

目 录

致谢

Introduction

如何写出优美的代码

 如何测试 Python 应用

 如何测试 Django 应用

 面向接口测试——Python mock 库

REPL 环境

 BPython

 IPython

 设置 Django Shell

科学计算

网络应用

 HTTP

 分布式系统

网站应用

 Context

 浅谈 web 框架

 web 服务器

 WSGI 服务器

 服务器最佳实践

 部署代码

 HTML 模板

网站的部署

 uWSGI

 Gunicorn

HTML 与 XML

 HTML

 lxml 与 requests

控制台程序

 sh

 wget

 progressbar

 colorama

[Goocy——把 CLI 程序变成 GUI](#)

数据库

[DB API](#)

[SQLAlchemy](#)

[DjangoORM](#)

[Pony ORM](#)

[peewee](#)

NoSQL 数据库

[Redis](#)

[MongoDB](#)

[Neo4j](#)

网络基础

[Twisted](#)

[gevent](#)

[PyZMQ](#)

图片处理

[PIL](#)

[QRCode](#)

[几种图片转字符算法介绍](#)

[验证码破解](#)

开发环境

[编辑器](#)

[IDE](#)

提高代码效率

[Context](#)

[Tools](#)

[Libraries](#)

[Resources](#)

设计模式

[单例模式](#)

文件处理

[python-magic](#)

Python Magic Methods

[Python 中的魔法方法](#)

Jenkins

Escaping

Misaka

致谢

当前文档《Python One to Million 中文版》由 进击的皇虫 使用 书栈 (BookStack.CN) 进行构建,生成于 2018-04-27。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能,以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理,书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候,发现文档内容有不恰当的地方,请向我们反馈,让我们共同携手,将知识准确、高效且有效地传递给每一个人。

同时,如果您在日常生活、工作和学习中遇到有价值有营养的知识文档,欢迎分享到 书栈 (BookStack.CN) ,为知识的传承献上您的一份力量!

如果当前文档生成时间太久,请到 书栈(BookStack.CN) 获取最新的文档,以跟上知识更新换代步伐。

文档地址: <http://www.bookstack.cn/books/Python-One-to-Million>

书栈官网: <http://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享,让知识传承更久远! 感谢知识的创造者,感谢知识的分享者,也感谢每一位阅读到此处的读者,因为我们都将成为知识的传承者。

Introduction

- [前言](#)
 - [本书不提供什么](#)
 - [本书提供什么](#)
 - [KISS](#)
 - [来源\(书栈小编注\)](#)

前言

学完基本的语法之后不知道做什么？根本不知道学了编程能做什么？

本“书”的目的就在于提供一个这样的 index，内容涉及到 Python 开发的方方面面，并提供对应实例代码&教程。本书致力成为这样一个目录——如果你想知道怎么入门某一方面的开发就来翻看这本书吧！

本书不提供什么

- 新手指南。相反，你应该在有一定实际应用能力之后再来阅读本书。
- 进阶教程。本书主要提供的事内容的广度，帮你看大更大的世界。
- 一本详尽的手册。篇幅有限，不可以在这里对每一个知识点都做出详细的介绍。
- 公正的评测。本书内容严重受到我自己的主观影响，然而这是不可避免的。

本书提供什么

- 一本索引。希望你在新的领域无所适从时能够帮助到你。
- 思想的碰撞。欢迎一起交流使用心得和入坑经验。

KISS

Keep It Simple, Stupid!

是的，如果你在文中遇到任何看不懂的地方，并且没有找到对应引用 / 介绍，那一定是因为我没有把问题讲清楚！大胆的给我发 Issue、Pull Request 或者评论留言吧！

参考文章：[The Hitchhiker's Guide to Python!](#)

来源(书栈小编注)

<https://github.com/kxxoling/Python-One-to-Million>

如何写出优美的代码

- [好的结构是关键](#)

好的结构是关键

- [如何测试 Python 应用](#)
- [如何测试 Django 应用](#)
- [面向接口测试—Python mock 库](#)

如何测试 Python 应用

- 测试
 - 基本概念
 - 单元测试
 - 集成测试
 - 系统测试
 - 基本工具
 - doctest
 - unittest 和 unittest2
 - py.test
 - web 相关
 - Django
 - Flask
 - Tornado
 - 浏览器
 - 其它工具
 - nose
 - tox
 - mock
 - Code Coverage
 - CI
 - 如何提高测试速度
 - 参考链接

测试

对于一个开源项目来说，文档和测试都是必不可少的组成部分，没有足够测试和文档覆盖率的“开源项目”就是一坨垃圾！当然，对于某些能够做到自文档的大神来说，文档可以是不必要的，但测试依旧是代码质量的保证。

优秀的测试通常遵循一下基本规章：

1. 每个测试单元应该关注于一个功能，并保证其正确性。
2. 测试单元之间应该尽可能独立，也就是说可以独立运行，与顺序无关。
3. 测试的速度应该尽可能快，过慢的测试速度会成为开发的瓶颈。对于耗费时间很长的重型测试，应该将其独立出来。
4. 在集中编程前后都应该完整地运行一遍测试，以保证不会造成意外的破坏。

5. 在编程过程中，如果需要中断工作，那么编写一个不能运行的测试对于恢复工作非常有帮助。
6. debug 的第一步就是写一个针对性的单元测试，虽然这做起来并不一定容易，但却非常有价值。
7. 虽然 PEP8 提倡简短的命名，但在测试函数名称应该长而有意义。比如，编程中你可能使用 `square()` 甚至 `sqr()` 这样的函数名称，但是在测试中你应该写成： `test_square_of_number_2()` , `test_square_negative_number()` 。
8. 对于新成员来说，阅读测试代码可能是他们了解系统的最快途径之一，热点、难点、边界情况都会一目了然。
因此，加入新功能的第一步应该是编写一个对应的单元测试。

基本概念

单元测试

单元测试是针对程序最小模块单位

进行正确性检验的测试工作。最小单位通常是函数或者方法。理想情况下，每一个单元测试应该独立于其它用例。

单元测试通常由软件开发人员编写，用于确保他们所写的代码符合软件需求和遵循开发目标。

在自动化测试时，为了实现隔离的效果，测试将脱离待测程序单元（或代码主体）本身固有的运行环境之外，

即脱离产品环境或其本身被创建和调用的上下文环境，而在测试框架中运行。

以隔离方式运行有利于充分显露待测试代码与其它程序单元或者产品数据空间的依赖关系。

这些依赖关系在单元测试中可以被消除。隔离模块经常会使用 `stubs`、`mock` 或 `fake` 等测试马甲程序。

集成测试

整合测试又称组装测试，即对程序模块采用一次性或增殖方式组装起来，对系统的接口进行正确性检验的测试工作。

整合测试一般在单元测试之后、系统测试之前进行。实践表明，有时模块虽然可以单独工作，但是并不能保证组装起来也可以同时工作。

系统测试

系统测试是将需测试的软件，作为整个基于计算机系统的一个元素，

与计算机硬件、外设、某些支持软件、数据和人员等其他系统元素及环境结合在一起测试。

在实际运行(使用)环境下，对计算机系统进行一系列的组装测试和确认测试。

系统测试的目的在于通过与系统的需求定义作比较，发现软件与系统定义不符合或与之矛盾的地方。

基本工具

doctest

Python 还提供了一个叫做 `doctest` 的工具，写法如下：

```

1. """
2. 一个最简单的 doctest 写法，我这种缩进是为了照顾 Sphinx 文档自动生成工具::
3.
4.     >>> factorial(5)
5.     120
6. """
7.
8. def factorial(n):
9.     """依旧是 doctest，不过更加复杂::
10.
11.         >>> [factorial(n) for n in range(6)]
12.         [1, 1, 2, 6, 24, 120]
13.         >>> [factorial(long(n)) for n in range(6)]
14.         [1, 1, 2, 6, 24, 120]
15.         >>> factorial(30)
16.         2652528598121910586363084800000000L
17.         >>> factorial(30L)
18.         2652528598121910586363084800000000L
19.         >>> factorial(-1)
20.         Traceback (most recent call last):
21.             ...
22.         ValueError: n must be >= 0
23.
24.         Factorials of floats are OK, but the float must be an exact integer:
25.         >>> factorial(30.1)
26.         Traceback (most recent call last):
27.             ...
28.         ValueError: n must be exact integer
29.         >>> factorial(30.0)
30.         2652528598121910586363084800000000L
31.
32.         It must also not be ridiculously large:
33.         >>> factorial(1e100)
34.         Traceback (most recent call last):
35.             ...
36.         OverflowError: n too large
37.     """
38.     import math
39.

