> s = cPickle.dumps(d, 2)

>>> d2 = cPickle.loads(s)              # 反序列化并没有调用 __init__
>>> d2.__dict__
{'_x': 100, '_y': 200}
```

利用 pickle protocol 可以控制序列化的细节。比如下面例子中，我们不像保留 _y 字段。

```
>>> class Data(object):
...     def __init__(self, x, y):
...         self._x = x
...         self._y = y
...
...     def __getstate__(self):
...         d = self.__dict__.copy()
...         del d["_y"]
```

```
...         return d
...
...     def __setstate__(self, state):
...         self.__dict__.update(state)

>>> d = Data(10, 20)

>>> s = cPickle.dumps(d, 2)

>>> d2 = cPickle.loads(s)
>>> d2.__dict__
{'_x': 10}
```

16.2 shevle

将对象 pickle 序列化，然后保存到 anydbm 格式文件。anydbm 是个 KV 结构的数据库，可以保存多个序列化的对象。当然也可以选择使用 dbm、gdbm、bdb。

- flag: r 读, w 写, c 读写, n 新建、读写。
- protocol: pickle 版本。
- writeback: 允许将变更的对象同步到数据库。(还是显式修改保存比较好)

```
>>> import shelve
>>> from contextlib import closing

>>> with closing(shelve.open("test", protocol = 2)) as f:
...     f["a"] = dict(name = "Tom", age = 34, sex = "male")
...     f["b"] = (1, ["a", 3], "abcdefg")


>>> !xxd -g 1 -l 100 test.db
0000000: 00 06 15 61 00 00 00 02 00 00 04 d2 00 00 10 00  ...a............
0000010: 00 00 00 0c 00 00 01 00 00 00 01 00 00 00 00 08  ................
0000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000060: 00 00 00 00                                      ....

>>> with closing(shelve.open("test", protocol = 2)) as f:
...     print f["a"]
...     print f["b"]
...     print ["c"]

{'age': 34, 'name': 'Tom', 'sex': 'male'}
(1, ['a', 3], 'abcdefg')
['c']
```

第 17 章 数据压缩

第 18 章 格式解析

第 19 章 数据加密

第 20 章 操作系统

20.1 time

Unix-Like 系统使用自基准点以来消逝的秒数来表达绝对时间。

- 绝对时间: 某个绝对精确的时间值。如 2010-11-1 13:48:05 。
- 相对时间: 相对于某个时间的前后差。如 5分钟以前。
- epoch: 基准点。1970-01-01 00:00:00 UTC。
- UTC: 协调世界时。世界不同时区的一个基准，比如中国为 UTC+8。
- DST: 阳光节约时 (夏时制)。好在我国已经取消了，真麻烦。

用 time() 返回自 epoch 以来的秒数，gmtime()、localtime() 将其转换为 struct_time 结构体。

```
>>> from time import *

>>> t = time()
>>> t
1357761634.903692

>>> gmtime(t)                    # epoch -> UTC
time.struct_time(tm_year=2013, tm_mon=1, tm_mday=9, tm_hour=20, tm_min=0, tm_sec=34,
tm_wday=2, tm_yday=9, tm_isdst=0)

>>> localtime(t)                 # epoch -> Local (UTC+8)
time.struct_time(tm_year=2013, tm_mon=1, tm_mday=10, tm_hour=4, tm_min=0, tm_sec=34,
tm_wday=3, tm_yday=10, tm_isdst=0)
```

将 struct_time 转回 epoch。

```
>>> from calendar import timegm

>>> t = time()
>>> t
1357762219.162796

>>> utc = gmtime(t)              # epoch -> UTC
>>> timegm(utc)                  # UTC -> epoch
1357762219

>>> local = localtime(t)         # epoch -> local
>>> mktime(local)                # local -> epoch
1357762219
```

与 datetime 的转换，注意返回的是 localtime 时间。

```
>>> from datetime import datetime
>>> from time import time

>>> t = time()

>>> d = datetime.fromtimestamp(t)             # localtime 时间
>>> d
datetime.datetime(2013, 1, 10, 4, 20, 27, 301148)

>>> d.timetuple()
time.struct_time(tm_year=2013, tm_mon=1, tm_mday=10, tm_hour=4, tm_min=20, tm_sec=27,
tm_wday=3, tm_yday=10, tm_isdst=-1)
```

相关函数:

ctime: 将 epoch 转换为字符串。
asctime: 将 struct_time 转换为字符串。

```
>>> t = time()

