**Python 基础教程**



Python是一种解释型、面向对象、动态数据类型的高级程序设计语言。

Python由Guido van Rossum于1989年底发明，第一个公开发行版发行于1991年。

像Perl语言一样, Python 源代码同样遵循 GPL(GNU General Public License)协议。

[现在开始学习 Python！](http://www.runoob.com/python/python-intro.html)

## 谁适合阅读本教程？

本教程适合想从零开始学习Python编程语言的开发人员。当然本教程也会对一些模块进行深入，让你更好的了解Python的应用。

## 学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过PHP，ASP等编程语言，将有助于你更快的了解Python编程。

## 执行Python程序

对于大多数程序语言，第一个入门编程代码便是"Hello World！"，以下代码为使用Python输出"Hello World！"：

## 实例(Python 2.0+)

#!/usr/bin/python  
print "Hello, World!";

[**运行实例 »**](http://www.runoob.com/try/showpy.php?filename=HelloWorld&language=py)

#!/usr/bin/python

print "Hello, World!";

Python 3.0+版本已经把print作为一个内置函数，正确输出"Hello World！"代码如下：

## 实例(Python 3.0+)

#!/usr/bin/python  
print("Hello, World!");

[**运行实例 »**](http://www.runoob.com/try/showpy.php?filename=HelloWorld&language=py3)

#!/usr/bin/python

print("Hello, World!");

# Python IDE

本文为大家推荐几款款不错的**Python IDE**（集成开发环境），比较推荐 PyCharm，当然你可以根据自己的喜好来选择适合自己的 Python IDE。

## PyCharm

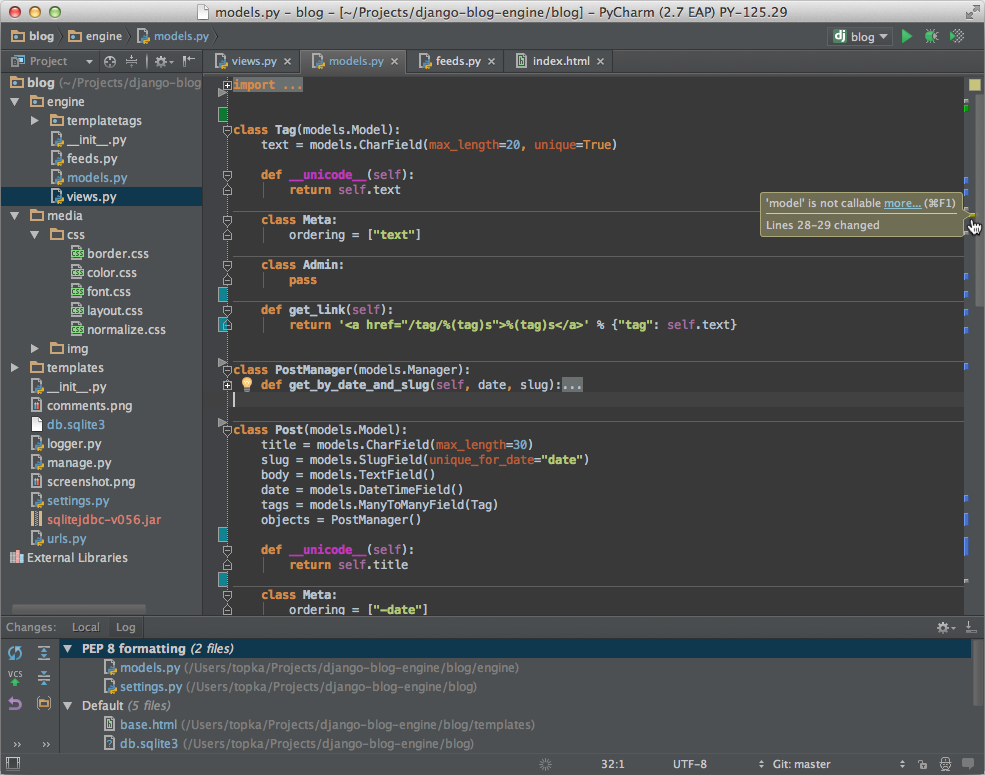
PyCharm是由JetBrains打造的一款Python IDE。

PyCharm具备一般 **Python IDE** 的功能，比如：调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。

另外，PyCharm还提供了一些很好的功能用于Django开发，同时支持Google App Engine，更酷的是，PyCharm支持IronPython。

PyCharm 官方下载地址：<http://www.jetbrains.com/pycharm/download/>

效果图查看：

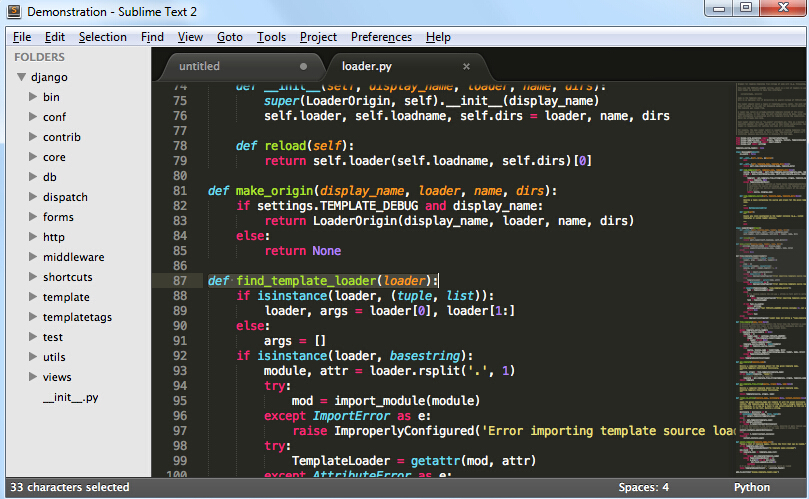


## Sublinme Text 2

Sublime Text具有漂亮的用户界面和强大的功能，例如代码缩略图，Python的插件，代码段等。还可自定义键绑定，菜单和工具栏。

Sublime Text 的主要功能包括：拼写检查，书签，完整的 Python API ， Goto 功能，即时项目切换，多选择，多窗口等等。

Sublime Text 是一个跨平台的编辑器，同时支持Windows、Linux、Mac OS X等操作系统。



使用Sublinme Text 2的插件扩展功能，你可以轻松的打造一款不错的Python IDE，以下推荐几款插件（你可以找到更多）：

* CodeIntel：自动补全+成员/方法提示（强烈推荐）
* SublimeREPL：用于运行和调试一些需要交互的程序（E.G. 使用了Input()的程序）
* Bracket Highlighter：括号匹配及高亮
* SublimeLinter：代码pep8格式检查

## Eclipse+Pydev

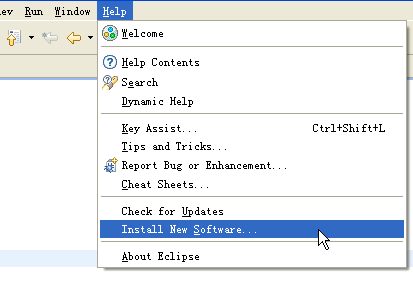
### 1、安装Eclipse

Eclipse可以在它的官方网站[Eclipse.org](http://eclipse.org/)找到并下载，通常我们可以选择适合自己的Eclipse版本，比如Eclipse Classic。下载完成后解压到到你想安装的目录中即可。

