代码库和插件 FAQ

目录

通用的代码库问题

如何找到可以用来做 X 任务的模块或应用?

在 标准库参考 中查找是否有适合的标准库模块。（如果你已经了解标准库的内容，可以跳过这一步）

对于第三方软件包，请搜索 Python Package Index 或者是尝试 Google 或其他网络搜索引擎。 搜索 "Python" 加上一两个你感兴趣的关键词通常就会找到一些有用的信息。

math.py（socket.py，regex.py 等）的源文件在哪?

如果找不到模块的源文件，可能它是一个内建的模块，或是使用 C，C++ 或其他编译型语言实现的动态加载模块。这种情况下可能是没有源码文件的，类似 `mathmodule.c` 这样的文件会存放在 C 代码目录中（但不在 Python 目录中）。

Python 中（至少）有三类模块：

1. 使用 Python 编写的模块（.py）；
2. 使用 C 编写的动态加载模块（.dll，.pyd，.so，.sl 等）；
3. 使用 C 编写并链接到解释器的模块，要获取此列表，输入：

```
import sys
print(sys.builtin_module_names)
```

在 Unix 中怎样让 Python 脚本可执行?

你需要做两件事：文件必须是可执行的，并且第一行需要以 `#!` 开头，后面跟上 Python 解释器的路径。

第一点可以用执行 `chmod +x scriptfile` 或是 `chmod 755 scriptfile` 做到。

第二点有很多种做法，最直接的方式是：

```
#!/usr/local/bin/python
```

在文件第一行，使用你所在平台上的 Python 解释器的路径。

如果你希望脚本不依赖 Python 解释器的具体路径，你也可以使用 **env** 程序。假设你的 Python 解释器所在目录已经添加到了 PATH 环境变量中，几乎所有的类 Unix 系统都支持下面的写法：

```
#!/usr/bin/env python
```

不要 在 CGI 脚本中这样做。CGI 脚本的 PATH 环境变量通常会非常精简，所以你必须使用解释器的完整绝对路径。

有时候，用户的环境变量如果太长，可能会导致 **/usr/bin/env** 执行失败；又或者甚至根本就不存在 env 程序。在这种情况下，你可以尝试使用下面的 hack 方法（来自 Alex Rezinsky）：

```
#! /bin/sh
""":"
exec python $0 ${1+"$@"}
"""
```

这样做有一个小小的缺点，它会定义脚本的 __doc__ 字符串。不过可以这样修复：

```
__doc__ = """...Whatever..."""
```

Python 中有 curses/termcap 包吗?

对于类 Unix 系统：标准 Python 源码发行版会在 Modules 子目录中附带 curses 模块，但默认并不会编译。（注意：在 Windows 平台下不可用 —— Windows 中没有 curses 模块。）

`curses` 模块支持基本的 curses 特性，同时也支持 ncurses 和 SYSV curses 中的很多额外功能，比如颜色、不同的字符集支持、填充和鼠标支持。这意味着这个模块不兼容只有 BSD curses 模块的操作系统，但是目前仍在维护的系统应该都不会存在这种情况。

Python 中存在类似 C 的 onexit() 函数的东西吗?

`atexit` 模块提供了一个与 C 的 `onexit()` 类似的注册函数。

为什么我的信号处理函数不能工作?

最常见的问题是信号处理函数没有正确定义参数列表。它会被这样调用：

```
handler(signum, frame)
```

因此它应当声明为带有两个形参:

```
def handler(signum, frame):
    ...
```

通用任务

怎样测试 Python 程序或组件?

Python 带有两个测试框架。`doctest` 模块从模块的 docstring 中寻找示例并执行，对比输出是否与 docstring 中给出的是否一致。

`unittest` 模块是一个模仿 Java 和 Smalltalk 测试框架的更棒的测试框架。

为了使测试更容易，你应该在程序中使用良好的模块化设计。程序中的绝大多数功能都应该用函数或类方法封装 —— 有时这样做会有额外惊喜，程序会运行得更快（因为局部变量比全局变量访问要快）。除此之外，程序应该避免依赖可变的局部变量，这会使得测试困难许多。

程序的“全局主逻辑”应该尽量简单：

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

并放置在程序主模块的最后面。

一旦你的程序已经组织为一个函数和类行为的有完整集合，你就应该编写测试函数来检测这些行为。 可以将自动执行一系列测试的测试集关联到每个模块。 这听起来似乎需要大量的工作，但是由于 Python 是如此简洁灵活因此它会极其容易。 你可以通过与"生产代码"同步编写测试函数使编程更为愉快和有趣，因为这将更容易并更早发现代码问题甚至设计缺陷。

程序主模块之外的其他"辅助模块"中可以增加自测试的入口。

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

通过使用 Python 实现的"假"接口，即使是需要与复杂的外部接口交互的程序也可以在外部接口不可用时进行测试。

怎样用 docstring 创建文档?

pydoc 模块可以用你的 Python 源代码中的文档字符串来创建 HTML。 纯粹通过文档字符串来创建 API 文档的一种替代方案是 epydoc。 Sphinx 也可以包括文档字符串的内容。

怎样一次只获取一个按键?