>>> ctime(t)
'Thu Jan 10 04:26:01 2013'

>>> asctime(localtime(t))
'Thu Jan 10 04:26:01 2013'
```

clock: 返回当前进程消耗的CPU时间 (秒)。
sleep: 暂停进程 (秒，可以是小数，以便设置毫秒、微秒级暂停)。

```
>>> clock()
0.56022400000000006

>>> sleep(0.1)
```

strftime: 将 struct_time 格式化为字符串。
strptime: 将字符串格式化为 struct_time。

```
>>> t = time()

>>> s = strftime("%Y-%m-%d %H:%M:%S", localtime(t))
>>> s
'2013-01-10 04:27:39'

>>> strptime(s, "%Y-%m-%d %H:%M:%S")
time.struct_time(tm_year=2013, tm_mon=1, tm_mday=10, tm_hour=4, tm_min=27, tm_sec=39,
tm_wday=3, tm_yday=10, tm_isdst=-1)
```

timezone: 与 UTC 的时差。
tzname: 当前时区名称。

```
>>> timezone / 3600
-8

>>> tzname                          # 北京时间，China Standard Time
('CST', 'CST')
```

20.2 threading

尽管因为 GIL 的缘故，Python 多线程一直遭受种种非议。但作为多个并发执行流程，多线程是无法完全用 "手工" 切换的协程来替代的。

20.2.1 Thread

创建 Thread 实例，传入待执行函数。

```
>>> from threading import Thread, currentThread, activeCount

>>> def test(s):
...     print "ident:", currentThread().ident
...     print "count:", activeCount()
...     print s
...

>>> Thread(target = test, args = ("Hello",)).start()
ident: 4353970176
count: 3
Hello
```

除了标识符，还可以线程取个名字，这有助于调试。

还可以继承 Thread 实现自己的线程类。

```
>>> class MyThread(Thread):
...     def __init__(self, name, *args):
...         super(MyThread, self).__init__(name = name)
...         self.data = args
...
...     def run(self):
...         print self.name, self.data

>>> MyThread("abc", range(10)).start()
abc ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9],)
```

将线程 daemon 属性设为 True，那么表示这是一个背景线程，进程退出时不会等待该线程结束。

调用 join() 等待线程结束，可提供超时参数 (秒，浮点数设定更小粒度)。isAlive() 检查线程状态，join() 可多次调用。

```
>>> from time import sleep

>>> def test():
...     print "__thread_start__"
...     sleep(10)
...     print "__thread_exit__"

>>> def run():
...     t = Thread(target = test)
...     t.start()
...     t.join(2)                // 超时
...
...     print t.isAlive()        // 检查状态
...     t.join()                 // 再次等待
...
...     print "over!"

>>> run()
__thread_start__
True
__thread_exit__
over!
```

20.2.2 Lock

Lock 不支持递归加锁，也就是说即便在同一线程中，也必须等待锁释放。通常建议改用 RLock，它会处理 "owning thread" 和 "recursion level" 状态，对于同一线程的多次请求锁行为，只累加计数器。每次调用 release() 将递减该计数器，直到 0 时释放锁，因此 acquire() 和 release() 必须要成对出现。

threading 中的成员大多实现了上下文协议，尽可能用 with 代替手工调用。

```
>>> lock = RLock()

>>> def show(i):
...     with lock:                           // 递归请求锁
...         print currentThread().name, i
...         sleep(0.1)

>>> def test():
```

```
...     with lock:                              // 加锁
...         for i in range(5):
...             show(i)

>>> for i in range(2):
...     Thread(target = test).start()

Thread-1 0
Thread-1 1
Thread-1 2
Thread-1 3
Thread-1 4
Thread-2 0
Thread-2 1
Thread-2 2
Thread-2 3
Thread-2 4
```

20.2.3 Event

Event 通过通过一个内部标记来协调多线程运行。方法 wait() 阻塞线程执行，直到标记为 True。set() 将标记设为 True，clear() 更改标记为 False。isSet() 用于判断标记状态。

```
>>> def test():
...     e = Event()
...     def test():
...         for i in range(5):
...             e.wait()
...             e.clear()
...             print i
...
...     Thread(target = test).start()
...     return e

>>> e = test()