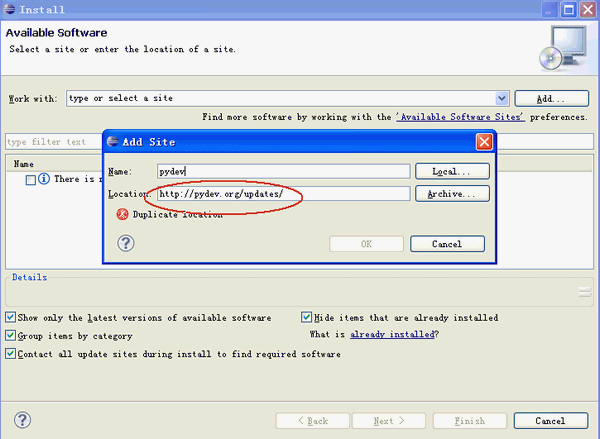
当然在执行Eclipse之前，你必须确认安装了Java运行环境,即必须安装JRE或JDK，你可以到（<http://www.java.com/en/download/manual.jsp>）找到JRE下载并安装。

### 2、安装Pydev

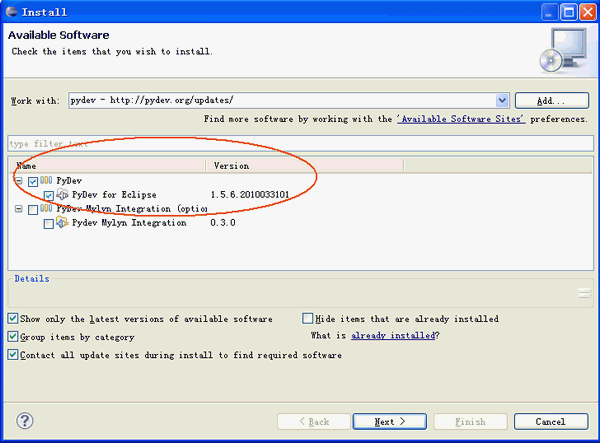
运行Eclipse之后，选择help-->Install new Software，如下图所示。



点击Add，添加pydev的安装地址：<http://pydev.org/updates/>，如下图所示。



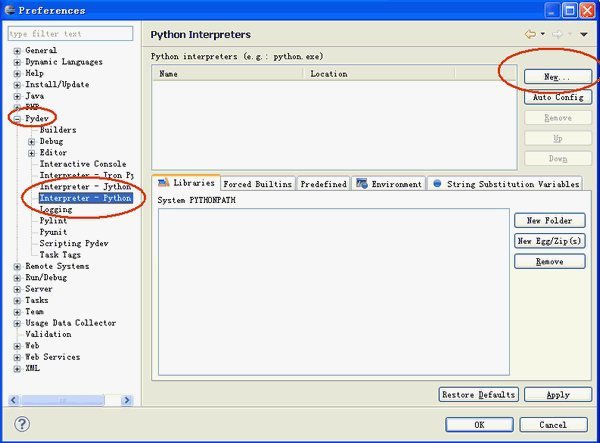
完成后点击"ok"，接着点击PyDev的"+"，展开PyDev的节点，要等一小段时间，让它从网上获取PyDev的相关套件，当完成后会多出PyDev的相关套件在子节点里，勾选它们然后按next进行安装。如下图所示。



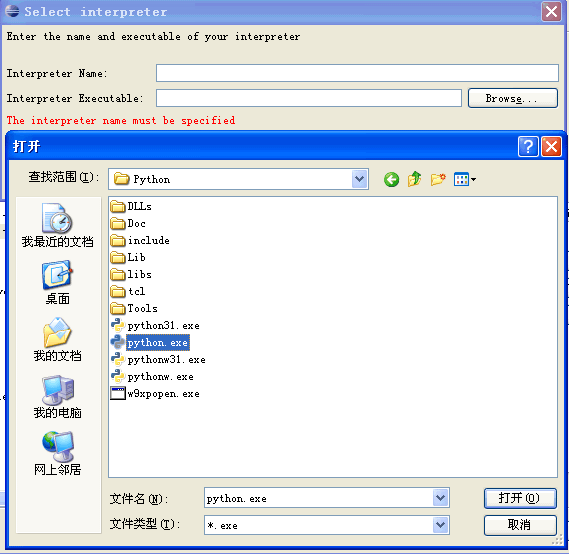
安装完成后，重启Eclipse即可

### 3、设置Pydev

安装完成后，还需要设置一下PyDev，选择Window -> Preferences来设置PyDev。设置Python的路径，从Pydev的Interpreter - Python页面选择New



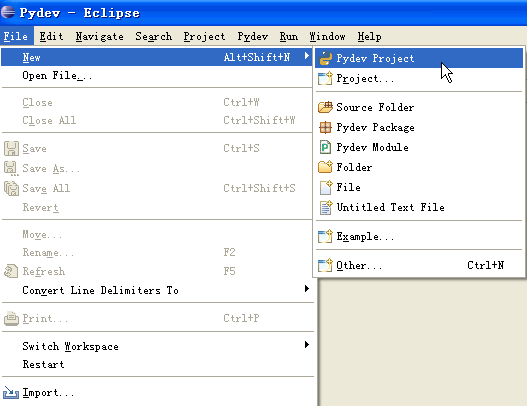
会弹出一个窗口让你选择Python的安装位置，选择你安装Python的所在位置。