在类 Unix 系统中有多种方案。最直接的方法是使用 curses，但是 curses 模块太大了，难以学习。

线程相关

程序中怎样使用线程?

一定要使用 threading 模块，不要使用 _thread 模块。 threading 模块对 _thread 模块提供的底层线程原语做了更易用的抽象。

我的线程都没有运行，为什么?

一旦主线程退出，所有的子线程都会被杀掉。你的主线程运行得太快了，子线程还没来得及工作。

简单的解决方法是在程序中加一个时间足够长的 sleep，让子线程能够完成运行。

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)  # <---------------------------!
```

但目前（在许多平台上）线程不是并行运行的，而是按顺序依次执行！原因是系统线程调度器在前一个线程阻塞之前不会启动新线程。

简单的解决方法是在运行函数的开始处加一个时间很短的 sleep。

```
def thread_task(name, n):
    time.sleep(0.001)  # <--------------------!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

比起用 `time.sleep()` 猜一个合适的等待时间，使用信号量机制会更好些。有一个办法是使用 `queue` 模块创建一个 queue 对象，让每一个线程在运行结束时 append 一个令牌到 queue 对象中，主线程中从 queue 对象中读取与线程数量一致的令牌数量即可。

如何将任务分配给多个工作线程？

最简单的方式是使用 `concurrent.futures` 模块，特别是其中的 `ThreadPoolExecutor` 类。

或者，如果你想更好地控制分发算法，你也可以自己写逻辑实现。使用 `queue` 模块来创建任务列表队列。`Queue` 类维护一个了一个存有对象的列表，提供了 `.put(obj)` 方法添加元素，并且可以用 `.get()` 方法获取元素。这个类会使用必要的加锁操作，以此确保每个任务只会执行一次。

这是一个简单的例子：

```
import threading, queue, time

# 工作线程会将任务移出队列。 当队列为空时，
# 它将认为工作已完成并退出。
# （在真实场景下工作线程将持续运行直到被终结。）
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.current_thread(), end=' ')
            print('queue empty')
            break
        else:
            print('Worker', threading.current_thread(), end=' ')
            print('running with argument', arg)
            time.sleep(0.5)

# 创建队列
q = queue.Queue()

# 启动包含 5 个工作线程的线程池
for i in range(5):
```

```
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# 开始向队列添加任务
for i in range(50):
    q.put(i)

# 为线程留出运行的时间
print('Main thread sleeping')
time.sleep(5)
```

运行时会产生如下输出：

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

查看模块的文档以获取更多信息；`Queue` 类提供了多种接口。

怎样修改全局变量是线程安全的？

Python VM 内部会使用 global interpreter lock （GIL）来确保同一时间只有一个线程运行。通常 Python 只会在字节码指令之间切换线程；切换的频率可以通过设置 `sys.setswitchinterval()` 指定。从 Python 程序的角度来看，每一条字节码指令以及每一条指令对应的 C 代码实现都是原子的。

理论上说，具体的结果要看具体的 PVM 字节码实现对指令的解释。而实际上，对内建类型（int，list，dict 等）的共享变量的“类原子”操作都是原子的。

举例来说，下面的操作是原子的（L、L1、L2 是列表，D、D1、D2 是字典，x、y 是对象，i，j 是 int 变量）：

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

这些不是原子的：

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

替换其他对象的操作可能会在其他对象的引用计数变为零时唤起这些对象的 `__del__()` 方法，这可能会产生一些影响。 对字典和列表进行大量更新尤其如此。 如有疑问，请使用互斥锁！

不能删除全局解释器锁吗?

global interpreter lock （GIL）通常被视为 Python 在高端多核服务器上开发时的阻力，因为（几乎）所有 Python 代码只有在获取到 GIL 时才能运行，所以多线程的 Python 程序只能有效地使用一个 CPU。

在 **PEP 703** 通过后目前已着手从 Python 的 CPython 实现中移除 GIL。 最初它将它将作为构建解释器时的可选编译器旗标来实现，因此将会存在有 GIL 和没有 GIL 的构建版本。 从长远来看，目标是在移除 GIL 对性能的影响被完全了解之后确定唯一的构建版本。 Python 3.13 大概是第一个包含此项工作的发布版，尽管在这个发布版中该功能可能尚不完整。