```

```

40.     if not n >= 0:
41.         raise ValueError("n must be >= 0")
42.     if math.floor(n) != n:
43.         raise ValueError("n must be exact integer")
44.     if n+1 == n: # catch a value like 1e300
45.         raise OverflowError("n too large")
46.     result = 1
47.     factor = 2
48.     while factor <= n:
49.         result *= factor
50.         factor += 1
51.     return result
52.
53. if __name__ == "__main__":
54.     import doctest
55.     doctest.testmod()

```

如果不在代码中显式 `import doctest` 也可以在运行文件的时候输入这样的命令：`python -m doctest -v filename.py`。

从上面的示例代码中也可以看出，`doctest` 并便于不提供完整的边界数据测试的支持，因此并不能完全替代单元测试。

unittest 和 unittest2

Python 自带了 `unittest` 库，是 Java JUnit 库的 Python 实现，虽然很好用，但我还是想在这里吐槽一下驼峰式命名的方法。

在 Python 2.7 版本以后，`unittest.TestCase` 类自带了 `assertListEqual()` 等方法，非常便利，也是我不愿意兼容 Python 2.6 的重要原因。

附即将弃用的方法对照表：

方法名	即将弃用的方法名
<code>assertEqual()</code>	<code>failUnlessEqual</code> , <code>assertEquals</code>
<code>assertNotEqual()</code>	<code>failIfEqual</code>
<code>assertTrue()</code>	<code>failUnless</code> , <code>assert_</code>
<code>assertFalse()</code>	<code>failIf</code>
<code>assertRaises()</code>	<code>failUnlessRaises</code>
<code>assertAlmostEqual()</code>	<code>failUnlessAlmostEqual</code>
<code>assertNotAlmostEqual()</code>	<code>failIfAlmostEqual</code>

unittest2

是 `unittest` 的增强版本，几乎完全兼容 `unittest` 的接口，升级时只需要将 `unittest` 替换为 `import unittest2` 即可，提供的新方法更强大也更严谨。

pytest

`pytest` 是一个成熟的全功能测试框架。

web 相关

对于 web 功能的测试，最简单的可以使用 `urllib2.get(url)`，然后测试输出的 HTML 结果是否符合预期。

当然针对每一个功能都这样写未免太过低效，因此知名 web 框架大多有专门的测试库提供测试：

- Django 内置了 `django.test`
- Tornado 内置了 `tornado.testing`
- Flask 可以使用 `werkzeug.test` 和第三方的 `Flask-Testing`

Django

Django 的启动互相之间的依赖严重，大部分文件都不能单独执行，测试时建议使用封装后的工具，如：`django.test`、`django_nose` 等等。

Flask

Flask 在写测试的时候需要主要 `app_context` 和 `request_context` 中的陷阱。

Tornado

Tornado 的 `testing` 库很简陋，主要是针对自身异步特性封装了一些工具。

浏览器

浏览器端的测试自动化最常用的还是 `Selenium`，Python 版本的 [文档](#)并不复杂。示例代码：

```
1. import unittest
2. from selenium import webdriver
3.
4. class TestOne(unittest.TestCase):
5.
```

```

6.     def setUp(self):
7.         self.driver = webdriver.Firefox()           # 初始化浏览器，也可以选择 Chrome 或者 PhantomJS
8.         self.driver.set_window_size(1280, 550)
9.
10.    def test_url(self):
11.        self.driver.get("http://duckduckgo.com/")
12.        self.driver.find_element_by_id(
13.            'search_form_input_homepage').send_keys("realpython")
14.        self.driver.find_element_by_id("search_button_homepage").click()
15.        self.assertIn(
16.            "https://duckduckgo.com/?q=realpython", self.driver.current_url
17.        )
18.
19.    def tearDown(self):
20.        self.driver.quit()
21.
22. if __name__ == '__main__':
23.     unittest.main()

```

上面的代码会启动浏览器（这里设置的是 Firefox），并触发浏览器事件模拟用户输入。对于没有浏览器的机器，比如服务器或，可以配置远程 Selenium server 或者使用 Headless 的 PhantomJS 代替。使用 Headless 浏览器因为减少了打开和关闭浏览器的时间，因此在测试效率上也更高一些。

使用之后感触最深的是错误提示不够丰富，基本上只能断定页面结果并不符合预期，结果反馈跟 `unittest.TestCase` 简直天壤之别。

其它工具

nose

tox

发布独立库的时候通常要考虑不同版本间兼容性的问题，虽然可以通过 `virtualenv` 实现环境的模拟，但毕竟很不方便，`tox` 正是解决这一问题的工具。

`tox` 简化了 `virtualenv` 的管理，提供了简便的配置。我常用的配置是这样的：

```

1. # 文件 tox.ini 的内容，需要和 setup.py 置于同一目录
2. [tox]
3. envlist = py26,py27

```

```

4. [testenv]
5. deps=                # 测试依赖
6. commands=make test   # 执行测试的命令

```

mock

`mock` 是一个测试库，提供模拟对象供测试用例使用。

Python 3 以后，

已将 `mock` 已经加入标准库，调用方法是 `from unittest import mock`。

Code Coverage

对于任何充分覆盖测试的代码，其 Code Coverage 程度肯定是 100%，任何覆盖率没能达到 100%

的代码都有隐藏 bug 的可能。在 Python 社区，代码覆盖计算工具的标准是 `coverage.py`，当然，在计算覆盖率时要记得配合 `tox`，以保证你针对不同环境的代码都被运行过。

`coverage.py` 的工作流程请参阅：[How coverage.py works](#)；
详细文档请参阅：[Documentation](#)。

`coverage.py` 也有 `nose` 插件，可以配合使用。

CI

- [travis CI](#) 对于开源项目免费
- [GitLab CI](#) 免费，提供本地部署支持

如何提高测试速度

如果测试很耗费时间，很容易引起开发人员的不满，因而怠于编写测试，所以说提高测试速度对于落实测试来说十分重要。

总结了一些提升测试效率的方法：

1. 合理使用 `setUpClass` 和 `tearDownClass` 方法。作为类方法，在拥有多个测试方法时也只会在一个测试用例中执行一次。
2. 数据库很慢，避免使用数据库。一定需要的话，请使用内存数据库（比如 `SQLite`）。
3. 使用 `mock`，避免使用 `model`。
4. 如果测试写起来很困难，说明需要重构了。
5. Celery 可以使用如下配置：

```
1. CELERY_ALWAYS_EAGER = True
```

```
2. CELERY_EAGER_PROPAGATES_EXCEPTIONS = True
3. BROKER_BACKEND = 'memory'
```

6. `django.test.utils.override_settings`
7. 关闭调试和日志
8. 删除不必要的中间件和app

这部分建议对于 Python 项目基本上也适用。

参考链接

- [Introduction to Python/Django tests](#)
- [DjangoCon 2013 - How to Write Fast and Efficient Unit Tests in Django](#)
- [Testing and Django](#)

如何测试 Django 应用

- [如何测试 Django 应用？](#)
 - [doctest](#)
 - [TestCase](#)
 - [Fixture](#)
 - [Client](#)
 - [testserver](#)
 - [Selenium](#)

如何测试 Django 应用？

Django 的启动互相之间的依赖严重，很多参数和依赖都需要在运行的时候导入，导致大部分文件都不能单独执行。

不过 Django 的社区非常活跃，对于知名的测试框架都有进行封装，如：

`django.test`、`django_nose` 等等，
以配合自身的测试命令使用。

doctest

在 Flask 中测试一个文件的 doctest 只需要运行：`python filename.py`，然而这在 Django 中行不通。

在 Django 中依赖自身的 test 命令：`python manage.py test [app_name]`，其中 `app_name` 若为空

默认测试所有应用。在 1.6 及以后版本中，

需要首先在 `settings.py` 中指定 `TEST_RUNNER`：

```
1. INSTALLED_APPS = (  
2.     ...  
3.     'django_nose',  
4. )  
5. TEST_RUNNER = 'django_nose.NoseTestSuiteRunner'  
6. NOSE_ARGS = ['--with-doctest']
```

TestCase

Django 的 `TestCase` 类是 `unittest.TestCase` 的子类，使用起来非常相似。

Fixture

Fixture 是 unittest 提供读取测试数据的一种方式，在 Django 的 TestCase 中也可以直接使用，使用前需要导出数据：

```
python manage.py dumpdata --format=yaml --indent=4 > fixtures_dir/filename.yaml
```

支持的数据格式包括 YAML、JSON 等等，YAML 可读性较高，不过需要安装额外的依赖。

配合 testserver 命令启动：

```
1. python manage.py testserver fixtures_dir/filename.yaml
```

在测试用例中指定：

```
1. from django.test import TestCase
2. from django.contrib.auth import authenticate
3.
4. class LoginTest(TestCase):
5.     fixtures = ['mysite.yaml']
6.
7.     def setUp(self):
8.         # 导入 fixture 中用户数据，省去创建用户的流程，也免去了清除用户数据的流程。
9.
10.    def test_has_user(self):
11.        # 如果已导入 fixture 中数据，则可以使用其中的账号登录。
12.        self.assertIsNotNone(authenticate(username='windrunner', password='password'))
```

Client

Client 提供了用户代理的模拟，其使用类似于 requests 库，不过使用前需要先初始化：

```
= Client() ,
```

Client 默认会提供 CSRF 认证，如果需要手动验证 CSRF，需要这样初始化：

```
csrf_enabled_client = Client(enforce_csrf_checks=True) 。
```

```
1. import unittest
2. from django.test.client import Client
3.
4. class PageTest(unittest.TestCase):
5.     def setUp(self):
6.         self.client = Client()
7.
8.     def test_home(self):
```

```

9.         res = self.client.get('/')
10.        self.assertEqual(200, res.status_code)
11.
12.    def test_login(self):
13.        """普通测试。client 实例会自动解决 csrf 问题。"""
14.        res = self.client.get('/login/')
15.
16.        self.assertEqual(200, res.status_code)
17.        self.assertIn('Username', res.content)
18.
19.        res_post = self.client.post('/login/', {'username': 'windrunner', 'password':
20.        'password', })
21.
22.        self.assertEqual(200, res_post.status_code)
23.        self.assertIn('windrunner', res_post.content)
24.
25.    def test_login_csrf(self):
26.        """强制 csrf 检查"""
27.        self.client = Client(enforce_csrf_checks=True)          # 使用检查 CSRF 的 Client 示例代替
28.        默认实例
29.        res = self.client.get('/login/')
30.        csrf_token = '%s' % res.context['csrf_token']           # 获取 csrf_token
31.
32.        res_fail = self.client.post('/login/', {'user': 'windrunner', 'pass': 'password', })
33.        self.assertEqual(403, res_fail.status_code)              # 没有处理 CSRF token 会返回 403 错
34.        误代码
35.
36.        res_csrf = self.client.post('/login/', {'user': 'windrunner', 'pass': 'password',
37.        'csrfmiddlewaretoken': csrf_token, })
38.        self.assertIn('windrunner', res_csrf.content)
39.
40.    def test_logout(self):
41.        res = self.client.post('/logout/')
42.        self.assertEqual(302, res.status_code)