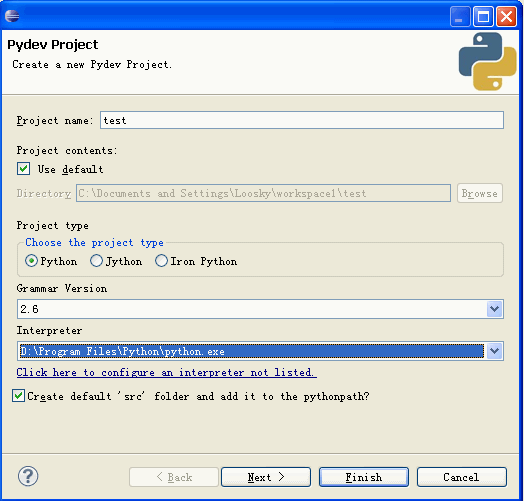
完成之后PyDev就设置完成，可以开始使用。

### 4、建立Python Project：

安装好Eclipse+PyDev以后，我们就可以开始使用它来开发项目了。首先要创建一个项目，选择File -> New ->Pydev Project

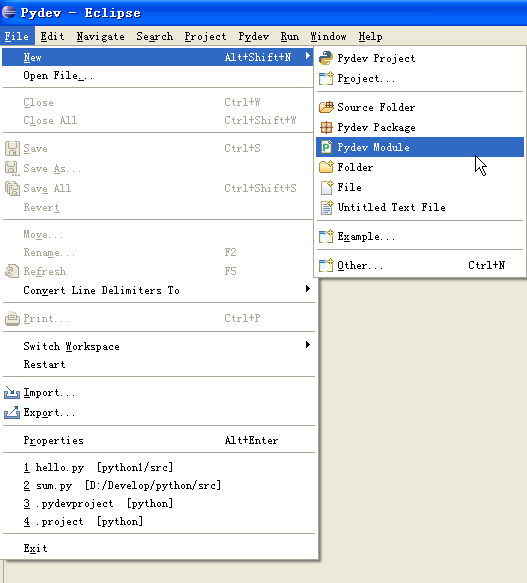


会弹出一个新窗口，填写Project Name，以及项目保存地址，然后点击next完成项目的创建。

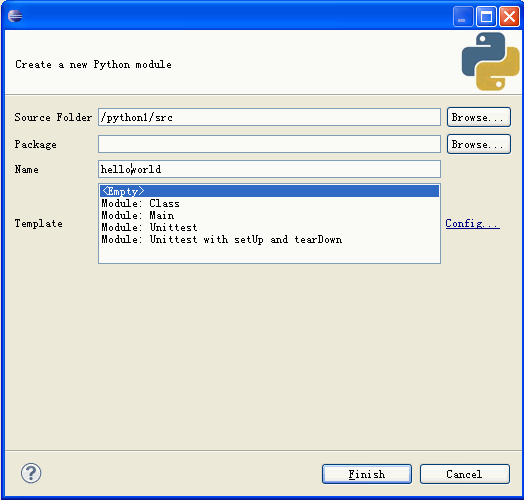


### 5、创建新的Pydev Module

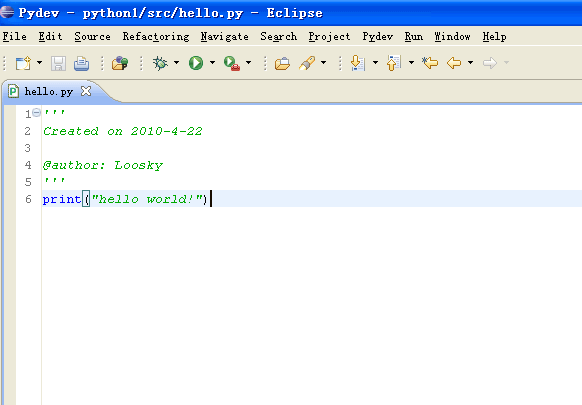
光有项目是无法执行的，接着必须创建新的Pydev Moudle，选择File -> New -> Pydev Module



在弹出的窗口中选择文件存放位置以及Moudle Name，注意Name不用加.py，它会自动帮助我们添加。然后点击Finish完成创建。

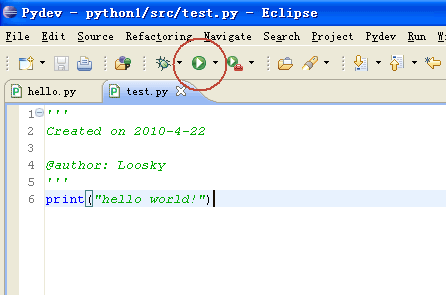


输入"hello world"的代码。

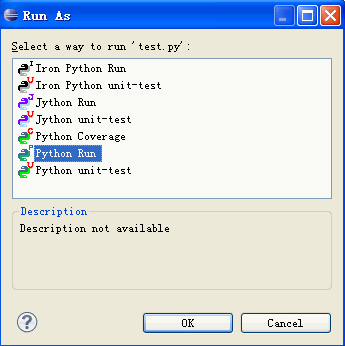


### 6、执行程序

程序写完后，我们可以开始执行程序,在上方的工具栏上面找到执行的按钮。



之后会弹出一个让你选择执行方式的窗口，通常我们选择Python Run，开始执行程序。



## 更多 Python IDE

推荐10 款最好的 Python IDE：<http://www.w3cschool.cc/w3cnote/best-python-ide-for-developers.html>

当然还有非常多很棒的 Python IDE，你可以自由的选择，更多 Python IDE 请参阅：<http://wiki.python.org/moin/PythonEditors>

## Python 编码规范(Google)

### 分类 [编程技术](http://www.runoob.com/w3cnote/w3cnote_genre/code)

Python 风格规范(Google)

本项目并非 Google 官方项目, 而是由国内程序员凭热情创建和维护。

如果你关注的是 Google 官方英文版, 请移步 [Google Style Guide](https://google-styleguide.googlecode.com/svn/trunk/pyguide.html)

以下代码中 **Yes** 表示推荐，**No** 表示不推荐。

## 分号

不要在行尾加分号, 也不要用分号将两条命令放在同一行。

## 行长度

每行不超过80个字符

以下情况除外：

1. 长的导入模块语句
2. 注释里的URL

不要使用反斜杠连接行。

Python会将 [圆括号, 中括号和花括号中的行隐式的连接起来](http://docs.python.org/2/reference/lexical_analysis.html#implicit-line-joining) , 你可以利用这个特点. 如果需要, 你可以在表达式外围增加一对额外的圆括号。

推荐: foo\_bar(self, width, height, color='black', design=None, x='foo',

emphasis=None, highlight=0)

if (width == 0 and height == 0 and

color == 'red' and emphasis == 'strong'):

如果一个文本字符串在一行放不下, 可以使用圆括号来实现隐式行连接:

x = ('这是一个非常长非常长非常长非常长 '

'非常长非常长非常长非常长非常长非常长的字符串')

在注释中，如果必要，将长的URL放在一行上。

Yes: # See details at

# http://www.example.com/us/developer/documentation/api/content/v2.0/csv\_file\_name\_extension\_full\_specification.html

No: # See details at

# http://www.example.com/us/developer/documentation/api/content/\

# v2.0/csv\_file\_name\_extension\_full\_specification.html

注意上面例子中的元素缩进; 你可以在本文的 :ref:`缩进 <indentation>`部分找到解释.

## 括号

宁缺毋滥的使用括号

除非是用于实现行连接, 否则不要在返回语句或条件语句中使用括号. 不过在元组两边使用括号是可以的.

Yes: if foo:

bar()

while x:

x = bar()

if x and y:

bar()

if not x:

bar()

return foo

for (x, y) in dict.items(): ...

No: if (x):

bar()

if not(x):

bar()

return (foo)

## 缩进

用4个空格来缩进代码

绝对不要用tab, 也不要tab和空格混用. 对于行连接的情况, 你应该要么垂直对齐换行的元素(见 :ref:`行长度 <line\_length>` 部分的示例), 或者使用4空格的悬挂式缩进(这时第一行不应该有参数):

Yes: # 与起始变量对齐

foo = long\_function\_name(var\_one, var\_two,

var\_three, var\_four)

# 字典中与起始值对齐

foo = {

long\_dictionary\_key: value1 +

value2,

...

}

# 4 个空格缩进，第一行不需要

foo = long\_function\_name(

var\_one, var\_two, var\_three,

var\_four)

# 字典中 4 个空格缩进

foo = {

long\_dictionary\_key:

long\_dictionary\_value,

...

}

No: # 第一行有空格是禁止的

foo = long\_function\_name(var\_one, var\_two,

var\_three, var\_four)

# 2 个空格是禁止的

foo = long\_function\_name(

var\_one, var\_two, var\_three,

var\_four)

# 字典中没有处理缩进

foo = {

long\_dictionary\_key:

long\_dictionary\_value,

...

}

## 空行

顶级定义之间空两行, 方法定义之间空一行

顶级定义之间空两行, 比如函数或者类定义. 方法定义, 类定义与第一个方法之间, 都应该空一行. 函数或方法中, 某些地方要是你觉得合适, 就空一行.

## 空格

按照标准的排版规范来使用标点两边的空格

括号内不要有空格.

按照标准的排版规范来使用标点两边的空格

Yes: spam(ham[1], {eggs: 2}, [])

No: spam( ham[ 1 ], { eggs: 2 }, [ ] )

不要在逗号, 分号, 冒号前面加空格, 但应该在它们后面加(除了在行尾).

Yes: if x == 4:

print x, y

x, y = y, x

No: if x == 4 :

print x , y

x , y = y , x

参数列表, 索引或切片的左括号前不应加空格.