>>> e.set()
0
>>> e.set()
1
```

如果不调用 clear()，那么标记一直为 True，wait() 就不会发生阻塞行为。

在实际编程中，我们通常为每个线程准备一个独立的 Event，而不是多个线程共享，以避免未及时调用 clear() 时发生意外情况。

20.2.4 Condition

Condition 像 Lock 和 Event 的综合体，除基本的锁操作外，还提供了类似 yield 的功能。

在获取锁以后，可以调用 wait() 临时让出锁，当前线程被阻塞，直到 notify() 发送通知后再次请求锁来恢复执行。将 wait 当做 yield，那么 notify 就是 send。

可以将已有的锁对象传给 Condition。

```
>>> def t1():
...     with cond:
...         for i in range(5):
...             print currentThread().name, i
...             sleep(0.1)
...             if i == 3: cond.wait()

>>> def t2():
...     with cond:
...         for i in range(5):
...             print currentThread().name, i
...             sleep(0.1)
...         cond.notify()

>>> Thread(target = t1).start(); Thread(target = t2).start()
Thread-1 0
Thread-1 1
Thread-1 2
Thread-1 3                  // 让出锁
Thread-2 0
Thread-2 1
Thread-2 2
Thread-2 3
Thread-2 4
Thread-1 4                  // 重新获取锁，继续执行。
```

只有获取锁的线程才能调用 wait() 和 notify()，因此必须在锁释放前调用。

当 wait() 释放锁后，其他线程也可进入 wait 状态。notifyAll() 激活所有等待线程，让它们去抢锁然后完成后续执行。

```
>>> def test():
...     with cond:
...         for i in range(5):
...         print currentThread().name, i
...         sleep(0.1)
...         if i == 2: cond.wait()

>>> Thread(target = t1).start(); Thread(target = t1).start()
```

```
Thread-1 0
Thread-1 1
Thread-1 2                              // Thread-1: 等待
Thread-2 0
Thread-2 1
Thread-2 2                              // Thread-2: 等待

>>> with cond: cond.notifyAll()         // 通知所有 cond.wait 线程。
Thread-2 3                              // Thread-1 和 Thread-2 再次抢锁以完成后续执行，
Thread-2 4                              //    至于谁先抢到，就不好说了。
Thread-1 3
Thread-1 4
```

20.2.5 Semaphore

Semaphore 通过一个计数器来限制可同时运行的线程数量。计数器表示还可以运行的线程数量，acquire() 递减计数器，release() 则是增加计数器。

```
>>> sem = Semaphore(2)

>>> def test():
...     with sem:
...         for i in range(5):
...             print currentThread().name, i
...             sleep(0.1)

>>> for i in range(3):
...     Thread(target = test).start()

Thread-1 0                // 1 和 2 同时执行。因为计数器为 0，所以 3 被阻塞。
Thread-2 0
Thread-1 1
Thread-2 1
Thread-1 2
Thread-2 2
Thread-1 3
Thread-2 3
Thread-1 4
Thread-2 4
Thread-3 0                // 1 和 2 释放信号量，3 开始执行。
Thread-3 1
Thread-3 2
Thread-3 3
Thread-3 4
```

20.2.6 Timer

用一个独立线程在 n 秒后执行某个函数。如定时器尚未执行，可用 cancel() 取消，定时器仅执行一次。

```
>>> def test():
...     print datetime.datetime.now()

>>> Timer(2, test).start()
2013-03-26 11:06:19.840455
```

20.2.7 Local

TLS (thread-local storage) 为线程提供独立的存储空间。

```
>>> data = local()

>>> def test(fn, x):
...     data.x = x
...     for i in range(5):
...         data.x = fn(data.x)
...         print currentThread().name, data.x
...         sleep(0.1)

>>> t1 = (lambda x: x + 1, 0)
>>> t2 = (lambda x: x + "a", "a")

>>> for d in (t1, t2):
...     Thread(target = test, args = d).start()

Thread-1 1
Thread-2 aa
Thread-2 aaa
Thread-1 2
Thread-2 aaaa
Thread-1 3
Thread-2 aaaaa
Thread-1 4
Thread-1 5
Thread-2 aaaaaa
```

20.3 multiprocessing

看上去和 threading 类似，区别在于用进程代替线程。这是规避 GIL，实现多核并发的常用方法。

20.3.1 Process

创建子进程执行指定函数。

```
from multiprocessing import Process, current_process

def test(*args, **kwargs):
    p = current_process()
    print p.name, p.pid
    print args
    print kwargs

if __name__ == "__main__":
    p = Process(target=test, args=(1, 2), kwargs = {"a": "hello"}, name = "TEST")
    p.start()
    p.join()
```

输出:

```
TEST, 2570
(1, 2)
{'a': 'hello'}
```

方法 start() 创建子进程，然后在新进程中通过 run() 执行目标函数。构建参数 args、kwargs 会传递给目标函数。在父进程中用 join() 等待并获取子进程退出状态，否则会留下僵尸进程，除非父进程先终止。

从下例输出结果，可以看到 __init__() 在父进程执行，但 run() 已经是子进程了。

```
class MyProcess(Process):
    def __init__(self):
        print "init:", os.getpid()
        super(MyProcess, self).__init__()

    def run(self):
        print "run:", os.getpid()

if __name__ == "__main__":
    print "parent:", os.getpid()
    p = MyProcess()
    p.start()
    p.join()
```

输出:

```
parent: 12093
init: 12093
run: 12094
```

子进程不会调用退出函数，而且只有后台 (daemon) 进程才可捕获主进程退出信号，默认处理自然是终止子进程。另外，后台进程不能创建新的子进程，这将导致僵尸出现。

```
from os import getpid
from time import sleep
from signal import signal, SIGTERM
from multiprocessing import Process

def test():
    def handler(signum, frame):
        print "child exit.", getpid()
        exit(0)

    signal(SIGTERM, handler)
    print "child start:", getpid()
    while True: sleep(1)

if __name__ == "__main__":
    p = Process(target = test)
    p.daemon = True                     # 必须在 start() 前设置。
    p.start()

    sleep(2)                            # 给点时间让子进程进入 "状态"。
    print "parent exit."
```

输出:

```
child start: 12185
parent exit.
child exit. 12185
```

调用 terminate() 会立即强制终止子进程 (不会执行任何清理操作)。有关状态还有: is_alive()、pid、exitcode。

20.3.2 Pool

进程池。用多个可重复使用的后台 (daemon) 进程执行函数，默认数量和 CPU 核相等。

```
from multiprocessing import Pool

def test(*args, **kwargs):
    print args
    print kwargs
    return 123

if __name__ == "__main__":
    pool = Pool()
    print pool.apply(test, range(3), dict(a=1, b=2))
    pool.close()
    pool.join()
```

输出:

```
(0, 1, 2)
{'a': 1, 'b': 2}
123
```

调用 join() 等待所有工作进程结束前，必须确保用 close() 或 terminate() 关闭进程池。close() 阻止提交新任务，通知工作进程在完成全部任务后结束。该方法立即返回，不会阻塞等待。

使用异步模型时，callback 是可选的。

```
from multiprocessing import Pool
from time import sleep

def test(*args, **kwargs):
    sleep(2)
    return 123

def callback(ret):
    sleep(2)
    print "return:", ret

if __name__ == "__main__":
    pool = Pool()
    pool.apply_async(test, callback=callback)

    ar = pool.apply_async(test)
    print ar.get()

    pool.close()
    pool.join()
```

apply_async 返回 AsyncResult 实例，其 get([timeout])、wait()、successful() 等方法可获知任务执行状态和结果。

map() 和 imap() 用于批量执行，分别返回列表和迭代器结果。

```
from multiprocessing import Pool, current_process

def test(x):
    print current_process().pid, x
    return x + 100

def test2(s):
    print current_process().pid, s

if __name__ == "__main__":
    pool = Pool(3)

    print pool.map(test, xrange(5))
```

```
    pool.map(test2, "abc")
```

输出:

```
1566 0
1567 1
1566 3
1568 2
1567 4
[100, 101, 102, 103, 104]

1566 a
1568 b
1567 c
```

参数 chunksize 指定数据分块大小，如果待处理数据量很大，建议调高该参数。

```
if __name__ == "__main__":
    pool = Pool(3)
    print pool.map(test, xrange(10), chunksize=2)
```

输出:

```
1585 0                          # 实际输出顺序可能不同。
1585 1
1586 2
1586 3
1587 4
1587 5
1585 6
1585 7
1586 8
1586 9
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
```

20.3.3 Queue

Queue 是最常用的数据交换方法。参数 maxsize 限制队列中的数据项数量，这会影响 get/put 等阻塞操作。默认值无限制。

通常直接使用 JoinableQueue，其内部使用 Semaphore 进行协调。在执行 put()、task_done() 时调整信号量计数器。当 task_done() 发现计数值等于 0，立即通知 join() 解除阻塞。