```

testserver

testserver 是 Django 提供的启动测试服务器的方法，会创建一个测试数据库来替代默认数据库，
 通常会在启动时导入相应 fixture。命令如下：

```
1. python manage.py testserver --addrport 7000 fixture1 fixture2
```

Selenium

因为 Selenium 是控制浏览器测试 web 服务，因此并不会受到 Django 的干扰，这里有一段示例代码：

```

1. import unittest
2. from selenium import webdriver
3. from django.contrib.auth import get_user_model, authenticate
4.
5. class LoginTest(unittest.TestCase):
6.     def setUp(self):
7.         self.browser = webdriver.Firefox()          # 初始化浏览器，也可以选择 Chrome 或者 PhantomJS
8.
9.     def tearDown(self):
10.        self.browser.quit()                        # 测试结束后关闭浏览器
11.
12.    def _login(self):
13.        # 这个方法没有以 ``test`` 开始，因此并不会单独被执行。
14.        self.browser.get('http://localhost:8000/login')      # 发送 GET 请求并打开页面
15.
16.        # 使用浏览器的选择权选中 HTML 元素，并发送浏览器事件，复杂的元素选择可以借助 XPath
17.        self.browser.find_element_by_id('username').send_keys('windrunner')
18.        self.browser.find_element_by_id('password').send_keys('password')
19.        self.browser.find_element_by_id('submit').click()    # 触发点击事件
20.
21.    def test_login(self):
22.        self._login()
23.        self.assertIn('windrunner', self.browser.page_source) # 断言登录后的页面内容
24.
25.    def test_logout(self):
26.        self._login()
27.        self.assertIn('windrunner', self.browser.page_source)
28.        self.browser.get('http://localhost:8000/logout')
29.        self.assertIn('nobody', self.browser.page_source)
30.        self.assertNotIn('windrunner', self.browser.page_source)

```

面向接口测试——Python mock 库

- [mock](#)
 - [一些基本概念](#)
 - [为什么需要 mock](#)
 - [Mock](#)
 - [参数](#)
 - [属性](#)
 - [方法](#)
 - [MagicMock](#)
 - [patch](#)
 - [猴子补丁 \(monkey patch\)](#)
 - [例子](#)
 - [参考文章](#)

mock

mock 测试就是在测试过程中，对于某些不容易构建或获取的对象，用虚拟的对象来代替以便于测试的测试方法。

一些基本概念

double 可以理解为置换，它是所有模拟测试对象的统称，我们也可以称它为替身。一般来说，当你创建任意一种测试置换对象时，它将被用来替代某个指定类的对象。

stub 可以理解为测试桩，它能实现当特定的方法被调用时，返回一个指定的模拟值。如果你的测试用例需要一个伴生对象来提供一些数据，可以使用 stub 来取代数据源，在测试设置时可以指定返回每次一致的模拟数据。

spy 可以理解为侦查，它负责汇报情况，持续追踪什么方法被调用了，以及调用过程中传递了哪些参数。

你能用它来实现测试断言，比如一个特定的方法是否被调用或者是否使用正确的参数调用。当你需要测试两个对象间的某些协议或者关系时会非常有用。

mock 与 spy 类似，但在使用上有些许不同。spy 追踪所有的方法调用，并在事后让你写断言，而 mock 通常需要你事先设定期望。你告诉它你期望发生什么，然后执行测试代码并验证最后的结果与事先定义的期望是否一致。

fake 是一个具备完整功能实现和行为的对象，行为上来说它和这个类型的真实对象上一样，但不同于它所模拟的类，它使测试变得更加容易。一个典型的例子是使用内存中的数据库来生成一个

数据持久化对象，而不是去访问一个真正的生产环境的数据库。

实践中，这些术语常常用起来不同于它们的定义，甚至可以互换，因此不必太过于陷入这些词汇的细节。

这些定义更多的是为了在高层次上区分这些概念， 它也对考虑不同类型测试对象的行为会有帮助。

为什么需要 mock

对于为什么需要 mock，或者什么时候需要使用 mock，Tim Mackinnon 提出了一些建议：

- 真实对象具有不可确定的行为（产生不可预测的结果，如股票行情）
- 真实对象很难被创建
- 真实对象的某些行为很难触发（比如网络错误）
- 真实情况令程序的运行速度很慢
- 真实对象有用户界面
- 测试需要询问真实对象它是如何被调用的（比如测试可能需要验证某个回调函数是否被调用了）
- 真实对象实际上并不存在（当需要和其他开发小组，或者新的硬件系统打交道时）

Mock

`Mock` 类是 `CallableMixin` 和 `NonCallableMock` 的子类，实际上 `Mock` 中并没有其它定义：

```
class Mock(CallableMixin, NonCallableMock):pass
```

参数

- `spec`：这个参数是用来指定 `Mock` 实例的行为，那些方法是存在的，那些不存在。其值可以是一个类或实例，也可以是一个字符串列表。
- `spec_set`：比起 `spec` 更加严格。
- `side_effect`：在 `Mock` 实例被调用时被调用的方法，对应 `side_effect` 属性，可以用来返回动态值或者异常。其参数和 `mock` 相同，如果返回值不为 `DEFAULT` 则用作 `Mock` 实例的返回值。

如果 `side_effect` 是一个迭代器，则每次调用的时候返回其中下一个元素。如果迭代器中的元素是异常则将其抛出而非返回。

- `return_value`：mock 对象被调用时的返回值，默认是一个新的 `Mock` 实例。
- `wraps`：Item for the mock object to wrap. If `wraps` is not None then

calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an `AttributeError`).

If the mock has an explicit `return_value` set then calls are not passed to the wrapped object and the `return_value` is returned instead. 如果 Mock 实例存在 `return_value` , 不会调用被封装的对象。

- `name` : Mock 对象在 repr 时的名字, 调试时会很有帮助。该参数会传递给子 mock。

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created.

属性

- `call_args`
- `call_args_list`
- `call_count`
- `called`
- `return_value`
- `side_effect`

方法

- `attach_mock` Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the `method_calls` and `mock_calls` attributes of this one.
 - Set attributes on the mock through keyword arguments. Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```
1. >>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
2. >>> mock.configure_mock(**attrs)
```

- `mock_add_spec` Add a spec to a mock. `spec` can either be an object or a list of strings. Only attributes on the `spec` can be fetched as attributes from the mock. If `spec_set` is True then only attributes on the spec can be set.

- `reset_mock`

MogicMock

MagicMock 是 Mock 的子类，和 Mock 的不同之处在于 MagicMock 默认已经 mock 了对象的魔术方法（magic method）。

推荐使用 MagicMock。

patch

```
patch(target, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None,
**kwargs)
```

有两种主要用法：

装饰器：

```
1. @patch.object(SomeClass, 'attribute', sentinel.attribute)
2. def test():
3.     assert SomeClass.attribute == sentinel.attribute
```

和上下文管理器：

```
1. with patch('__builtin__.open', mock):
2.     handle = open('filename', 'r')
```

两种情况下都保证仅仅在函数体内或者 with 表达式中，`target` 被 `new` 所替代。当函数执行完或者离开 with 环境时，`target` 恢复回去。

`new` 默认是一个 `MagicMock` 对象

If `patch` is used as a decorator and `new` is omitted, the created mock is passed in as an extra argument to the decorated function. If `patch` is used as a context manager the created mock is returned by the context manager.

`target` should be a string in the form `'package.module.ClassName'`. The `target` is imported and the specified object replaced with the `new` object, so the `target` must be importable from the environment you are calling `patch` from. The target is imported when the decorated function is executed, not at decoration time.

The `spec` and `spec_set` keyword arguments are passed to the `MagicMock` if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the spec/spec_set object.

`new_callable` allows you to specify a different class, or callable object, that will be called to create the `new` object. By default `MagicMock` is used.

A more powerful form of `spec` is `autospec`. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a `TypeError` if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default `patch` will fail to replace attributes that don't exist. If you pass in `create=True`, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. `patch` finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is `test`, which matches the way `unittest` finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the `with` statement. Here the patching applies to the indented block after the `with` statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if `patch` is creating a mock object for you.

`patch` takes arbitrary keyword arguments. These will be passed to the `Mock` (or `new_callable`) on construction.

`patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

猴子补丁 (monkey patch)

`patch` 本质上是一个函数，但是在实现的时候通过 `MP` 添加了很多属性：

```
1. patch.object = _patch_object
2. patch.dict = _patch_dict
3. patch.multiple = _patch_multiple
4. patch.stopall = _patch_stopall
5. patch.TEST_PREFIX = 'test'
```

例子

[IPython notebook 在线示例](#)

参考文章

- [使用模拟对象 \(Mock Object\) 技术进行测试驱动开发](#)
- [置换测试: Mock, Stub 和其他](#)
- [PyPI - mock](#)
- [mock - getting started](#)
- [Terminology for Mocks/Doubles/Fake/Dummy/Stub](#)

REPL 环境

- [Python 的 REPL 环境](#)

Python 的 REPL 环境

安装完 Python 之后都会提供一个 REPL—Read-Eval-Print Loop，不过 Python 默认的 REPL 环境实在是简陋，

很多开发者就开发了更加强大的 REPL 工具，包括控制台和基于 web 的，其中比较有名的有：

- IPython (IPython Notebook)
- BPython
- ptPython

- [BPython](#)
- [IPython](#)
- [设置 Django Shell](#)

BPython

- BPython
 - 行内的语法高亮
 - 自动提示
 - 函数参数提示
 - 时光倒流 (Rewind)
 - 历史输出
 - 支持 Python 3
 - 自定义配置

BPython

相比 IPython, BPython 并不算得上功能强大, 它的开发初衷仅在于部分增强原始 REPL 的交互性,

并没有向 IDE 发展的意思, 以保持轻量、简单为目的。

根据[官网](#)介绍, 它有这些特性:

行内的语法高亮

自动提示

函数参数提示

时光倒流 (Rewind)

这里需要注意一下, BPython 的 `rewind` 并不是撤销, 而是将之前执行过的命令在全新的环境中重新执行到上一步。

历史输出

退出 BPython 时会将屏幕中所有的信息输出到 `stdout`, 也就是说你可以在 `shell` 中看到之前看过的一切。

支持 Python 3

自定义配置