Yes: spam(1)

no: spam (1)

Yes: dict['key'] = list[index]

No: dict ['key'] = list [index]

在二元操作符两边都加上一个空格, 比如赋值(=), 比较(==, <, >, !=, <>, <=, >=, in, not in, is, is not), 布尔(and, or, not). 至于算术操作符两边的空格该如何使用, 需要你自己好好判断. 不过两侧务必要保持一致.

Yes: x == 1

No: x<1

当'='用于指示关键字参数或默认参数值时, 不要在其两侧使用空格.

Yes: def complex(real, imag=0.0): return magic(r=real, i=imag)

No: def complex(real, imag = 0.0): return magic(r = real, i = imag)

不要用空格来垂直对齐多行间的标记, 因为这会成为维护的负担(适用于:, #, =等):

Yes:

foo = 1000 # 注释

long\_name = 2 # 注释不需要对齐

dictionary = {

"foo": 1,

"long\_name": 2,

}

No:

foo = 1000 # 注释

long\_name = 2 # 注释不需要对齐

dictionary = {

"foo" : 1,

"long\_name": 2,

}

## Shebang

大部分.py文件不必以#!作为文件的开始. 根据 [PEP-394](http://www.python.org/dev/peps/pep-0394/) , 程序的main文件应该以 #!/usr/bin/python2或者 #!/usr/bin/python3开始.

(译者注: 在计算机科学中, [Shebang](http://en.wikipedia.org/wiki/Shebang_(Unix)) (也称为Hashbang)是一个由井号和叹号构成的字符串行(#!), 其出现在文本文件的第一行的前两个字符. 在文件中存在Shebang的情况下, 类Unix操作系统的程序载入器会分析Shebang后的内容, 将这些内容作为解释器指令, 并调用该指令, 并将载有Shebang的文件路径作为该解释器的参数. 例如, 以指令#!/bin/sh开头的文件在执行时会实际调用/bin/sh程序.)

#!先用于帮助内核找到Python解释器, 但是在导入模块时, 将会被忽略. 因此只有被直接执行的文件中才有必要加入#!.

## 注释

确保对模块, 函数, 方法和行内注释使用正确的风格

**文档字符串**

*Python有一种独一无二的的注释方式: 使用文档字符串. 文档字符串是包, 模块, 类或函数里的第一个语句. 这些字符串可以通过对象的\_\_doc\_\_成员被自动提取, 并且被pydoc所用. (你可以在你的模块上运行pydoc试一把, 看看它长什么样). 我们对文档字符串的惯例是使用三重双引号"""(*[*PEP-257*](http://www.python.org/dev/peps/pep-0257/)*). 一个文档字符串应该这样组织: 首先是一行以句号, 问号或惊叹号结尾的概述(或者该文档字符串单纯只有一行). 接着是一个空行. 接着是文档字符串剩下的部分, 它应该与文档字符串的第一行的第一个引号对齐. 下面有更多文档字符串的格式化规范.*

**模块**

*每个文件应该包含一个许可样板. 根据项目使用的许可(例如, Apache 2.0, BSD, LGPL, GPL), 选择合适的样板.*

**函数和方法**

*下文所指的函数,包括函数, 方法, 以及生成器.*

*一个函数必须要有文档字符串, 除非它满足以下条件:*

1. 外部不可见
2. 非常短小
3. 简单明了

*文档字符串应该包含函数做什么, 以及输入和输出的详细描述. 通常, 不应该描述"怎么做", 除非是一些复杂的算法. 文档字符串应该提供足够的信息, 当别人编写代码调用该函数时, 他不需要看一行代码, 只要看文档字符串就可以了. 对于复杂的代码, 在代码旁边加注释会比使用文档字符串更有意义.*

*关于函数的几个方面应该在特定的小节中进行描述记录， 这几个方面如下文所述. 每节应该以一个标题行开始. 标题行以冒号结尾. 除标题行外, 节的其他内容应被缩进2个空格.*

Args:

列出每个参数的名字, 并在名字后使用一个冒号和一个空格, 分隔对该参数的描述.如果描述太长超过了单行80字符,使用2或者4个空格的悬挂缩进(与文件其他部分保持一致). 描述应该包括所需的类型和含义. 如果一个函数接受\*foo(可变长度参数列表)或者\*\*bar (任意关键字参数), 应该详细列出\*foo和\*\*bar.

Returns: (或者 Yields: 用于生成器)

描述返回值的类型和语义. 如果函数返回None, 这一部分可以省略.

Raises:

列出与接口有关的所有异常.

def fetch\_bigtable\_rows(big\_table, keys, other\_silly\_variable=None):

"""Fetches rows from a Bigtable.

Retrieves rows pertaining to the given keys from the Table instance

represented by big\_table. Silly things may happen if

other\_silly\_variable is not None.

Args:

big\_table: An open Bigtable Table instance.

keys: A sequence of strings representing the key of each table row

to fetch.

other\_silly\_variable: Another optional variable, that has a much

longer name than the other args, and which does nothing.

Returns:

A dict mapping keys to the corresponding table row data

fetched. Each row is represented as a tuple of strings. For

example:

{'Serak': ('Rigel VII', 'Preparer'),

'Zim': ('Irk', 'Invader'),

'Lrrr': ('Omicron Persei 8', 'Emperor')}

If a key from the keys argument is missing from the dictionary,

then that row was not found in the table.

Raises:

IOError: An error occurred accessing the bigtable.Table object.

"""

pass

**类**

*类应该在其定义下有一个用于描述该类的文档字符串. 如果你的类有公共属性(Attributes), 那么文档中应该有一个属性(Attributes)段. 并且应该遵守和函数参数相同的格式.*

class SampleClass(object):

"""Summary of class here.

Longer class information....

Longer class information....

Attributes:

likes\_spam: A boolean indicating if we like SPAM or not.

eggs: An integer count of the eggs we have laid.

"""

def \_\_init\_\_(self, likes\_spam=False):

"""Inits SampleClass with blah."""

self.likes\_spam = likes\_spam

self.eggs = 0

def public\_method(self):

"""Performs operation blah."""

**块注释和行注释**

*最需要写注释的是代码中那些技巧性的部分. 如果你在下次*[*代码审查*](http://en.wikipedia.org/wiki/Code_review)*的时候必须解释一下, 那么你应该现在就给它写注释. 对于复杂的操作, 应该在其操作开始前写上若干行注释. 对于不是一目了然的代码, 应在其行尾添加注释.*

# We use a weighted dictionary search to find out where i is in

# the array. We extrapolate position based on the largest num

# in the array and the array size and then do binary search to

# get the exact number.

if i & (i-1) == 0: # true iff i is a power of 2

*为了提高可读性, 注释应该至少离开代码2个空格.*

*另一方面, 绝不要描述代码. 假设阅读代码的人比你更懂Python, 他只是不知道你的代码要做什么.*

# BAD COMMENT: Now go through the b array and make sure whenever i occurs

# the next element is i+1

## 类

如果一个类不继承自其它类, 就显式的从object继承. 嵌套类也一样.

Yes: class SampleClass(object):

pass

class OuterClass(object):

class InnerClass(object):

pass

class ChildClass(ParentClass):

"""Explicitly inherits from another class already."""

No: class SampleClass:

pass

class OuterClass:

class InnerClass:

pass

继承自 object 是为了使属性(properties)正常工作, 并且这样可以保护你的代码, 使其不受Python 3000的一个特殊的潜在不兼容性影响. 这样做也定义了一些特殊的方法, 这些方法实现了对象的默认语义, 包括 \_\_new\_\_, \_\_init\_\_, \_\_delattr\_\_, \_\_getattribute\_\_, \_\_setattr\_\_, \_\_hash\_\_, \_\_repr\_\_, and \_\_str\_\_ .