```
from Queue import Empty
from multiprocessing import Process, current_process, JoinableQueue


def test(q):
    pid = current_process().pid
    while True:
        try:
```

```
            d = q.get(timeout=2)                    # 阻塞 + 超时。照顾生产者未及生产情形。
            print pid, d
            q.task_done()
        except Empty:
            print pid, "empty!"
            break


if __name__ == "__main__":
    q = JoinableQueue(maxsize=1000)

    map(q.put, range(5))                            # 未超出队列容量限制，不会阻塞。
    print "put over!"

    for i in range(3):                              # 创建多个 consumer。
        Process(target=test, args=(q,)).start()

    q.join()                                        # 等待任务完成。
    print "task done!"
```

输出:

```
put over!
2127 0
2127 1
2127 2
2127 3
2127 4
task done!
2127 empty!
2128 empty!
2129 empty!
```

或许你会考虑压入同等数量的 None 作为结束标志，但无法保证每个 Consumer 都能获取。

20.4 ctypes

标准库 ctypes 模块可以非常方便地调用动态库 (.so)，这有助于解决安全和性能问题。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int add(int x, int y)
{
    return x + y;
}

void inc(int* x)
```

```
{
    *x += 1;
}

void cprint(char* s)
{
    printf("%s: %s\n", __func__, s);
}
```

编译:

```
$ gcc -fPIC -shared -o test.so test.c
```

测试:

```
>>> from ctypes import *
>>> so = cdll.LoadLibrary("./test.so")

>>> so.add(10, 20)
30

>>> so.cprint("Hello, World!")
cprint: Hello, World!
22

>>> x = c_int(123)
>>> so.inc(byref(x))                          # 传入指针
124
>>> x
c_int(124)
```

当然也可以直接调用系统库的函数。

```
>>> libc = cdll.LoadLibrary("libc.dylib")     # Linux: libc.so.6

>>> libc.printf("Hi!\n")
Hi!
4

>>> import time
>>> time.time(), libc.time()
(1364284691.803043, 1364284691)
```

第 21 章 进程通信

21.1 subprocess

执行程序，获取返回码或输出信息。

- call: 返回 ExitCode。
- check_call: 如果 ExitCode != 0，抛出 CalledProcessError 异常。
- check_output: 返回输出信息。ExitCode != 0 抛出异常。

命令行参数可以用 shlex.split 分解成列表。

```
>>> from subprocess import *
>>> from shlex import split

>>> s = check_output(split("ls -l"))
>>> print s
total 0
drwx------+  4 yuhen  staff   136  1 11 07:40 Desktop
drwx------+ 10 yuhen  staff   340  1  4 01:53 Documents
drwx------+  4 yuhen  staff   136  1 11 08:35 Downloads
drwx------@ 56 yuhen  staff  1904  1 11 08:28 Library
drwx------+  3 yuhen  staff   102  9 22 15:20 Movies
drwx------+  5 yuhen  staff   170  1  9 19:37 Music
drwx------+  5 yuhen  staff   170  1  3 21:14 Pictures
drwxr-xr-x+  4 yuhen  staff   136  9 15 16:21 Public
```

如果需要获取 ExitCode，又不想看到输出信息。可以将 stdout 重定位到 /dev/null。

```
>>> null = open(os.devnull, "w")
>>> call(split("ls -l"), stdout = null, stderr = null)
0
```

官方建议用 subprocess 代替 os.system、os.spawn*、os.popen*、popen2.*、commands.* 这个传统用法。基于以后向 Python 3 迁移的需要，还是放弃所有打上 obsolete 标记的库。

除使用简便函数外，还可以创建 Popen 对象以获取更细节的控制。subprocess 不能控制终端和 TTY 交互程序，建议使用第三方库 Fabric 或 pexpect。进程信息可以用 psutil 获取。

22.2 signal

信号是软中断，提供了一种异步事件通知机制。Python 默认已经安装了一些信号处理器，比如 SIGPIPE 被忽略，SIGINT 引发 KeyboardInterrupt 异常，捕获 SIGTERM 调用退出函数。

常用信号

- SIGINT: 用户中断 (ctrl + c)。
- SIGTERM: 由 kill() 发送，进程终止。
- SIGCHLD: 子进程终止。
- SIGHUP: 终端会话终止。
- SIGSTP: 进程暂停 (ctrl + z)。
- SIGALRM: 告警。

注意: 信号 SIGKILL、SIGSTOP 不能被捕获。

signal

仅能在主线程调用 signal() 注册信号处理器函数，它会移除当前处理动作。可用 getsignal() 获取，在需要时重新注册。有两个特殊的处理器：SIG_IGN 忽略信号，SIG_DFL 默认处理。