```
1. bpython --config=/path/to/a/bpython/config/file
```

IPython

- IPython
 - 启动 & 退出
 - 特性
 - Tab 补全
 - 魔法指令 (Magic Command)
 - Plot
 - 内省
 - 函数内省
 - 输入历史
 - 编辑
 - 交互调试
 - 快捷键及配置
 - 作为库使用
 - 问题和方法
 - 粘贴代码缩进问题
 - Vim 模式
 - Mac 下的快捷启动
 - IPython notebook
 - 启动

IPython

IPython 是一个增强了交互能力的 Python REPL 环境，IPython v1-3 版本安装时会安装 IPython notebook，相当于一个网页版的 REPL 环境。

相对于原生的 Python REPL，IPython 主要提供了 Tab 自动补全、shell 命令交互、快速文档查询等功能。

除了在 shell 中使用外，Django shell 以及 PyCharm IDE 都提供了 IPython 的支持——安装了 IPython 后会选择其作为默认 python shell。

启动 & 退出

启动 IPython: `ipython`

启动 IPython notebook: `ipython notebook`

启动 IPython Qt shell: `ipython qtshell`

退出: 双击 Ctrl-D 或者 `exit` 或者 `quit` 或者 `exit()`。

特性

Tab 补全

原生的 Python shell 可以通过 `rlcompleter` 和 `readline` 库获得 Tab 补全能力：

```
1. import rlcompleter, readline
2. readline.parse_and_bind('tab: complete')
```

IPython 的支持更强大一些。不过 IPython 的自动补全依旧是循环式的，需要菜单式的可以试试 `ptpython`。

魔法指令 (Magic Command)

魔法指令是 IPython 定义的命令，以提供 IPython 环境下与环境交互的能力，通常对应于 shell 命令。

你可以在 IPython 控制台中输入 `ls` 或者 `%ls` 来查看当前目录文件。

魔法指令通常以 `%` 开始，对于非 `%` 开始的魔法指令，IPython 会首先判断它是不是 Python 对象，如是对象则优先作为 Python 对象处理。

更全面的 magic 命令列表和功能介绍见[这里](#)。

魔法指令中还包括了很多 Shell 命令，但并不是全部，IPython 也支持直接执行 Shell 命令，只需要以感叹号 `!` 起始，例如：`!ls`。

Plot

常见用法是配合 `matplotlib` 等 plot 库和 IPython 插件，在终端、Qt console 或者 Notebook 中展示图表。

内省

函数内省

IPython 提供了 `?` 和 `??` 命令用于内省，`?` 会显示函数简介，`??` 会连源代码一起显示（如果可用的话）。

输入历史

`hist` 魔法指令可以用来输出输入历史：

```
1. In [7]: hist
2. 1: a = 1
3. 2: b = 2
4. 3: c = 3
5. 4: d = {}
6. 5: e = []
7. 6: for i in range(20):
8.     e.append(i)
9.     d[i] = b
```

要去掉历史记录中的序号，使用命令 `hist -n`：

```
1. In [8]: hist -n
2. a = 1
3. b = 2
4. c = 3
5. d = {}
6. e = []
7. for i in range(20):
8.     e.append(i)
9.     d[i] = b
```

这样你就可以方便的将代码复制到一个文本编辑器中。

要在历史记录中搜索可以先输入一个匹配模型，然后按 `Ctrl-p`，找到一个匹配后，继续按 `Ctrl-p` 会向后搜索再上一个匹配，`Ctrl-n` 则是向前搜索最近的匹配。

编辑

当在 Python 提示符下试验一个想法时，经常需要通过编辑器修改源代码（甚至是反复修改）。

在 IPython 下输入魔法指令 `edit` 就会根据环境变量 `$EDITOR` 调用相应的编辑器。

如果 `$EDITOR` 为空，则会调用 `vi` (Unix) 或记事本 (Windows)。

要回到 IPython 提示符，直接退出编辑器即可。

如果是保存并退出编辑器，输入编辑器的代码会在当前名字空间下被自动执行。

如果你不想这样，使用 `edit -x`。如果要再次编辑上次最后编辑的代码，使用 `edit -p`。

在上一个特性里，我提到使用 `hist -n` 可以很容易的将代码拷贝到编辑器。

一个更简单的方法是 `edit` 加 Python 列表的切片 (slice) 语法。假定 `hist` 输出如下：

```
1. In [29]: hist -n
2. a = 1
3. b = 2
```

```
4. c = 3
5. d = {}
6. e = []
7.
8. for i in range(20):
9.     e.append(i)
10.    d[i] = b
```

现在要将第 4、5、6 句代码导出到编辑器，只要输入：

```
1. edit 4:7
```

交互调试

调试方面常用的魔术命令有 `ru`、`prun`、`time`、`timeit` 等等。更直接的方式是借助系统调试工具 `pdb`。

快捷键及配置

作为库使用

NotFinished，虽然 `ipython -p pysh` 提供了一个强大的shell替代品，但它缺少正确的job控制。在运行某个很耗时的任务时按下Ctrl-z将会停止IPython session而不是那个子进程。

问题和方法

虽然作为标准 Python Shell 的替代，IPython 总的来说很完美，但仍然存在一些问题。

粘贴代码缩进问题

默认情况下，IPython 会对粘贴的已排好缩进的代码重新缩进。例如下面的代码：

```
1. for i in range(10):
2.     for j in range(10):
3.         for k in range(10):
4.             pass
```

会变成：


```

1. for i in range(10):
2.     for j in range(10):
3.         for k in range(10):
4.             pass

```

这是因为 `autoindent` 默认是启用状态，可以用 `magic` 命令 `autoindent` 来开关自动缩进，就像在 Vim 中设置 `set paste` 一样。

或者，你也可以使用魔法指令 `%paste` 粘贴并执行剪贴板中的代码。

Vim 模式

Mac 下的快捷启动

IPython wiki 提供了一段 [AppleScript launcher](#) 脚本，把它写入 `~/bin/ipy` 在 Mac Spotlight 中输入 `ipy` 并回车就可以在新 iTerm 窗口中打开 IPython Shell。下面是我稍微修改后的版本：

```

1. #!/usr/bin/osascript
2. -- Run ipython on a new iTerm
3. -- See http://www.iterm2.com/#/section/documentation/scripting
4.
5. tell application "iTerm"
6.     activate
7.     set ipyterm to (make new terminal)
8.     tell ipyterm
9.         set ipysession to (make new session)
10.        tell ipysession
11.            set name to "IPython"
12.            exec command "${which ipython}"
13.        end tell
14.    end tell
15. end tell


```


IPython notebook

启动

在终端输入：

```
ipython notebook
```

IPython notebook 的使用和 Vim 有些类似，分“命令模式”和“编辑模式”，切换方法同样为  键。

在命令模式下按  可以呼出帮助菜单，貌似不区分大小写。

设置 Django Shell

- [Django Shell 增强](#)
 - [Django Extensions](#)
 - [shell_plus](#)
 - [设置](#)
 - [IPython Notebook](#)

Django Shell 增强

由于 Django 系统的特殊性，很难在 Shell 中直接导入 Django 应用，而只能使用它自己提供的 shell 命令。不过，并不是说就没有办法使用增强的第三方 Shell 了，[Django-Extensions](#) 插件提供了切换默认 Shell 的能力。

Django Extensions

[Django-Extensions](#) 是一个 Django 第三方插件集，其中囊括了很多实用的 Django 插件，当然也包括本文的主角 `shell_plus`。

shell_plus

装上插件后只需要将 `django_extensions` 加入 `INSTALLED_APPS` 列表中即可使用 IPython 作为默认 Shell（如果已安装的话）。

需要注意的是，如果在 virtualenv 环境中使用 Django 的话，同样需要在 virtualenv 中安装 IPython。

如果你偏好的是 BPython 的话，[shell_plus](#) 同样提供了相关命令：`shell_plus`。使用方法：

```
1. ./manage.py shell_plus --bpython
```

该命令对 IPython 同样适用：

```
1. ./manage.py shell_plus --ipython
```

或者，有需要的话，也可以切换回原生 Python：

```
1. ./manage.py shell_plus --plain
```

默认 Shell 的选择顺序是 BPython、IPython、Python。

你同样可以在文件中配置默认 Shell：

```
1. SHELL_PLUS = "ipython"
```

设置

默认情况下 shell_plus 会自动加载你的全部 Model，其中可能会出现重名或者其它问题，你可以通过配置文件来修改它的加载机制。

注意：这里的设置只会作用于 shell_plus 而不会影响你的整改系统。

将名为 blog 的 app 中的 Messages 模块以 blog_messages 名加载

```
1. SHELL_PLUS_MODEL_ALIASES = {'blog': {'Messages': 'blog_messages'},}
```

不加载 app 'sites' 中的全部 model，以及 app 'blog' 中的 'pictures' Model。

```
1. SHELL_PLUS_DONT_LOAD = ['sites', 'blog.pictures']
```

`SHELL_PLUS_MODEL_ALIASES` 和 `SHELL_PLUS_DONT_LOAD` 可以同时使用。

也可以像这样在运行 manage.py 时再设定不自动加载的模块：

```
1. ./manage.py shell_plus --dont-load app1 --dont-load app2.module1
```

命令行和配置文件中的设置会合并后再生效。

IPython Notebook

shell_plus 同时还支持 [IPython Notebook]：

```
1. ./manage.py shell_plus --notebook
```

Notebook 相关的设置参数有两个：`NOTEBOOK_ARGUMENTS` 和 `IPYTHON_ARGUMENTS`：

`NOTEBOOK_ARGUMENTS` 用于配置 IPython Notebook：