## 字符串

Yes: x = a + b

x = '%s, %s!' % (imperative, expletive)

x = '{}, {}!'.format(imperative, expletive)

x = 'name: %s; score: %d' % (name, n)

x = 'name: {}; score: {}'.format(name, n)

No: x = '%s%s' % (a, b) # use + in this case

x = '{}{}'.format(a, b) # use + in this case

x = imperative + ', ' + expletive + '!'

x = 'name: ' + name + '; score: ' + str(n)

避免在循环中用+和+=操作符来累加字符串. 由于字符串是不可变的, 这样做会创建不必要的临时对象, 并且导致二次方而不是线性的运行时间. 作为替代方案, 你可以将每个子串加入列表, 然后在循环结束后用 .join 连接列表. (也可以将每个子串写入一个 cStringIO.StringIO 缓存中.)

Yes: items = ['<table>']

for last\_name, first\_name in employee\_list:

items.append('<tr><td>%s, %s</td></tr>' % (last\_name, first\_name))

items.append('</table>')

employee\_table = ''.join(items)

No: employee\_table = '<table>'

for last\_name, first\_name in employee\_list:

employee\_table += '<tr><td>%s, %s</td></tr>' % (last\_name, first\_name)

employee\_table += '</table>'

在同一个文件中, 保持使用字符串引号的一致性. 使用单引号'或者双引号"之一用以引用字符串, 并在同一文件中沿用. 在字符串内可以使用另外一种引号, 以避免在字符串中使用. PyLint已经加入了这一检查.

Yes:

Python('Why are you hiding your eyes?')

Gollum("I'm scared of lint errors.")

Narrator('"Good!" thought a happy Python reviewer.')

No:

Python("Why are you hiding your eyes?")

Gollum('The lint. It burns. It burns us.')

Gollum("Always the great lint. Watching. Watching.")

为多行字符串使用三重双引号"""而非三重单引号'''. 当且仅当项目中使用单引号'来引用字符串时, 才可能会使用三重'''为非文档字符串的多行字符串来标识引用. 文档字符串必须使用三重双引号""". 不过要注意, 通常用隐式行连接更清晰, 因为多行字符串与程序其他部分的缩进方式不一致.

Yes:

print ("This is much nicer.\n"

"Do it this way.\n")

No:

print """This is pretty ugly.

Don't do this.

"""

## 文件和sockets

在文件和sockets结束时, 显式的关闭它.

除文件外, sockets或其他类似文件的对象在没有必要的情况下打开, 会有许多副作用, 例如:

1. 它们可能会消耗有限的系统资源, 如文件描述符. 如果这些资源在使用后没有及时归还系统, 那么用于处理这些对象的代码会将资源消耗殆尽.
2. 持有文件将会阻止对于文件的其他诸如移动、删除之类的操作.
3. 仅仅是从逻辑上关闭文件和sockets, 那么它们仍然可能会被其共享的程序在无意中进行读或者写操作. 只有当它们真正被关闭后, 对于它们尝试进行读或者写操作将会跑出异常, 并使得问题快速显现出来.

而且, 幻想当文件对象析构时, 文件和sockets会自动关闭, 试图将文件对象的生命周期和文件的状态绑定在一起的想法, 都是不现实的. 因为有如下原因:

1. 没有任何方法可以确保运行环境会真正的执行文件的析构. 不同的Python实现采用不同的内存管理技术, 比如延时垃圾处理机制. 延时垃圾处理机制可能会导致对象生命周期被任意无限制的延长.
2. 对于文件意外的引用,会导致对于文件的持有时间超出预期(比如对于异常的跟踪, 包含有全局变量等).

推荐使用 ["with"语句](http://docs.python.org/reference/compound_stmts.html#the-with-statement) 以管理文件:

with open("hello.txt") as hello\_file:

for line in hello\_file:

print line

对于不支持使用"with"语句的类似文件的对象,使用 contextlib.closing():

import contextlib

with contextlib.closing(urllib.urlopen("http://www.python.org/")) as front\_page:

for line in front\_page:

print line

Legacy AppEngine 中Python 2.5的代码如使用"with"语句, 需要添加 "from \_\_future\_\_ import with\_statement".

## TODO注释

为临时代码使用TODO注释, 它是一种短期解决方案. 不算完美, 但够好了.

TODO注释应该在所有开头处包含"TODO"字符串, 紧跟着是用括号括起来的你的名字, email地址或其它标识符. 然后是一个可选的冒号. 接着必须有一行注释, 解释要做什么. 主要目的是为了有一个统一的TODO格式, 这样添加注释的人就可以搜索到(并可以按需提供更多细节). 写了TODO注释并不保证写的人会亲自解决问题. 当你写了一个TODO, 请注上你的名字.

# TODO(kl@gmail.com): Use a "\*" here for string repetition.

# TODO(Zeke) Change this to use relations.

如果你的TODO是"将来做某事"的形式, 那么请确保你包含了一个指定的日期("2009年11月解决")或者一个特定的事件("等到所有的客户都可以处理XML请求就移除这些代码").

## 导入格式

每个导入应该独占一行

Yes: import os

import sys

No: import os, sys

导入总应该放在文件顶部, 位于模块注释和文档字符串之后, 模块全局变量和常量之前. 导入应该按照从最通用到最不通用的顺序分组:

1. 标准库导入
2. 第三方库导入
3. 应用程序指定导入

每种分组中, 应该根据每个模块的完整包路径按字典序排序, 忽略大小写.

import foo

from foo import bar

from foo.bar import baz

from foo.bar import Quux

from Foob import ar

## 语句

通常每个语句应该独占一行

不过, 如果测试结果与测试语句在一行放得下, 你也可以将它们放在同一行. 如果是if语句, 只有在没有else时才能这样做. 特别地, 绝不要对 try/except 这样做, 因为try和except不能放在同一行.

Yes:

if foo: bar(foo)

No:

if foo: bar(foo)

else: baz(foo)

try: bar(foo)

except ValueError: baz(foo)

try:

bar(foo)

except ValueError: baz(foo)

## 访问控制

在Python中, 对于琐碎又不太重要的访问函数, 你应该直接使用公有变量来取代它们, 这样可以避免额外的函数调用开销. 当添加更多功能时, 你可以用属性(property)来保持语法的一致性.

(译者注: 重视封装的面向对象程序员看到这个可能会很反感, 因为他们一直被教育: 所有成员变量都必须是私有的! 其实, 那真的是有点麻烦啊. 试着去接受Pythonic哲学吧)

另一方面, 如果访问更复杂, 或者变量的访问开销很显著, 那么你应该使用像 get\_foo() 和 set\_foo() 这样的函数调用. 如果之前的代码行为允许通过属性(property)访问 , 那么就不要将新的访问函数与属性绑定. 这样, 任何试图通过老方法访问变量的代码就没法运行, 使用者也就会意识到复杂性发生了变化.

## 命名

module\_name, package\_name, ClassName, method\_name, ExceptionName, function\_name, GLOBAL\_VAR\_NAME, instance\_var\_name, function\_parameter\_name, local\_var\_name.