当前移除 GIL 的工作是基于 Sam Gross 的 移除了 GIL 的 Python 3.9 分叉。 更早的时候，在 Python 1.5 时期，Greg Stein 实际上实现了一个完整的补丁集（“自由线程”补丁），移除了 GIL 并用更细粒度的锁来代替。 Adam Olsen 也在他的 python-safethread 项目里做了类似的实验。 不幸的是，由于为移除 GIL 而使用了大量细粒度的锁这两个早期实验在单线程中的性能都有明显的下降（至少慢 30%）。 Python 3.9 分叉是在移除 GIL 的同时保持可接受的性能影响的首次尝试。

当前 Python 发布版存在 GIL 并不意味着你无法在多CPU 机器上很好地使用 Python！ 你仅需发挥创造性将任务在多个 *进程* 而不是多个 *threads* 线程之间进行分配。 新的 `concurrent.futures` 模块中的 `ProcessPoolExecutor` 类提供了完成此项工作的简单方式；如果你想要对任务分发有更强的控制那么 `multiprocessing` 模块提供了更低层级的 API。

恰当地使用 C 拓展也很有用；使用 C 拓展处理耗时较久的任务时，拓展可以在线程执行 C 代码时释放 GIL，让其他线程执行。 `zlib` 和 `hashlib` 等标准库模块已经这样做了。

减小 GIL 的影响的一种替代方式是让 GIL 成为每解释器状态锁而不是真正的全局状态锁。 此特性 在 Python 3.12 中首次实现 并在 C API 中可用。 预期会在 Python 3.13 中提供它的 Python 接口。 目前它的主要限制在于第 3 方扩展模块，因此这些模块的编写必须考虑到多解释器的情况才能够被使用，这样大量较旧的扩展模块将不再可用。

输入与输出

怎样删除文件？ （以及其他文件相关的问题……）

使用 `os.remove(filename)` 或 `os.unlink(filename)`。查看 `os` 模块以获取更多文档。这两个函数是一样的，`unlink()` 是这个函数在 Unix 系统调用中的名字。

如果要删除目录，应该使用 `os.rmdir()`；使用 `os.mkdir()` 创建目录。`os.makedirs(path)` 会创建 `path` 中任何不存在的目录。`os.removedirs(path)` 则会删除其中的目录，只要它们都是空的；如果你想删除整个目录以及其中的内容，可以使用 `shutil.rmtree()`。

重命名文件可以使用 `os.rename(old_path, new_path)`。

如果需要截断文件，使用 `f = open(filename, "rb+")` 打开文件，然后使用 `f.truncate(offset)`；offset 默认是当前的搜索位置。也可以对使用 `os.open()` 打开的文件使用 `os.ftruncate(fd, offset)`，其中 *fd* 是文件描述符（一个小的整型数）。

`shutil` 模块也包含了一些处理文件的函数，包括 `copyfile()`，`copytree()` 和 `rmtree()`。

怎样复制文件？

`shutil` 模块包含一个 `copyfile()` 函数。注意，在 Windows NTFS 卷上，它不复制 替代数据流，也不复制 macOS HFS+ 卷上的 资源分叉，尽管这两者现在很少使用。它也不复制文件权限和元数据，尽管使用 `shutil.copy2()` 可以保留大部分（但不是全部）的内容。

怎样读取（或写入）二进制数据？

要读写复杂的二进制数据格式，最好使用 `struct` 模块。该模块可以读取包含二进制数据（通常是数字）的字符串并转换为 Python 对象，反之亦然。

举例来说，下面的代码会从文件中以大端序格式读取一个 2 字节的整型和一个 4 字节的整型：

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

格式字符串中的 '>' 强制以大端序读取数据；字母 'h' 从字符串中读取一个"短整型"（2 字节），字母 'l' 读取一个"长整型"（4 字节）。

对于更常规的数据（例如整型或浮点类型的列表），你也可以使用 `array` 模块。

> **备注：** 要读写二进制数据的话，需要强制以二进制模式打开文件（这里为 `open()` 函数传入 `"rb"`）。如果（默认）传入 `"r"` 的话，文件会以文本模式打开，`f.read()` 会返回 `str` 对象，而不是 `bytes` 对象。

似乎 os.popen() 创建的管道不能使用 os.read()，这是为什么？

`os.read()` 是一个底层函数，它接收的是文件描述符 —— 用小整型数表示的打开的文件。`os.popen()` 创建的是一个高级文件对象，和内建的 `open()` 方法返回的类型一样。因此，如果要从 `os.popen()` 创建的管道 *p* 中读取 *n* 个字节的话，你应该使用 `p.read(n)`。

怎样访问（RS232）串口？

对于 Win32, OSX, Linux, BSD, Jython, IronPython:

pyserial

对于 Unix，查看 Mitch Chapman 发布的帖子：

https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com

为什么关闭 sys.stdout（stdin，stderr）并不会真正关掉它?