试着用 SIGINT 代替 KeyboardInterrupt 异常来处理用户中断。

```
from signal import *
from time import time, sleep

def sig_handler(signum, frame):
    print "exit"
    exit(0)

def main():
    signal(SIGINT, sig_handler)

    while True:
        sleep(1)
        print time()

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

输出:

```
$ ./main.py
1357987332.33
1357987333.33
1357987334.33
^Cexit
```

中断信号被拦截，我们可以自主决定是否终止进程。在 GDB 里，用 SIGINT 来处理调试中断。也有一些软件用 SIGUSR1、SIGUSR2 作为外部通知事件，比如重启什么的。信号处理会被带入 fork() 创建的子进程。

pause

函数 pause() 会使进程休眠，直到进程接收到信号。信号要么被处理，要么终止进程。

```
def sig_handler(signum, frame):
    print "sig:", signum

def main():
    signal(SIGUSR1, sig_handler)

    while True:
        print time()
        pause()
```

如果收到 SIGUSR1 信号，则进程苏醒后显示时间，然后再次休眠。如是其他信号，进程终止。

alarm

在 n 秒后发送一个 SIGALRM 告警信号。或用 0 秒取消所有尚未到期的告警。

```
signal(SIGALRM, sig_alarm)       # 捕获信号
alarm(2)                         # 2 秒后发送告警信号。仅一次。
```

timer

用来设置在 seconds 秒后发出信号，并在此以后每隔 interval 秒重复发出信号。参数 which 决定了发出何种信号。

- ITIMER_REAL: SIGALRM
- ITIMER_VIRTUAL: SIGVTALRM
- ITIMER_PROF: SIGPROF

```
signal(SIGALRM, sig_alarm)
setitimer(ITIMER_REAL, 2, 5)     # 2 秒后首次发出信号，随后每隔 5 秒发一次。
```

将 seconds 设置为 0，将清除定时器。

第 22 章 网络编程

第 23 章 程序框架

23.1 cmd

可用 cmd 写出 mongo、redis-cli 那样的交互命令行客户端。支持常用快捷键和命令补全提示。

从 Cmd 继承，然后按需要重写相关的方法。比较郁闷的是 Cmd 是 Classic Class，建议用多继承加个 object 基类，否则无法使用 super 调用基类方法。

```
preloop
cmdloop
        precmd                  修正命令信息，返回 line。
        onecmd                  查找并执行 do_* 方法。          返回 stop 给 postcmd。
                do_*            比如 test 命令，对应 do_test。  返回 stop 给 onecmd。
                default         没找到 do_* 或无法解析的命令，默认显示错误提示。
                emptyline       不输入命令，直接回车。默认重复上次命名。

                do_shell        ! 调用系统命令。
                complete_*      参数补全。如果没有对应方法，默认调用 completedefault。

                help_*          help cmd 或 ? cmd 显示具体命令帮助信息 (__doc__)。
                do_help         help 默认显示全部命令列表，可重写本方法修改输出。
        postcmd                 接收 stop 参数，返回 True 终止 cmdloop 循环。
postloop
```

通过返回 stop 值决定是否继续 cmdloop 循环。好在函数默认返回 None，没必要显式 return。参数 line 不包括命令串。

相关属性：

```
prompt                  命令提示符。
identchars              有效命令字符，默认是数字、字母和下划线。
lastcmd                 最后一条命令。
intro                   介绍。
doc_header              命令帮助标题。
misc_header             没找到帮助时显示的标题。
undoc_header            没有 __doc__ 时显示的标题。
ruler                   帮助信息分隔线，默认 "="。
use_rawinput            默认 True。
```

示例:

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

from os import popen
from cmd import Cmd
from shlex import split as shsplit

class Shell(Cmd, object):
    intro = "TEST shell, version 0.0.0.1"
    prompt = "$ "

    def default(self, line):
        # 退出 (EOF, <ctrl> + d)
        if line in ("exit", "quit", "bye", "EOF"):
            print "bye..."
            return True

        # 未知命令
        super(Shell, self).default(line)

    def do_test(self, line):
        # 参数分割
        print shsplit(line)

    def complete_test(self, text, line, beginidx, endidx):
        # 参数补全
        args = ("a1", "a2", "b1", "b2")
        return args if not text else filter(lambda s: s.startswith(text), args)

    def do_shell(self, line):
        # 系统命令
        print popen(line).read()


if __name__ == "__main__":
    Shell().cmdloop()
```