**应该避免的名称**

1. 单字符名称, 除了计数器和迭代器.
2. 包/模块名中的连字符(-)
3. 双下划线开头并结尾的名称(Python保留, 例如\_\_init\_\_)

**命名约定**

1. 所谓"内部(Internal)"表示仅模块内可用, 或者, 在类内是保护或私有的.
2. 用单下划线(\_)开头表示模块变量或函数是protected的(使用import \* from时不会包含).
3. 用双下划线(\_\_)开头的实例变量或方法表示类内私有.
4. 将相关的类和顶级函数放在同一个模块里. 不像Java, 没必要限制一个类一个模块.
5. 对类名使用大写字母开头的单词(如CapWords, 即Pascal风格), 但是模块名应该用小写加下划线的方式(如lower\_with\_under.py). 尽管已经有很多现存的模块使用类似于CapWords.py这样的命名, 但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致, 这会让人困扰.

**Python之父Guido推荐的规范**

| **Type** | **Public** | **Internal** |
| --- | --- | --- |
| Modules | lower\_with\_under | \_lower\_with\_under |
| Packages | lower\_with\_under |  |
| Classes | CapWords | \_CapWords |
| Exceptions | CapWords |  |
| Functions | lower\_with\_under() | \_lower\_with\_under() |
| Global/Class Constants | CAPS\_WITH\_UNDER | \_CAPS\_WITH\_UNDER |
| Global/Class Variables | lower\_with\_under | \_lower\_with\_under |
| Instance Variables | lower\_with\_under | \_lower\_with\_under (protected) or \_\_lower\_with\_under (private) |
| Method Names | lower\_with\_under() | \_lower\_with\_under() (protected) or \_\_lower\_with\_under() (private) |
| Function/Method Parameters | lower\_with\_under |  |
| Local Variables | lower\_with\_under |  |

## Main

即使是一个打算被用作脚本的文件, 也应该是可导入的. 并且简单的导入不应该导致这个脚本的主功能(main functionality)被执行, 这是一种副作用. 主功能应该放在一个main()函数中.

在Python中, pydoc以及单元测试要求模块必须是可导入的. 你的代码应该在执行主程序前总是检查 if \_\_name\_\_ == '\_\_main\_\_' , 这样当模块被导入时主程序就不会被执行.

def main():

...

if \_\_name\_\_ == '\_\_main\_\_':

main()

所有的顶级代码在模块导入时都会被执行. 要小心不要去调用函数, 创建对象, 或者执行那些不应该在使用pydoc时执行的操作.

# Python 简介

Python 是一个高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。

Python 的设计具有很强的可读性，相比其他语言经常使用英文关键字，其他语言的一些标点符号，它具有比其他语言更有特色语法结构。

* **Python 是一种解释型语言：** 这意味着开发过程中没有了编译这个环节。类似于PHP和Perl语言。
* **Python 是交互式语言：** 这意味着，您可以在一个Python提示符，直接互动执行写你的程序。
* **Python 是面向对象语言:** 这意味着Python支持面向对象的风格或代码封装在对象的编程技术。
* **Python 是初学者的语言：**Python 对初级程序员而言，是一种伟大的语言，它支持广泛的应用程序开发，从简单的文字处理到 WWW 浏览器再到游戏。

## Python 发展历史

Python 是由 Guido van Rossum 在八十年代末和九十年代初，在荷兰国家数学和计算机科学研究所设计出来的。

Python 本身也是由诸多其他语言发展而来的,这包括 ABC、Modula-3、C、C++、Algol-68、SmallTalk、Unix shell 和其他的脚本语言等等。

像 Perl 语言一样，Python 源代码同样遵循 GPL(GNU General Public License)协议。

现在 Python 是由一个核心开发团队在维护，Guido van Rossum 仍然占据着至关重要的作用，指导其进展。

## Python 特点

* **1.易于学习：**Python有相对较少的关键字，结构简单，和一个明确定义的语法，学习起来更加简单。
* **2.易于阅读：**Python代码定义的更清晰。
* **3.易于维护：**Python的成功在于它的源代码是相当容易维护的。
* **4.一个广泛的标准库：**Python的最大的优势之一是丰富的库，跨平台的，在UNIX，Windows和Macintosh兼容很好。
* **5.互动模式：**互动模式的支持，您可以从终端输入执行代码并获得结果的语言，互动的测试和调试代码片断。
* **6.可移植：**基于其开放源代码的特性，Python已经被移植（也就是使其工作）到许多平台。
* **7.可扩展：**如果你需要一段运行很快的关键代码，或者是想要编写一些不愿开放的算法，你可以使用C或C++完成那部分程序，然后从你的Python程序中调用。
* **8.数据库：**Python提供所有主要的商业数据库的接口。
* **9.GUI编程：**Python支持GUI可以创建和移植到许多系统调用。
* **10.可嵌入:**你可以将Python嵌入到C/C++程序，让你的程序的用户获得"脚本化"的能力。

# Python 环境搭建

本章节我们将向大家介绍如何在本地搭建Python开发环境。

Python可应用于多平台包括 Linux 和 Mac OS X。

你可以通过终端窗口输入 "python" 命令来查看本地是否已经安装Python以及Python的安装版本。

* Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, 等等。)
* Win 9x/NT/2000
* Macintosh (Intel, PPC, 68K)
* OS/2
* DOS (多个DOS版本)
* PalmOS
* Nokia 移动手机
* Windows CE
* Acorn/RISC OS
* BeOS
* Amiga
* VMS/OpenVMS
* QNX
* VxWorks
* Psion
* Python 同样可以移植到 Java 和 .NET 虚拟机上。

## Python下载

Python最新源码，二进制文档，新闻资讯等可以在Python的官网查看到：

Python官网：<http://www.python.org/>

你可以在一下链接中下载Python的文档，你可以下载 HTML、PDF 和 PostScript 等格式的文档。

Python文档下载地址：[www.python.org/doc/](http://www.python.org/doc/)

## Python安装

Python已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。

您需要下载适用于您使用平台的二进制代码，然后安装Python。

如果您平台的二进制代码是不可用的，你需要使用C编译器手动编译源代码。

编译的源代码，功能上有更多的选择性， 为python安装提供了更多的灵活性。

以下为不同平台上安装Python的方法：

### Unix & Linux 平台安装 Python:

以下为在Unix & Linux 平台上安装 Python 的简单步骤：

* 打开WEB浏览器访问<http://www.python.org/download/>
* 选择适用于Unix/Linux的源码压缩包。
* 下载及解压压缩包。
* 如果你需要自定义一些选项修改*Modules/Setup*
* **执行** ./configure 脚本
* make
* make install

执行以上操作后，Python会安装在 /usr/local/bin 目录中，Python库安装在/usr/local/lib/pythonXX，XX为你使用的Python的版本号。

### Window 平台安装 Python:

以下为在 Window 平台上安装 Python 的简单步骤：

* 打开WEB浏览器访问<http://www.python.org/download/>
* 在下载列表中选择Window平台安装包，包格式为：*python-XYZ.msi* 文件 ， XYZ 为你要安装的版本号。
* 要使用安装程序 *python-XYZ.msi*, Windows系统必须支持Microsoft Installer 2.0搭配使用。只要保存安装文件到本地计算机，然后运行它，看看你的机器支持MSI。Windows XP和更高版本已经有MSI，很多老机器也可以安装MSI。
* 下载后，双击下载包，进入Python安装向导，安装非常简单，你只需要使用默认的设置一直点击"下一步"直到安装完成即可。