Python 文件对象 是一个对底层 C 文件描述符的高层抽象。

对于在 Python 中通过内建的 `open()` 函数创建的多数文件对象来说，`f.close()` 从 Python 的角度将其标记为已关闭，并且会关闭底层的 C 文件描述符。在 `f` 被垃圾回收的时候，析构函数中也会自动处理。

但由于 stdin，stdout 和 stderr 在 C 中的特殊地位，在 Python 中也会对它们做特殊处理。运行 `sys.stdout.close()` 会将 Python 的文件对象标记为已关闭，但是 *不会* 关闭与之关联的文件描述符。

要关闭这三者的 C 文件描述符的话，首先你应该确认确实需要关闭它（比如，这可能会影响到处理 I/O 的拓展）。如果确实需要这么做的话，使用 `os.close()`：

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

或者也可以使用常量 0，1，2 代替。

网络 / Internet 编程

Python 中的 WWW 工具是什么?

参阅代码库参考手册中 互联网协议和支持 和 互联网数据处理 这两章的内容。Python 有大量模块来帮助你构建服务端和客户端 web 系统。

Paul Boddie 维护了一份可用框架的概览，见 https://wiki.python.org/moin/WebProgramming 。

生成 HTML 需要使用什么模块?

你可以在 Web 编程 wiki 页面 找到许多有用的链接。

怎样使用 Python 脚本发送邮件?

使用 `smtplib` 标准库模块。

下面是一个很简单的交互式发送邮件的代码。这个方法适用于任何支持 SMTP 协议的主机。

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs  = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
```

```
    msg += line

# 实际发送的邮件
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

在 Unix 系统中还可以使用 sendmail。sendmail 程序的位置在不同系统中不一样，有时是在 `/usr/lib/sendmail`，有时是在 `/usr/sbin/sendmail`。sendmail 手册页面会对你有所帮助。以下是示例代码：

```
import os

SENDMAIL = "/usr/sbin/sendmail"  # sendmail 的位置
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n")  # 分隔标头和消息体的空白行
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

socket 的 connect() 方法怎样避免阻塞？

通常会用 `select` 模块处理 socket 异步 I/O。

要防止 TCP 连接发生阻塞，你可以将 socket 设为非阻塞模式。 这样当你执行 `connect()` 时，你将要么立即完成连接（不大可能）要么得到一个包含错误编号如 `.errno` 的异常。`errno.EINPROGRESS` 表示连接正在进行中，但尚未完成。 不同的操作系统将返回不同的值，所以你需要检查一下你的系统会返回什么值。

你可以使用 `connect_ex()` 方法来避免创建异常。 它将只返回 errno 值。 要进行轮询，你可以稍后再次调用 `connect_ex()` -- `0` 或 `errno.EISCONN` 表示已经连接 -- 或者你也可以将此 socket 传给 `select.select()` 来检查它只否可写。

> **备注：** `asyncio` 模块提供了通用的单线程并发异步库，它可被用来编写非阻塞的网络代码。 第三方的 Twisted 库是一个热门且功能丰富的替代选择。

数据库

Python 中有数据库包的接口吗？

有的。

标准 Python 还包含了基于磁盘的哈希接口例如 `DBM` 和 `GDBM` 。除此之外还有 `sqlite3` 模块，该模块提供了一个轻量级的基于磁盘的关系型数据库。

大多数关系型数据库都已经支持。查看 数据库编程 wiki 页面 获取更多信息。

在 Python 中如何实现持久化对象?

`pickle` 库模块以一种非常通用的方式解决了这个问题（虽然你依然不能用它保存打开的文件、套接字或窗口之类的东西），此外 `shelve` 库模块可使用 pickle 和 (g)dbm 来创建包含任意 Python 对象的持久化映射。

数学和数字

Python 中怎样生成随机数?

`random` 标准库模块实现了随机数生成器，使用起来非常简单：

```
import random
random.random()
```

这将返回一个 [0, 1) 区间的随机浮点数。

该模块中还有许多其他的专门的生成器，例如：

- `randrange(a, b)` 返回 [a, b) 区间内的一个整型数。
- `uniform(a, b)` 在 [a, b) 区间选择一个浮点数。
- `normalvariate(mean, sdev)` 使用正态（高斯）分布采样。

还有一些高级函数直接对序列进行操作，例如：

- `choice(S)` 从给定的序列中随机选择一个元素。
- `shuffle(L)` 会对列表执行原地重排，即将其随机地打乱。

还有 `Random` 类，你可以将其实例化，用来创建多个独立的随机数生成器。