```
1. ipython notebook -h
```

```

1. NOTEBOOK_ARGUMENTS = [
2.     '--ip=x.x.x.x',
3.     '--port=xx',
4. ]

```

`IPYTHON_ARGUMENTS` 用于设置 IPython:

```

1. ipython -h

```

使用 IPython Notebook 时同样会自动加载 Django settings 模块和数据库 Model。其实现依赖于定制的 IPython 插件，Django Extensions 默认会向 IPython Notebook 发送命令

`--ext django_extensions.management.notebook_extension` 开启该功能，

你可以在 Django 设置中修改 `IPYTHON_ARGUMENTS` 参数覆盖该设置：

```

1. IPYTHON_ARGUMENTS = [
2.     '--ext', 'django_extensions.management.notebook_extension',
3.     '--ext', 'myproject.notebook_extension',
4.     '--debug',
5. ]

```

开启自动加载可以通过引用 Django Extensions 默认 Notebook，或者复制它的自动加载代码到你自己的扩展中。

IPython Notebook 扩展目前并不支持 `--dont-load` 选项。

科学计算

- [科学计算](#)
 - [常用库介绍](#)
 - [IPython 和 Jupyter Notebook](#)
 - [NumPy](#)
 - [Pandas](#)
 - [Matplotlib](#)
 - [Scipy](#)
 - [参考链接](#)

科学计算

Python 在科学计算上的应用非常广泛，包括数学、统计学、图形学.....等等，也是科学计算领域的首选编程语言之一。

这一部分的文章主要是介绍 Python 在科学计算领域常用的库，以及科学计算在日常中可能的实际用例。

常用库介绍

IPython 和 Jupyter Notebook

NumPy

[NumPy](#) 是 Python 科学计算生态系统的基础，提供了多维数组操作、线性代数运算、傅立叶变换等多种数学操作，并且使用 C 编写了其中性能瓶颈部分，大大降低科学计算的使用难度。

Pandas

[Pandas](#) 侧重于结构化数据的分析处理，也是 Python 在数据分析领域的杀手应用。

Pandas 本是面向金融数据分析领域的工具，因此出去对一般数据处理功能外，对于金融数据分析有更好的支持。

Matplotlib

[Matplotlib](#) 是最流行的 Python 图表绘制库，在使用各种工具对数据处理后，通常会选用 Matplotlib

作为最终展示工具。如果你只是想绘制图表，完全只需要学会使用这一个库。

Matplotlib 和 IPython（Notebook）之间的结合非常完善。

Scipy

[Scipy](#) 包含了众多科学计算标准问题域的包，主要包括数值积分例程和微分方程求解器 `scipy.integrate`、`numpy.linalg` 的扩展 `scipy.linalg`、信号处理工具 `scipy.signal`、稀疏矩阵和稀疏线性系统求解器 `scipy.sparse`、概率分布、统计相关的 `scipy.stats` 等等。

参考链接

《[Python 网页爬虫](#) & [文本处理](#) & [科学计算](#) & [机器学习](#) & [数据挖掘兵器谱](#)》

网络应用

- [HTTP](#)
- [分布式系统](#)

HTTP

- [HTTP](#)

HTTP

分布式系统

- [分布式系统](#)

分布式系统

网站应用

- [Context](#)
- [浅谈 web 框架](#)
- [web 服务器](#)
- [WSGI 服务器](#)
- [服务器最佳实践](#)
- [部署代码](#)
- [HTML 模板](#)

Context

- [Context](#)

Context

浅谈 web 框架

- [浅谈 web 框架](#)
 - [Django](#)
 - [Flask](#)
 - [Tornado](#)

浅谈 web 框架

Python 语言简洁，开发者可以专注于逻辑的编写，因此重新造一遍轮子的代价非常之低，这恐怕是 Python 社区之所以会出现如此众多 web 框架的主要原因。

不过硬要评比最流行的 web 框架，当然还是要数 Django、Flask 和 Tornado，三者在 GitHub 上的 star 数要远远超过其它竞争对手。而且三者都特点鲜明，深受不同风格的开发者拥趸：Django 大而全，适合快速开发；Flask 小而美，专注于自己慢慢选配插件；Tornado 性能好，支持异步、高并发。针对不同的需求，你应该最适合自己 and 项目的 web 框架。

除了以上三者，其它知名框架还有很多：Twisted、web.py、web2py、bottle、Pylons。。。飞龙非龙的博客上有一篇[比较简单的比较](#)，可以帮助你认识他们之间设计理念的区别。

个人经验上来看，Flask、Bottle、web.py 这样简单的 web 框架更适合初学者学习，比较适合一步步进阶。而 Django 在快速开发 demo 上比较有优势（前提是你熟悉它们），Tornado 适合开发 API 和聊天室这样高并发的应用，但要注意不要混用同步和异步的库，否则会失去它自身的优势。

Django

说到 Python web 框架，绝对绕不过去的就是 Django 了，类似于 Ruby 社区的 RoR，两者都擅长 10 分钟搭建博客，喜欢自己提供一套轮子。即使如此，仍然免不了使用第三方库，两者开发进度上的差异往往会带来痛苦的兼容性问题。顺道一提，Django 的长期支持版本间隔为 0.4，也就是说 Django 1.x 中的长期支持版本包括 1.0，1.4，1.8。

Django 的快速开发性质使得总有人将它和 WordPress 相提并论，不过 Django 的抽象度明显更高，是 web 框架基本的。Django 社区也有类似于 WordPress 的产品，而且并不少，因为 Django 本身就是做 CMS 系统出身的。

Flask

Flask 像是 Django 的精简版本，设计上也有很多和 Django 类似的地方，其第三方库充斥着

Django 功能的仿制品，因此使用起来更像是定制版的 Django。

Flask 官方自称是“微框架”，这一点我不太认同，这种说法有误导嫌疑。虽然 Flask 的代码很精简，

但是它极度依赖于 Werkzeug 中间件和 Jinja2 模版库，并不是所有功能都是自身精简的代码所提供，

所以我并不承认它和 Bottle、Pylons 一样能被称为微框架。

Tornado

Tornado 和上面两者的差异就比较大了，它的特点在于轻量、异步、高性能。轻量是因为它的代码很精简，

主要只提供了网络方面功能的封装（原先还自带了一个异步 MySQL 连接库，后来也剥离出去了）。

高性能主要来自于它的异步实现，相对于同步框架，异步非阻塞能够处理更多的并发连接。

异步自然是 Tornado 最大的特性了。

Tornado 的异步实现称为 epoll，Linux 下依赖 libev，BSD 和 Mac 下可以用 kqueue 代替，

在 Windows 下并没有对应的实现。是的，这里说到了一些系统底层的实现，这是因为 WSGI 规范并没有提供异步特性的支持，因此为了使用异步特性，Tornado 框架必须运行在支持异步的 web 服务器应用上，Tornado 自带的 server 就是一个很好的选择，其它替代还有 Twisted server。

web 服务器

- [web 服务器](#)

web 服务器

WSGI 服务器

- [WSGI 服务器](#)

WSGI 服务器

服务器最佳实践

- [服务器最佳实践](#)

服务器最佳实践

部署代码

- [部署代码](#)

部署代码

HTML 模板

- [HTML 模板](#)

HTML 模板

网站的部署

- [部署](#)
 - [web 应用服务器](#)
 - [uWSGI](#)
 - [Gunicorn](#)
 - [Tornado](#)
 - [Nginx/Apache](#)
 - [服务器部署](#)
 - [Fabric](#)
 - [Ansible](#)

部署

根据和代码/系统的紧密程度，我们可以简单地把部署分为应用级和系统级。应用级通常提供把 HTTP 请求递交给应用，而系统级和运维的关系更为紧密。

下面是一个非常常见的 Python 应用部署架构：

- [nginx](#)：静态文件服务；SSL 负载转移；反向代理；
- [Memcached](#) / [Redis](#)：缓存；
- [Celery](#)：运行后台任务；
- [Redis](#) / [RabbitMQ](#)：任务队列（通常对接 Celery）；
- [[uWSGI](#)] / [[Gunicorn](#)]：WSGI 服务器；

web 应用服务器

uWSGI

[uWSGI](#) 是一个主要以 C 语言实现的高性能服务器，性能优异、配置灵活，可以使用 C、C++ 甚至 Objective-C 编写插件。

uWSGI 并不仅仅是一个 WSGI 服务器，它也可以用作 Ruby Rack 应用或者 Perl PSGI 应用的后端服务器。

Gunicorn

[Gunicorn](#) 是一个类 Unix 系统上一个 Python 实现的 WSGI 服务器，性能很高使用也很简单。

作为 WSGI 服务器，Gunicorn 拥有很好的性能，但是它并不擅长静态文件处理，因此在实际部署中

常常隐藏于

Nginx 之后，由 Nginx 提供静态文件服务，动态请求则通过反向代理发送给 Gunicorn。

Tornado

Nginx/Apache

服务器部署

Python 应用部署的常用工具有 [Fabric](#)、[SaltStack](#)、[Ansible](#) 和 [Puppet](#)，前三者都是 Python

应用，Puppet 这是 Ruby 便携的服务器部署工具，除 Fabric 是完全开源外，后三者背后都有商业公司，

提供应用的商业版本或者应用之外的企业服务。

Fabric

[Fabric](#) 是一个基于 SSH 的命令式部署工具，通过编写 `fabfile` 来扩展和定制 Fabric 的功能。不得不说，对于复杂的部署环境来说，Fabric 的部署方式已经有些落后，但是对于少量、简单的服务环境，Fabric 使用起来简单、方便。

Ansible

[Ansible](#) 同样是基于 SSH 的服务器运维工具。

uWSGI

- [uWSGI](#)
 - [安装指南](#)
 - [简单使用](#)

uWSGI

虽然 [uWSGI](#) 最常见的使用场景是当作 WSGI 服务器，但它其实还提供了更多的其它功能，[这篇博客](#)

中介绍了 uWSGI 如同瑞士军刀般强大的各项功能。

安装指南

官方提供了多种安装方式，比如使用 pip 安装 uWSGI 安装包：`pip install uwsgi`。

或者使用安装脚本：