还可以用 code.interact() 嵌入 Python Shell 交互环境。

23.2 shlex

shlex 是一个分割 Unix Shell 命令行参数的简单词法分析器。

```
>>> from shlex import split
```

```
>>> split("ls -l /usr/local")
['ls', '-l', '/usr/local']

>>> split('test a b "c d"')             # 对引号参数的支持
['test', 'a', 'b', 'c d']
```

如果要将分解的列表还原，可以用 subprocess.list2cmdline。

```
>>> from subprocess import list2cmdline

>>> args = split('test a b "c d"')

>>> list2cmdline(args)
'test a b "c d"'
```

复杂的参数处理，应该使用 argparse。

第 24 章 开发工具

第 25 章 运行时服务

第 26 章 语言服务

第三部分 扩展库

A. Fabric

通过 SSH 进行软件部署或系统管理。

设置

在连接目标主机之前，必须提供足够的配置信息。

- env.hosts: 目标主机列表。格式: ip, user@ip, user@ip:port。
- env.host_string: 单机，格式同 hosts。
- env.roledefs: 按角色定义主机列表。格式: {name:[host, ...]}。
- env.passwrod: 主机密码字典。格式: {host: password}，和 hosts 中保持格式一致。
- env.user: 默认用户名。
- env.port: 默认端口。
- env.password: 默认密码。
- env.parallel: 是否并行执行任务。
- env.skip_bad_hosts: 是否跳过无法连接的主机。
- env.timeout: 连接超时，默认 10 秒。
- env.warn_only: 出错时是否仅显示警告信息。默认 False 终止任务。

任务

任务就是些普通函数，可以直接用 execute 执行，或在命令行调用。

- 默认: 所有 env.hosts 或 host_string。
- 主机: host 单主机，host 多主机列表。
- 角色: role 单个角色，roles 多个角色名列表。

也可用装饰器设置这些参数。另外，最好显式关闭连接。

```
from fabric.api import *
from fabric.network import *

def rcmd(s): run(s)

env.user = "root"
env.password = "123456"
env.hosts = ["192.168.1.1", "192.168.1.2", "192.168.1.3"]
env.roledefs = {"A": ["192.168.1.1", "192.168.1.2"], "B":["192.168.1.3"] }

try:
    execute(rcmd, "uname -a", roles = ["A", "B"])
finally:
```

```
    disconnect_all()
```

颜色

fabric.colors 提供了一些颜色包装函数，可配合 print 显示一些需要特别注意的信息。

```
from fabric.colors import *
print(green("This text is green!"))
```

上下文

fabric.context_managers 为命令提供区域设置。

- cd: 切换主机工作目录。
- lcd: 切换本地目录。
- hide: 隐藏输出信息。
- quiet: 安静模式，hide('everything'), warn_only=True。
- path: 修改 PATH 环境变量，可选 append 或 prepend。

切换到合适的工作目录，减少命令行输入。

```
with cd("/etc"):
    run("pwd")                  # /etc
    with cd ("init.d"):
       run("pwd")               # /etc/init.d
```

操作

fabric.operations 包含用于任务的一些命令。

- get: 下载文件。
- put: 上传文件。
- run: 运行远程命令。
- sudo: 以超级用户执行命令。
- local: 运行本地命令。
- prompt: 用户输入。
- open_shell: 远程交互式环境。
- reboot: 重启主机。

最常用的是 run 命令，它执行远程命令，返回输出结果字符串。该字符串对象还有 failed、succeeded、 return_code、command、real_command 等属性用来检查运行结果。

```
out = run("uname -a")
```

```
print out.succeeded, out.failed, out.return_code
print out.command, out.real_command
```

其他

fabric.utils 提供了一些辅助操作。

- abort: 引发异常，终止任务执行。
- warn: 显示警告信息，但不终止。
- indent: 获取一个缩进字符串。
- fastprint, puts: 显示信息。

fabric.contrib.console 提供 confirm 函数，让用户输入 [Y/n] 确认信息。

```
if confirm("Continue?", default = False):
    out = run("uname -a")
```

fabric.contrib.files 提供了远程文件操作功能。

- append: 添加信息。
- comment: 按条件注释掉某些内容。
- contains: 是否包含特定内容。
- exists: 路径是否存在。

问题

(1) 对后台运行命令 nohup 支持不好，可以考虑用 screen、pexpect 等代替。

```
$ screen -d -m -S <session_name> [cmd args]; sleep 5
```