### MAC 平台安装 Python:

最近的Macs系统都自带有Python环境，你也可以在链接 <http://www.python.org/download/> 上下载最新版安装。

## 环境变量配置

程序和可执行文件可以在许多目录，而这些路径很可能不在操作系统提供可执行文件的搜索路径中。

path(路径)存储在环境变量中，这是由操作系统维护的一个命名的字符串。这些变量包含可用的命令行解释器和其他程序的信息。

Unix或Windows中路径变量为PATH（UNIX区分大小写，Windows不区分大小写）。

在Mac OS中，安装程序过程中改变了python的安装路径。如果你需要在其他目录引用Python，你必须在path中添加Python目录。

### 在 Unix/Linux 设置环境变量

* **在 csh shell:** 输入

setenv PATH "$PATH:/usr/local/bin/python"

, 按下"Enter"。

* **在 bash shell (Linux):** 输入

export PATH="$PATH:/usr/local/bin/python"

，按下"Enter"。

* **在 sh 或者 ksh shell:** 输入

PATH="$PATH:/usr/local/bin/python"

, 按下"Enter"。

**注意:**/usr/local/bin/python 是Python的安装目录。

### 在 Windows 设置环境变量

在环境变量中添加Python目录：

**在命令提示框中(cmd) :** 输入

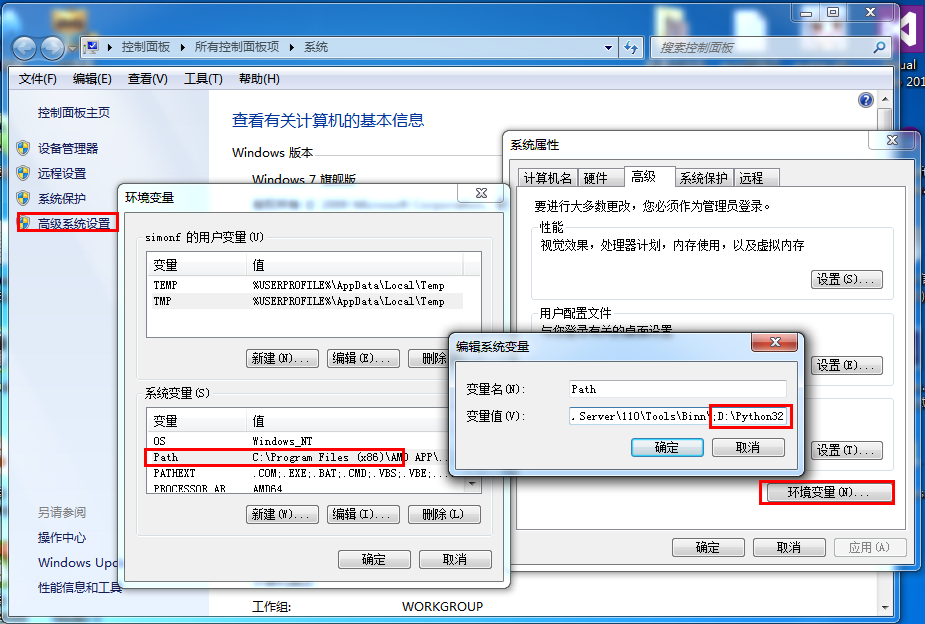
path=%path%;C:\Python

按下"Enter"。

**注意:**C:\Python 是Python的安装目录。

也可以通过以下方式设置：

* 右键点击"计算机"，然后点击"属性"
* 然后点击"高级系统设置"
* 选择"系统变量"窗口下面的"Path",双击即可！
* 然后在"Path"行，添加python安装路径即可(我的D:\Python32)，所以在后面，添加该路径即可。 **ps：记住，路径直接用分号"；"隔开！**
* 最后设置成功以后，在cmd命令行，输入命令"python"，就可以有相关显示。



## Python 环境变量

下面几个重要的环境变量，它应用于Python：

|  |  |
| --- | --- |
| **变量名** | **描述** |
| PYTHONPATH | PYTHONPATH是Python搜索路径，默认我们import的模块都会从PYTHONPATH里面寻找。 |
| PYTHONSTARTUP | Python启动后，先寻找PYTHONSTARTUP环境变量，然后执行此文件中变量指定的执行代码。 |
| PYTHONCASEOK | 加入PYTHONCASEOK的环境变量, 就会使python导入模块的时候不区分大小写. |
| PYTHONHOME | 另一种模块搜索路径。它通常内嵌于的PYTHONSTARTUP或PYTHONPATH目录中，使得两个模块库更容易切换。 |

## 运行Python

有三种方式可以运行Python：

### 1、交互式解释器：

你可以通过命令行窗口进入python并开在交互式解释器中开始编写Python代码。

你可以在Unix，DOS或任何其他提供了命令行或者shell的系统进行python编码工作。

$ python # Unix/Linux   
  
或者   
  
C:>python # Windows/DOS

以下为Python命令行参数：

|  |  |
| --- | --- |
| **选项** | **描述** |
| -d | 在解析时显示调试信息 |
| -O | 生成优化代码 ( .pyo 文件 ) |
| -S | 启动时不引入查找Python路径的位置 |
| -V | 输出Python版本号 |
| -X | 从 1.6版本之后基于内建的异常（仅仅用于字符串）已过时。 |
| -c cmd | 执行 Python 脚本，并将运行结果作为 cmd 字符串。 |
| file | 在给定的python文件执行python脚本。 |

### 2、命令行脚本

在你的应用程序中通过引入解释器可以在命令行中执行Python脚本，如下所示：

$ python script.py # Unix/Linux   
  
或者   
  
C:>python script.py # Windows/DOS

**注意：**在执行脚本时，请检查脚本是否有可执行权限。

### 3、集成开发环境（IDE：Integrated Development Environment）

您可以使用图形用户界面（GUI）环境来编写及运行Python代码。以下推荐各个平台上使用的IDE：

* **Unix:** IDLE 是 UNIX 上最早的 Python IDE 。
* **Windows:** PythonWin 是一个 Python 集成开发环境,在许多方面都比 IDE 优秀
* **Macintosh:** Python 的 Mac 可以使用 IDLE IDE，你可以在网站上下载对应MAC的IDLE 。

继续下一章之前，请确保您的环境已搭建成功。如果你不能够建立正确的环境，那么你就可以从您的系统管理员的帮助。

在以后的章节中给出的例子已在 Python2.7.6 版本测试通过。

# Python 中文编码

前面章节中我们已经学会了如何用 Python 输出 "Hello, World!"，英文没有问题，但是如果你输出中文字符"你好，世界"就有可能会碰到中文编码问题。

Python 文件中如果未指定编码，在执行过程会出现报错：

#!/usr/bin/python

print "你好，世界";

以上程序执行输出结果为：

File "test.py", line 2

SyntaxError: Non-ASCII character '\xe4' in file test.py on line 2, but no encoding declared; see http://www.python.org/peps/pep-0263.html for details

Python中默认的编码格式是 ASCII 格式，在没修改编码格式时无法正确打印汉字，所以在读取中文时会报错。

解决方法为只要在文件开头加入 **# -\*- coding: UTF-8 -\*-** 或者 **#coding=utf-8** 就行了。

## 实例(Python 2.0+)

#!/usr/bin/python  
# -\*- coding: UTF-8 -\*-  
  
print "你好，世界";

[**运行实例 »**](http://www.runoob.com/try/showpy.php?filename=helloworld_cn&language=py)