```
1. curl http://uwsgi.it/install | bash -s default /tmp/uwsgi
```

或者编译安装最新版本：

```
1. wget http://projects.unbit.it/downloads/uwsgi-latest.tar.gz
2. tar zxvf uwsgi-latest.tar.gz
3. cd <dir>
4. make
```

简单使用

以一个简单的 uwsgi 命令为例介绍它的常见用法：

```
1. uwsgi --http :9090 --wsgi-file foobar.py --master --processes 4 --threads 2 --stats
    127.0.0.1:9191
```

- `--http :9090` 是指监听端口 9090 的 HTTP 请求。
- `--wsgi-file foobar.py` 是指从文件 `foobar.py` 中寻找 WSGI 应用对象，默认寻找 `application` 对象。
- `--processes 4 --threads 2` 指 uWSGI 启动 4 个进程，每个进程启动 2 个线程。
- `--stats 127.0.0.1:9191` 提供了 uWSGI 状态监视端口，可以安装 `uwsgitop` 来查看 uWSGI

进程状态。

Gunicorn

- [Gunicorn](#)
 - [安装指南](#)
 - [简单使用](#)
 - [常用参数](#)
 - [部署实例](#)
 - [配置](#)

Gunicorn

[Gunicorn](#) 全称 Green Unicorn，是 Ruby 服务器 Unicorn 的 Python 版本，虽然完全是由 Python

实现，但是拥有作为 WSGI 服务器性能优秀，是现在最流行的 WSGI 服务之一。

安装指南

虽然在很多 Linux 操作系统上都有 Gunicorn 的安装包，但通常还是建议使用 Python 包管理工具 `pip` 来安装最新版本的 Gunicorn：

```
1. [sudo ]easy_install pip
2. pip install gunicorn
```

简单使用

Gunicorn 的使用非常简单。

假设你的 WSGI 应用包叫做 `myapp`，包中有一个符合 WSGI Application 规范的对象，就可以这样部署启动 Gunicorn 服务器来托管应用：

```
1. gunicorn -w 4 myapp:app
```

上面命令的 `-w 4` 是指启动 4 个 worker 进程，用以提高服务器的负载能力。

常用参数

- `-c CONFIG`, `--config=CONFIG` - 指定配置文件，格式可以是 `$(PATH)`、`file:$(PATH)` 或者 `python:$(MODULE_NAME)`。

- `-b BIND, --bind=BIND` - 指定 socket 地址, 可以是 `$(HOST)`、`$(HOST):$(PORT)` 或者 `unix:$(PATH)` 格式。`$(HOST)` 可以是单纯的 IP 地址。
- `-w WORKERS, --workers=WORKERS` - worker 进程数。推荐数量为服务器 CPU 核数的 2-4 倍。
- `-k WORKERCLASS, --worker-class=WORKERCLASS` - worker 进型的类型, 详细介绍请查看产品文档。你可以在这里设置 `$(NAME)`, 其值可以是 `sync`、`eventlet`、`gevent` 或者 `tornado`、`gthread`、`gaiohttp`。默认是 `sync`。
- `-n APP_NAME, --name=APP_NAME` - 如果安装了 `setproctitle`, 你可以使用该参数设置 Gunicorn 的进程名。

部署实例

Gunicorn 虽然是一个性能优秀的 WSGI 服务器, 但是并不擅长静态资源管理, 在生产环境下通常会配合

Nginx 使用。由 Nginx 作为服务器前端, 将动态请求通过反向代理传递给 Gunicorn。Nginx 反向代理配置示例:

```
1. server {
2.     listen 80;
3.     server_name example.org;
4.     access_log /var/log/nginx/example.log;
5.
6.     location / {
7.         proxy_pass http://127.0.0.1:8000;
8.         proxy_set_header Host $host;
9.         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
10.    }
11. }
```

配置

参考: <http://docs.gunicorn.org/en/latest/configure.html>

HTML 与 XML

- [HTML 与 XML](#)

HTML 与 XML

- [HTML](#)
- [lxml 与 requests](#)

HTML

- [HTML](#)

HTML

lxml 与 requests

- [lxml 与 requests](#)
 - [lxml](#)
 - [requests](#)

lxml 与 requests

lxml

requests

控制台程序

- [sh](#)
- [wget](#)
- [progressbar](#)
- [colorama](#)
- [Gooyey](#)—把 CLI 程序变成 GUI

sh

- [sh](#)

sh

[sh](#) 库提供了 Shell 命令的封装，如果你很熟悉 Shell 命令那么一定会喜欢上它。使用方法：

```
1. from sh import find
2. find("/tmp")
3. /tmp/foo
4. /tmp/foo/file1.json
5. /tmp/foo/file2.json
6. /tmp/foo/bar/file3.json
```

wget

- [wget](#)

wget

[wget](#) 是一个很强大的 Python 下载库，使用方法如下：

```
1. >>> import wget
2. >>> url = 'http://www.futurecrew.com/skaven/song_files/mp3/razorback.mp3'
3. >>> filename = wget.download(url)
4. 100% [.....] 3841532 / 3841532>
5. >> filename
6. 'razorback.mp3'
```

progressbar

- [progressbar](#)

progressbar

[progressbar](#)

提供了一个控制台进度条工具，示例代码：

```
1. from progressbar import ProgressBar
2. import time
3. pbar = ProgressBar(maxval=10)
4. for i in range(1, 11):
5.     pbar.update(i)
6.     time.sleep(1)
7. pbar.finish()
8. # 60% |#####
```


colorama

- [colorama](#)

colorama

提供彩色控制台输出：

```
>>> from colorama import Fore
>>> print Fore.RED + 'some red text'
some red text
```

Goocy——把 CLI 程序变成 GUI

- [Goocy——把 CLI 程序变成 GUI](#)
 - [简单的示例](#)
 - [控件](#)
 - [国际化](#)
 - [配置](#)
 - [布局流](#)

Goocy——把 CLI 程序变成 GUI

[Goocy](#) 是一个 Python GUI 程序开发框架，基于 [wxPython](#) GUI 库，其使用方法类似于 Python

内置 CLI 开发库 `argparse`，因此很容易把一个基于 `argparse` 的 CLI 应用转换成 GUI 程序。

简单的示例

我们首先从一个简单的基于 `argparse` 库的 CLI 应用开始：

```
1. from argparse import ArgumentParser
2.
3. def main():
4.     parser = ArgumentParser(description="My Cool GUI Program!")
5.     parser.add_argument('Filename')
6.     parser.add_argument('Date')
7.     parser.parse_args()
8.
9. if __name__ == '__main__':
10.     main()
```

这就有了一个接受 `Filename` 和 `Date` 的两个必填参数的 CLI 程序，为了简化程序我们并没有指定它的功能。

使用方法：`python cli.py FILENAME DATE`。

现在我们把它改成

```
1. from goocy import Goocy, GoocyParser
2.
3. @Goocy
```

```
4. def main():
5.     parser = GoocyParser(description="My Cool GUI Program!")
6.     parser.add_argument('Filename')
7.     parser.add_argument('Date')
8.     parser.parse_args()
9.
10. if __name__ == '__main__':
11.     main()
```

于是我们得到了一个简单的 GUI 对话框，并且基本上没有修过多少代码：



当然，仅仅这样是不够的，我们的 `Filename` 参数需要对应一个文件，而 `Date` 参数是一个日期，如果有专用的控件就好了！

Goocy 当然也想到了：



仅仅修改了 2 行代码：

```
1. parser.add_argument('Filename', widget="FileChooser")
2. parser.add_argument('Date', widget="DateChooser")
```

控件

上面已经看到了两个简单的控件：`FileChooser` 和 `DateChooser`，分别提供了一个“文件选择器”和“日期选择器”。现在支持的 `chooser` 类控件有：

控件名	控件类型
<code>FileChooser</code>	文件选择器
<code>MultiFileChooser</code>	文件多选器
<code>DirChooser</code>	目录选择器
<code>MultiDirChooser</code>	目录多选器
<code>FileSaver</code>	文件保存
<code>DateChooser</code>	日期选择
<code>TextField</code>	文本输入框
<code>Dropdown</code>	下拉列表

Counter	计数器
CheckBox	复选框
RadioGroup	单选框

国际化

Goocy 的国际化是通过配置实现的，使用方法如下：