注意用 sleep 暂停，避免 screen 尚未运行，会话就结束。

(2) 用 open_shell 进入交互模式，Ctrl+C 导致 fabric 进程退出，而不是远程进程。

附录

A. 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

B. IPython

远超 Python shell 的增强版本，可以当做 "IDE" 使用。

命令	说明
%quickref	快速导引。
%magic	Magic Functions 详细说明。 %fun? 获取具体帮助。
%lsmagic	列出所有可用 Magic Functions。
obj?, obj??	获取对象信息，?? 返回更详细的信息，比如源码。
?obj.*abc*	返回对象匹配的成员。比如: str.is*
!, !!	执行系统命令，捕获输出结果为字符串或列表。
%doctest_mode	切换 shell 提示样式，包括提示符、输出等设置。
%pprint	Pretty-Print 开关。
%bookmark	目录书签。
%cd, %pwd, _dh	工作目录。
%dirs, %popd, %pushd	目录栈。
%ed, %edit	使用编辑器打开文件。
%debug	进入最后一次异常场景，pdb.pm()。
%pdb	pdb 开关。引发异常时是否进入调试状态。
%pdoc	查看对象 __doc__ 信息。
%psource	显示对象源码。
%pfile	查看包含指定对象的文件内容。
%pycat	按页查看文件。
%run	执行指定文件。
%prun, %time, %timeit	性能测试。
%psearch	在当前名字空间按通配符搜索名字。
%who, %whos	查看所有变量。
%env	输出环境变量。
%hist	历史命令列表。
%reset	重置环境，移除所有名字。

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；-p 使用上一次 ed 命令。
- %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
```

C. 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	continue: 继续执行，直到下一个有效断点。
cl	clear: 清除断点。
d, u	down/up: 在 call stack frame 列表间移动。
disable, enable	禁用或启用断点。
q	exit/quit: 退出 pdb。
ignore	暂时忽略某个断点，可设置忽略次数。
j	jump: 跳转到指定行。
l	list: 显示源码。
n	next: 继续执行下一行代码。
p, pp	pretty-print: 显示变量或表达式的结果。
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，进入函数内部。
tbreak	创建临时断点，命中后失效。
unt	until: 继续执行，直到大于当前行号或当前函数结束。
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 。

相关方法：

- pm: 切换到最后的异常现场。

D. PIP-install

easy_install 替代品，功能更丰富一些，建议使用 1.3 以上版本。

命令	说明
pip install <package>	在线安装包
pip install -U <package>	升级包
pip install <file/url>	安装文件包
pip uninstall <package>	删除包
pip search <name>	搜索包
pip list	显示已安装包列表
pip list -o	显示过期可更新的包
pip list -u	显示已经是最新版本的包
pip freeze	以 requirements 格式现实已安装包列表
pip show <package>	显示已安装包信息

指定版本:

```
$ pip install SomePackage==1.04
$ pip install SomePackage>=1.04
```

清单文件:

```
$ pip install -r requirements.txt
```

文本文件，每行一个包记录，如:

```
Package1
Package2==1.0.4
Package3>=2.0
```

可以用 pip freeze > requirements.txt 将当前已安装的包导出。

E. VirtualEnv

通常配合 VirtualEnvWrapper 使用，它提供一些更易于使用的命令。

```
$ pip install virtualenv virtualenvwrapper
```

在 ~/.profile 添加虚拟环境、项目根目录等环境配置。

```
export WORKON_HOME=$HOME/projects/.virtualenv
export PROJECT_HOME=$HOME/projects
source /usr/local/bin/virtualenvwrapper.sh
```

相关命令:

命令	说明
mkproject	创建环境和项目目录。
mkvirtualenv	仅创建所需环境。
mktmpenv	创建临时环境，退出后删除。
setvirtualenvproject	为环境指定项目目录。
lsvirtualenv	显示所有环境名单。
showvirtualenv	显示某个环境信息。
rmvirtualenv	删除某个环境，但不包括项目文件。
cpvirtualenv	复制环境。
workon	激活或切换环境。
deactivate	退出环境。
cdproject	进入当前项目目录。
cdvirtualenv	进入当前环境目录。
cdsitepackages	进入当前环境 site-packages 目录。
lssitepackages	显示当前环境 site-packages 目录内容。
add2virtualenv	将路径添加到 _virtualenv_path_extensions.pth。
toggleglobalsitepackages	控制全局 site-packages 是否有效。