输出结果为：

你好，世界

所以如果大家再学习过程中，代码中包含中文，就需要在头部指定编码。

**注意：**Python3.X 源码文件默认使用utf-8编码，所以可以正常解析中文，无需指定 UTF-8 编码。

# Python 基础语法

Python语言与Perl，C和Java等语言有许多相似之处。但是，也存在一些差异。

在本章中我们将来学习Python的基础语法，让你快速学会Python编程。

## 第一个Python程序

### 交互式编程

交互式编程不需要创建脚本文件，是通过 Python 解释器的交互模式进来编写代码。

linux上你只需要在命令行中输入 Python 命令即可启动交互式编程,提示窗口如下：

$ python

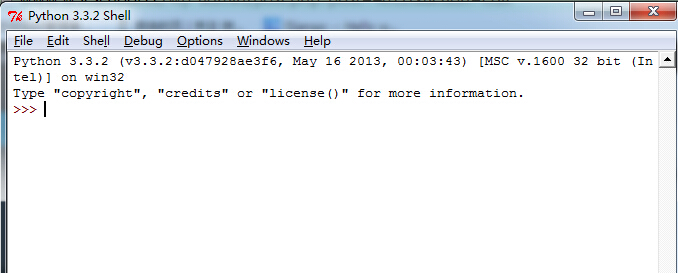
Python 2.7.6 (default, Sep 9 2014, 15:04:36)

[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>>

Window上在安装Python时已经已经安装了默认的交互式编程客户端，提示窗口如下：



在 python 提示符中输入以下文本信息，然后按 Enter 键查看运行效果：

>>> print "Hello, Python!";

在 Python 2.7.6 版本中,以上事例输出结果如下：

Hello, Python!

如果您运行的是新版本的Python，那么你就需要在print语句中使用括号如：

>>> print ("Hello, Python!");

### 脚本式编程

通过脚本参数调用解释器开始执行脚本，直到脚本执行完毕。当脚本执行完成后，解释器不再有效。

让我们写一个简单的Python脚本程序。所有Python文件将以.py为扩展名。将以下的源代码拷贝至test.py文件中。

print "Hello, Python!";

这里，假设你已经设置了Python解释器PATH变量。使用以下命令运行程序：

$ python test.py

输出结果：

Hello, Python!

让我们尝试另一种方式来执行Python脚本。修改test.py文件，如下所示：

#!/usr/bin/python

print "Hello, Python!";

这里，假定您的Python解释器在/usr/bin目录中，使用以下命令执行脚本：

$ chmod +x test.py # 脚本文件添加可执行权限

$./test.py

输出结果：

Hello, Python!

## Python 标识符

在python里，标识符有字母、数字、下划线组成。

在python中，所有标识符可以包括英文、数字以及下划线（\_），但不能以数字开头。

python中的标识符是区分大小写的。

以下划线开头的标识符是有特殊意义的。以单下划线开头（\_foo）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用"from xxx import \*"而导入；

以双下划线开头的（\_\_foo）代表类的私有成员；以双下划线开头和结尾的（\_\_foo\_\_）代表python里特殊方法专用的标识，如\_\_init\_\_（）代表类的构造函数。

## Python保留字符

下面的列表显示了在Python中的保留字。这些保留字不能用作常数或变数，或任何其他标识符名称。

所有Python的关键字只包含小写字母。

|  |  |  |
| --- | --- | --- |
| and | exec | not |
| assert | finally | or |
| break | for | pass |
| class | from | print |
| continue | global | raise |
| def | if | return |
| del | import | try |
| elif | in | while |
| else | is | with |
| except | lambda | yield |

## 行和缩进

学习Python与其他语言最大的区别就是，Python的代码块不使用大括号（{}）来控制类，函数以及其他逻辑判断。python最具特色的就是用缩进来写模块。

缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行。如下所示：

if True:

print "True"

else:

print "False"

以下代码将会执行错误：

#!/usr/bin/python

# -\*- coding: UTF-8 -\*-

# 文件名：test.py

if True:

print "Answer"

print "True"

else:

print "Answer"

# 没有严格缩进，在执行时保持

print "False"

$ python test.py

File "test.py", line 5

if True:

^

IndentationError: unexpected indent

**IndentationError: unexpected indent** 错误是python编译器是在告诉你"Hi，老兄，你的文件里格式不对了，可能是tab和空格没对齐的问题"，所有python对格式要求非常严格。

如果是 **IndentationError: unindent does not match any outer indentation level**错误表明，你使用的缩进方式不一致，有的是 tab 键缩进，有的是空格缩进，改为一致即可。

因此，在Python的代码块中必须使用相同数目的行首缩进空格数。

建议你在每个缩进层次使用 **单个制表符** 或 **两个空格** 或 **四个空格** , 切记不能混用

## 多行语句

Python语句中一般以新行作为为语句的结束符。

但是我们可以使用斜杠（ \）将一行的语句分为多行显示，如下所示：

total = item\_one + \

item\_two + \

item\_three

语句中包含[], {} 或 () 括号就不需要使用多行连接符。如下实例：

days = ['Monday', 'Tuesday', 'Wednesday',

'Thursday', 'Friday']

## Python 引号

Python 接收单引号(' )，双引号(" )，三引号(''' """) 来表示字符串，引号的开始与结束必须的相同类型的。

其中三引号可以由多行组成，编写多行文本的快捷语法，常用语文档字符串，在文件的特定地点，被当做注释。

word = 'word'

sentence = "这是一个句子。"

paragraph = """这是一个段落。

包含了多个语句"""

## Python注释

python中单行注释采用 # 开头。

#!/usr/bin/python

# -\*- coding: UTF-8 -\*-

# 文件名：test.py

# 第一个注释

print "Hello, Python!"; # 第二个注释

输出结果：

Hello, Python!

注释可以在语句或表达式行末：

name = "Madisetti" # 这是一个注释

python 中多行注释使用三个单引号(''')或三个双引号(""")。

#!/usr/bin/python

# -\*- coding: UTF-8 -\*-

# 文件名：test.py

'''

这是多行注释，使用单引号。

这是多行注释，使用单引号。

这是多行注释，使用单引号。

'''

"""

这是多行注释，使用双引号。

这是多行注释，使用双引号。

这是多行注释，使用双引号。

"""

## Python空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

空行与代码缩进不同，空行并不是Python语法的一部分。书写时不插入空行，Python解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

## 等待用户输入

下面的程序在按回车键后就会等待用户输入：

#!/usr/bin/python

raw\_input("\n\nPress the enter key to exit.")

以上代码中 ，"\n\n"在结果输出前会输出两个新的空行。一旦用户按下键时，程序将退出。

## 同一行显示多条语句

Python可以在同一行中使用多条语句，语句之间使用分号(;)分割，以下是一个简单的实例：

import sys; x = 'foo'; sys.stdout.write(x + '\n')

## 多个语句构成代码组

缩进相同的一组语句构成一个代码块，我们称之代码组。

像if、while、def和class这样的复合语句，首行以关键字开始，以冒号( : )结束，该行之后的一行或多行代码构成代码组。

我们将首行及后面的代码组称为一个子句(clause)。

如下实例：

if expression :

suite

elif expression :

suite

else :

suite

## 命令行参数

很多程序可以执行一些操作来查看一些基本信，Python可以使用-h参数查看各参数帮助信息：

$ python -h

usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...

Options and arguments (and corresponding environment variables):

-c cmd : program passed in as string (terminates option list)

-d : debug output from parser (also PYTHONDEBUG=x)

-E : ignore environment variables (such as PYTHONPATH)

-h : print this help message and exit

[ etc. ]