```
1. @Goocy(language='russian')
2. def main():
3.     ...
```

目前仅支持 `ru`，`en`、`nl`、`fr`、`pt` 的支持已在计划中。

配置

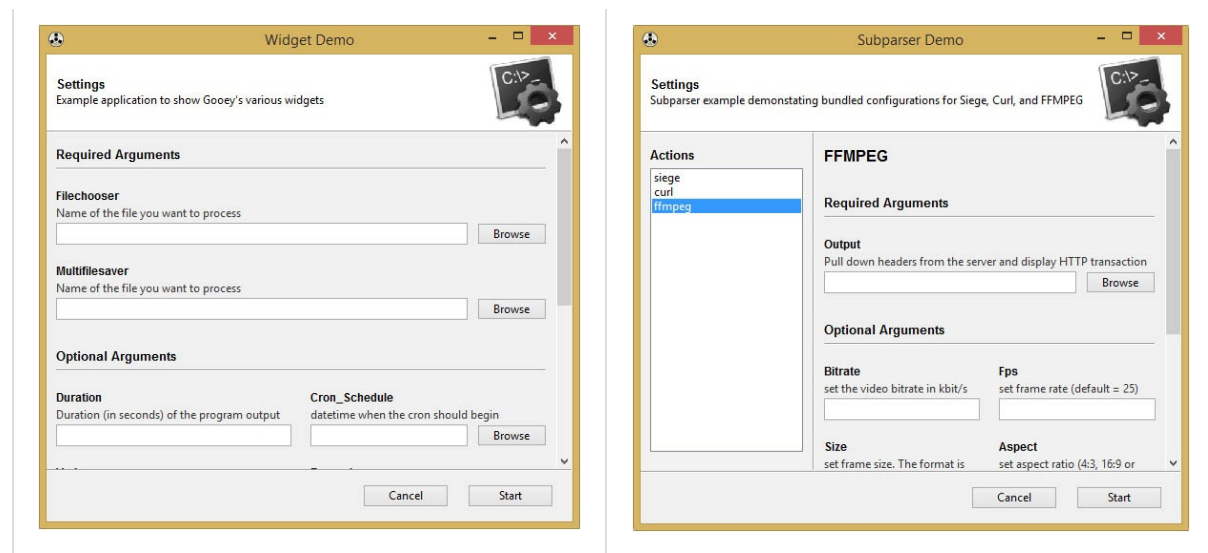
和 `language` 参数配置一样，Goocy 还支持很多其它配置，下面是它支持的参数列表：

参数	简介
<code>advanced</code>	切换显示全部设置还是仅仅是简化版本
<code>show_config</code>	Skips the configuration all together and runs the program immediately
<code>language</code>	指定从 <code>goocy/languages</code> 目录读取哪个语言包
<code>program_name</code>	GUI 窗口显示的程序名。默认会显 <code>sys.argv[0]</code> 。
<code>program_description</code>	<code>Settings</code> 窗口顶栏显示的描述性文字。默认值从 <code>ArgumentParser</code> 中获取。
<code>default_size</code>	窗口默认大小。
<code>required_cols</code>	设置必选参数行数。
<code>optional_cols</code>	设置可选参数行数。
<code>dump_build_config</code>	将设置以 JSON 格式保存在硬盘中以供编辑/重用。

布局流

布局实例可以在下载这个 [example](#) 库体验。

Flat Layout	Column Layout



数据库

- [DB API](#)
- [SQLAlchemy](#)
- [DjangoORM](#)
- [Pony ORM](#)
- [peewee](#)

DB API

- [DB API](#)

DB API

SQLAlchemy

- [SQLAlchemy](#)

SQLAlchemy

DjangoORM

- [DjangoORM](#)

DjangoORM

Pony ORM

- [Pony ORM](#)
 - [创建数据库](#)
 - [定义实体](#)
 - [添加与数据库表的映射](#)
 - [编写查询语句](#)
 - [获取对象](#)
 - [更新对象](#)
 - [数据库 Session](#)
 - [手动编写 SQL 语句](#)
 - [事务](#)
 - [实现](#)

Pony ORM

[Pony](#) 是一个很有意思的 ORM，它的特别之处在于可以使用 Python 生成器的语法来创建数据库请求，我们可以用这样的语句来查询数据库：

```
1.  
2. select(p for p in Product if p.name.startswith('A') and p.cost <= 1000)
```

[Pony](#) 还提供了一个在线的数据库结构编辑器：[Pony Editor](#)

演示示例：[University](#)

注意：[Pony](#) 采用多重许可证，对于开源项目采用 [GPL v3.0](#) 许可证，非商业使用免费，不同规模的商业使用会有不同的[收费政策](#)。

创建数据库

定义实体

添加与数据库表的映射

编写查询语句

获取对象

更新对象

数据库 Session

手动编写 SQL 语句

事务

实现

关于 Pony ORM 的实现，其作者在

[StackOverflow](#)

上有做过解释，这里附上中文：

[WindRunner](#)

参考链接：[官方文档](#)

peewee

- [peewee](#)

peewee

NoSQL 数据库

- [Redis](#)
- [MongoDB](#)
- [Neo4j](#)

Redis

- [Redis](#)

Redis

Redis 是一个键值数据库

MongoDB

- [MongoDB](#)

MongoDB

Neo4j

- [Neo4j](#)

Neo4j

网络基础

- [Twisted](#)
- [gevent](#)
- [PyZMQ](#)

Twisted

- [Twisted](#)

Twisted

gevent

- [gevent](#)

gevent

PyZMQ

- [PyZMQ](#)

PyZMQ

图片处理

- [PIL](#)
- [QRCode](#)
- [几种图片转字符算法介绍](#)
- [验证码破解](#)

PIL

- [PIL](#)
 - [实例联系](#)
 - [图片转 ASC II 码](#)
 - [图片相似度计算](#)
 - [图片相似度计算&索引](#)

PIL

实例联系

图片转 ASC II 码

原理:

<http://www.jave.de/image2ascii/algorithms.html>

示例代码: <https://github.com/kxxoling/image2ascii>

图片相似度计算

图片相似度计算&索引

QRCode

- [QRCode](#)
 - [Python QRCode 库](#)
 - [安装](#)
 - [使用 qrcode](#)
 - [qr 命令](#)
 - [Python API](#)

QRCode

QR码（全称为快速响应矩阵码；英语：Quick Response Code）是二维条码的一种，于1994年由日本DENSO WAVE公司发明。QR 码使用四种标准化编码模式（数字，字母数字，二进制和 Kanji）来存储数据。

Python QRCode 库

Python 可以安装 [qrcode](#) 库以获取 QR Code 生成的支持。

安装

qrcode 库依赖于 Python Image Library (PIL)，不过 PIL 已经停止更新，而且其设计并不符合 Python 规范。

因此推荐使用 PIL 的继任者 Pillow 代替。

安装 PIL 或者 Pillow 需要先安装 C 语言 PIL 库，各操作系统各有不同。

Python PIL 或者 Pillow 的安装可以通过 `pip` 或者 `easy_install`：

```
1. pip install pillow
```

或者

```
1. pip install PIL
```

安装 Python QRCode：

```
1. pip install qrcode
```

使用 qrcode

QRCode 库提供两种调用提供方式—Python 库和系统命令(qr)。

qr 命令

qr 命令的使用如下：

```
1. qr some_word[ > some_image.png]
```

qrcode 会根据文字的长度自动选择合适的 QRCode 版本

Python API

Python 下使用 qrcode 库：

```
1. import qrcode
2. qr = qrcode.QRCode(
3.     version=1,                                # QR Code version, 1-4
4.     error_correction=qrcode.constants.ERROR_CORRECT_L, # 错误纠正等级 L、M、Q、H 四等，默认是 M
5.     box_size=10,                                # QR Code 图片的大小，单位是像素
6.     border=4,                                    # QR Code 的边框，单位是像素，默认 4
7. )
8. qr.add_data('Some data')                       # 想要添加到 QR Code 中的内容
9. qr.make(fit=True)
10.
11. img = qr.make_image()
```

默认输出格式是 JPG，生成 SVG 需要设定 `image_factory` 参数：

```
1. import qrcode
2. import qrcode.image.svg
3.
4. if method == 'basic':
5.     # Simple factory, just a set of rects.
6.     factory = qrcode.image.svg.SvgImage
7. elif method == 'fragment':
8.     # Fragment factory (also just a set of rects)
9.     factory = qrcode.image.svg.SvgFragmentImage
10. else:
11.     # Combined path factory, fixes white space that may occur when zooming
12.     factory = qrcode.image.svg.SvgPathImage
13.
14. img = qrcode.make('Some data here', image_factory=factory)
```

如果需要 PNG 支持，还需要安装第三支持：


```
1. pip install git+git://github.com/ojii/pymaging.git#egg=pymaging
2. pip install git+git://github.com/ojii/pymaging-png.git#egg=pymaging-png
```

依旧是设定 `image_factory` 设置输出格式为 PNG:

```
1. import qrcode
2. from qrcode.image.pure import PymagingImage
3. img = qrcode.make('Some data here', image_factory=PymagingImage)
```

几种图片转字符算法介绍

- 图片转字符原理
 - 黑白算法
 - 灰度算法
 - 边际追踪 / 界定算法

图片转字符原理

图片转字符通常分以下几种：

1. 黑白算法
2. 灰度算法
3. 边际追踪 / 界定算法

黑白算法

黑白算法最简单

低保真度解法：

```
1. ##### ## #####
2. # # # #
3. # ## # #
4. # # # #
5. # ##### ## #
```

高保真度解法：

```
1. 88888 8888 d8b 88888
2. 8 8 ]b 8
3. 8 88 Yb 8
4. 8 8 o d8 8
5. 8 8888 YP 8
```

低保真度解法仅仅简单地通过字符 `#` 和空白符表示图形区域地颜色，高保真算法则使用 12 个字符对应一个 `2*2` 的区域，对照表如下：

Pattern:	<div>1.</div>	<div>1. .X . .</div>	<div>1. X. . .</div>	<div>1. . . .X.</div>	<div>1. . . .X</div>	<div>1</div>
Character:	<div>1. `</div>	<div>1. '</div>	<div>1. .</div>	<div>1. ,</div>	<div>1. "</div>	<div>1</div>

灰度算法

灰度算法的基本目标是追求转换后的每个字符亮度都尽可能接近该区域原始图像的亮度，因此在文字渐渐缩小或者与人眼的距离越来越远时，看起来越发接近图片。边缘、边框以及其它高对比度的地方应该使用更加契合其结构的字符。

灰度算法需要使用实现亮度一字符的转换，大多数具体实现都采用 1 个像素对 1 个字符的转换，“state of the art”算法使用 4 个像素对应一个字符，并且能够提供更清晰的结果。

边际追踪 / 界定算法

边际界定算法看似更加复杂，但其实不然！

全文主要翻译自 java.de



验证码破解

- [验证码的原理与破解](#)

验证码的原理与破解

[常见验证码的弱点与验证码识别](#)

开发环境

- [编辑器](#)
- [IDE](#)

编辑器

- [编辑器](#)

编辑器

IDE

- [IDE](#)

IDE

提高代码效率

- [Context](#)
- [Tools](#)
- [Libraries](#)
- [Resources](#)

Context

- [Context](#)

Context

Tools

- [Tools](#)

Tools

Libraries

- [Libraries](#)

Libraries

Resources

- [Resources](#)

Resources

设计模式

- [设计模式](#)

设计模式

- [单例模式](#)

单例模式

- 单例模式
 - 装饰器
 - metaclass

单例模式

单例模式，也叫单子模式，是一种常用的软件设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个全局对象，这样有利于我们协调系统整体的行为。

——以上来自[维基百科](#)

从定义上来看，这会是一个很有用的避免冲突的设计模式，相当于把所有同样资源的调用都交给了一个资源代理。那么 Python 中该如何实现这一模式呢？

装饰器

所有资源资源调用者都是同一个对象，我首先想到的就是装饰器，可以很方便的给不同的对象增添相同的功能。

[Python 官方 wiki](#)

给出了一个非常优雅的实现：

```
1. def singleton(cls):
2.     instance = cls()
3.     instance.__call__ = lambda: instance
4.     return instance
5.
6. # Sample use
7.
8. @singleton
9. class Highlander:
10.     x = 100
11.     # Of course you can have any attributes or methods you like.
12.
13.
14. highlander = Highlander()
15. another_highlander = Highlander()
16. assert id(highlander) == id(another_highlander)
```

上面的代码定义了一个 singleton 装饰器，覆盖了类的 `__call__` 方法，该方法会在类创建的时候创建一个类的实例，并在之后类每次的实例化时总是返回这个实例对象。

当然，如果你希望只在需要的时候创建类的实例对象也有别的方法：

```

1. def singleton(cls, *args, **kw):
2.     instances = {}
3.     def _singleton():
4.         if cls not in instances:
5.             instances[cls] = cls(*args, **kw)
6.         return instances[cls]
7.     return _singleton
8.
9. @singleton
10. class MyClass(object):
11.     a = 1
12.     def __init__(self, x=0):
13.         self.x = x
14.
15. one = MyClass()
16. two = MyClass()
17.
18. assert id(one) == id(two)

```

上面的代码中实现了这样一个装饰器：装饰器函数创建的时候会创建一个 `instances` 字典，该字典用于保存被装饰器修改过的类的实例，在类初始化的时候首先判断是否存在其实例，如果存在则直接返回，否则创建一个新的实例保存到 `instances` 字典中，并返回该实例。（这段代码出自 [cnblogs](#)）

metaclass

我自己对于不是很喜欢装饰器的实现，因为从语言逻辑上来看，我需要的是一个有单例特性的类而不是为类添加单例限制。于是我又找到了基于 `metaclass` 的实现：

```

1. class Singleton(type):
2.     def __init__(cls, name, bases, dict):
3.         super(Singleton, cls).__init__(name, bases, dict)
4.         cls._instance = None
5.     def __call__(cls, *args, **kw):
6.         if cls._instance is None:
7.             cls._instance = super(Singleton, cls).__call__(*args, **kw)
8.         return cls._instance
9.
10. class MyClass(object):
11.     __metaclass__ = Singleton
12.
13. one = MyClass()
14. two = MyClass()

```



```
15.  
16. assert id(one) == id(two)
```

上面的代码在类的第一次实例之后将这个实例作为其类变量保存，在之后调用类的构造函数的时候都直接返回这个实例对象。

这个解决方案强化了类与其单例之间的内聚性。

参考链接：[Creating a singleton in python](#)

文件处理

- [python-magic](#)

python-magic

- [python-magic](#)
 - [libmagic](#)

python-magic

libmagic

python-magic 是基于 libmagic 开发的文件识别库，功能强大、使用简单。

[GitHub](#)

Python Magic Methods

- [Python Magic 方法](#)

Python Magic 方法

链接<http://www.rafekettler.com/magicmethods.html>

- [Python 中的魔法方法](#)

Python 中的魔法方法

- [Python 中的魔法方法](#)
 - [操作符重载](#)

Python 中的魔法方法

Python 中的对象都有诸如 `__init__()` 这样的方法，它们都有各自特定的用途，却无法直接被调用。虽然无法被直接调用，实际上你却经常在使用这些方法，比如创建对象实例时就调用了 `__new__()` 和 `__init__` 方法。那么这些魔法方法还有那些其它的有趣用法呢？

操作符重载

Python 中的操作符运算实际上都是在隐式地调用这些魔法方法，如果你重写了对应的魔法方法，就能修改操作符的运算规则。比如，逻辑或操作符 `|` 对应了魔法方法 `__ror__`，因此我们可以重载 `__ror__` 来实现类似 Shell 中的管道：

```

1. class Pipe(object):
2.     def __init__(self, func):
3.         self.func = func
4.
5.     def __ror__(self, other):
6.         def generator():
7.             for obj in other:
8.                 if obj is not None:
9.                     yield self.func(obj)
10.        return generator()
11.
12. @Pipe
13. def even_filter(num):
14.     return num if num % 2 == 0 else None
15.
16. @Pipe
17. def multiply_by_three(num):
18.     return num*3
19.
20. @Pipe
21. def convert_to_string(num):
22.     return 'The Number: %s' % num
23.
24. @Pipe
25. def echo(item):
26.     print item

```

```

27.     return item
28.
29. def force(sqs):
30.     for item in sqs: pass
31.
32. nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
33.
34. force(nums | even_filter | multiply_by_three | convert_to_string | echo)

```

(代码出自[酷壳网友](#))

这里有张详细的对照表：

```

1. Binary Operators
2. Operator      Method
3. +   object.__add__(self, other)
4. -   object.__sub__(self, other)
5. *   object.__mul__(self, other)
6. //  object.__floordiv__(self, other)
7. /   object.__div__(self, other)
8. %   object.__mod__(self, other)
9. **  object.__pow__(self, other[, modulo])
10. << object.__lshift__(self, other)
11. >> object.__rshift__(self, other)
12. &   object.__and__(self, other)
13. ^   object.__xor__(self, other)
14. |   object.__or__(self, other)
15.
16. Extended Assignments
17. Operator      Method
18. += object.__iadd__(self, other)
19. -= object.__isub__(self, other)
20. *= object.__imul__(self, other)
21. /= object.__idiv__(self, other)
22. //= object.__ifloordiv__(self, other)
23. %=  object.__imod__(self, other)
24. **= object.__ipow__(self, other[, modulo])
25. <=<= object.__ilshift__(self, other)
26. >=>= object.__irshift__(self, other)
27. &=  object.__iand__(self, other)
28. ^=  object.__ixor__(self, other)
29. |=  object.__ior__(self, other)
30.
31. Unary Operators
32. Operator      Method
33. -   object.__neg__(self)
34. +   object.__pos__(self)

```

```
35. abs()    object.__abs__(self)
36. ~       object.__invert__(self)
37. complex() object.__complex__(self)
38. int()    object.__int__(self)
39. long()   object.__long__(self)
40. float()  object.__float__(self)
41. oct()    object.__oct__(self)
42. hex()    object.__hex__(self)
43.
44. Comparison Operators
45. Operator    Method
46. <           object.__lt__(self, other)
47. <=          object.__le__(self, other)
48. ==          object.__eq__(self, other)
49. !=          object.__ne__(self, other)
50. >=          object.__ge__(self, other)
51. >           object.__gt__(self, other)
```

参考: [Magic Methods and Operator Overloading](#)

Jenkins

- [Jenkins](#)

Jenkins

文档:

- <https://pythonhosted.org/jenkinsapi/>
- <https://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins>
- <https://python-jenkins.readthedocs.org/en/latest/example.html>

Escaping

- [Escaping](#)
 - [Markdown](#)
 - [Misaka](#)
 - [rST](#)

Escaping

Markdown

Misaka

- [官网](#)
- [API](#)

rST

Misaka

- [Misaka](#) 与其它 Python Markdown 库简易对比

Misaka 与其它 Python Markdown 库简易对比

[@Lepture](#) 在自己的英文博客中对常见的 Python Markdown 库做了一些测试和对比这里简单地做一下总结。

他介绍到的，知名的 Python Markdown 库就有：

- Misaka: Sundown 的 Python 封装（依赖 CPython）
- Hoedown: Hoedown 的 Python 封装 Misaka 的继承者。
- Discount: Discount 的 Python 封装（依赖 CPython）
- cMarkdown: Python Markdown 库，依靠 C 提升性能（依赖 CPython）
- Markdown: 纯粹的 Markdown 实现，老牌 Python Markdown 库。
- Markdown2: 另一个纯粹的 Markdown 库
- mistune: 作者 [@Lepture](#)

出于性能考虑，其中 Misaka Hoedown CMarkdown mistune 都是对 C 库的封装，其中 Misaka 和 mistune 都来自 Sundown，Python 的 Hoedown 库封装自 C 的 Hoedown，cMarkdown 来自于 upskirt，而其 C 语言库 Sundown fork 自 upskirt，Sundown 停止更新后，Hoedown 又 fork 了一份，因此 Hoedown 是最具有活力的那一个，也是 Misaka 的作者所推荐的 Markdown 库。

Markdown 和 Markdown2 则是纯粹的 Python 库，虽然 Markdown2 从名字上看起来比 Markdown 要更新鲜，而且它也宣称自己比 Markdown 快上一倍，但在 [@Lepture](#) 的测试中得出了恰恰相反的结论。除了性能上，Markdown2 也并没有提供更多的功能。

Mistune 是 [@Lepture](#) 自己发布的 Markdown 库，提供与 Misaka 相似的接口，但是在纯 Python 环境中就能达到 Markdown 4 倍的性能，在 CPython 的帮助下更能提升到 5 倍！

在 [@Lepture](#) 的测试中 Discount 则安装失败。

性能测试的结果如下：

```
1. Parsing the Markdown Syntax document 1000 times...
2. Mistune: 12.7255s
3. Mistune (with Cython): 9.74075s
```

```
4. Misaka: 0.550502s
5. Markdown: 46.4342s
6. Markdown2: 78.2267s
7. cMarkdown: 0.664128s
8. Discount is not available
```

测试用的源代码可以在[这里](#)获取。

原文: [Markdown Parsers in Python—Lepture](#)