13.1 通过重定向/管道/文件接受输入¶

问题¶

你希望你的脚本接受任何用户认为最简单的输入方式。包括将命令行的输出通过管道传递给该脚本、 重定向文件到该脚本，或在命令行中传递一个文件名或文件名列表给该脚本。

解决方案¶

Python内置的 `fileinput` 模块让这个变得简单。如果你有一个下面这样的脚本：

```
#!/usr/bin/env python3
import fileinput

with fileinput.input() as f_input:
    for line in f_input:
        print(line, end='')
```

那么你就能以前面提到的所有方式来为此脚本提供输入。假设你将此脚本保存为 `filein.py` 并将其变为可执行文件，那么你可以像下面这样调用它，得到期望的输出：

```
$ ls | ./filein.py          # Prints a directory listing to stdout.
$ ./filein.py /etc/passwd   # Reads /etc/passwd to stdout.
$ ./filein.py < /etc/passwd # Reads /etc/passwd to stdout.
```

讨论¶

`fileinput.input()` 创建并返回一个 `FileInput` 类的实例。 该实例除了拥有一些有用的帮助方法外，它还可被当做一个上下文管理器使用。 因此，整合起来，如果我们要写一个打印多个文件输出的脚本，那么我们需要在输出中包含文件名和行号，如下所示：

```
>>> import fileinput
>>> with fileinput.input('/etc/passwd') as f:
>>>     for line in f:
...         print(f.filename(), f.lineno(), line, end='')
...
/etc/passwd 1 ##
/etc/passwd 2 # User Database
/etc/passwd 3 #

<other output omitted>
```

通过将它作为一个上下文管理器使用，可以确保它不再使用时文件能自动关闭， 而且我们在之后还演示了 `FileInput` 的一些有用的帮助方法来获取输出中的一些其他信息。

13.10 读取配置文件¶

问题¶

怎样读取普通.ini格式的配置文件？

解决方案¶

`configparser` 模块能被用来读取配置文件。例如，假设你有如下的配置文件：

```
; config.ini
; Sample configuration file

[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local

# Setting related to debug configuration
[debug]
log_errors=true
show_warnings=False

[server]
port: 8080
nworkers: 32
pid-file=/tmp/spam.pid
root=/www/root
signature:
    =================================
    Brought to you by the Python Cookbook
    =================================
```

下面是一个读取和提取其中值的例子：

```
>>> from configparser import ConfigParser
>>> cfg = ConfigParser()
>>> cfg.read('config.ini')
['config.ini']
>>> cfg.sections()
['installation', 'debug', 'server']
>>> cfg.get('installation','library')
'/usr/local/lib'
>>> cfg.getboolean('debug','log_errors')

True
>>> cfg.getint('server','port')
8080
>>> cfg.getint('server','nworkers')
32
>>> print(cfg.get('server','signature'))

\=================================
Brought to you by the Python Cookbook
\=================================
>>>
```

如果有需要，你还能修改配置并使用 `cfg.write()` 方法将其写回到文件中。例如：

```
>>> cfg.set('server','port','9000')
>>> cfg.set('debug','log_errors','False')
>>> import sys
>>> cfg.write(sys.stdout)

[installation]
```

```
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local

[debug]
log_errors = False
show_warnings = False

[server]
port = 9000
nworkers = 32
pid-file = /tmp/spam.pid
root = /www/root
signature =
          =================================
          Brought to you by the Python Cookbook
          =================================
>>>
```

讨论¶

配置文件作为一种可读性很好的格式，非常适用于存储程序中的配置数据。 在每个配置文件中，配置数据会被分组（比如例子中的“installation”、 “debug” 和 “server”）。 每个分组在其中指定对应的各个变量值。

对于可实现同样功能的配置文件和Python源文件是有很大的不同的。 首先，配置文件的语法要更自由些，下面的赋值语句是等效的：

```
prefix=/usr/local
prefix: /usr/local
```

配置文件中的名字是不区分大小写的。例如：

```
>>> cfg.get('installation','PREFIX')
'/usr/local'
>>> cfg.get('installation','prefix')
'/usr/local'
>>>
```

在解析值的时候，`getboolean()` 方法查找任何可行的值。例如下面都是等价的：

```
log_errors = true
log_errors = TRUE
log_errors = Yes
log_errors = 1
```

或许配置文件和Python代码最大的不同在于，它并不是从上而下的顺序执行。 文件是安装一个整体被读取的。如果碰到了变量替换，它实际上已经被替换完成了。 例如，在下面这个配置中，`prefix` 变量在使用它的变量之前或之后定义都是可以的：

```
[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local
```

`ConfigParser` 有个容易被忽视的特性是它能一次读取多个配置文件然后合并成一个配置。 例如，假设一个用户像下面这样构造了他们的配置文件：

```
; ~/.config.ini
[installation]
prefix=/Users/beazley/test

[debug]
log_errors=False
```

读取这个文件，它就能跟之前的配置合并起来。如：

```
>>> # Previously read configuration
>>> cfg.get('installation', 'prefix')
'/usr/local'

>>> # Merge in user-specific configuration
>>> import os
>>> cfg.read(os.path.expanduser('~/.config.ini'))
['/Users/beazley/.config.ini']

>>> cfg.get('installation', 'prefix')
'/Users/beazley/test'
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.getboolean('debug', 'log_errors')
False
>>>
```

仔细观察下 `prefix` 变量是怎样覆盖其他相关变量的，比如 `library` 的设定值。 产生这种结果的原因是变量的改写采取的是后发制人策略，以最后一个为准。 你可以像下面这样做试验：

```
>>> cfg.get('installation','library')
'/Users/beazley/test/lib'
>>> cfg.set('installation','prefix','/tmp/dir')
>>> cfg.get('installation','library')
'/tmp/dir/lib'
>>>
```

最后还有很重要一点要注意的是Python并不能支持.ini文件在其他程序（比如windows应用程序）中的所有特性。 确保你已经参阅了configparser文档中的语法详情以及支持特性。

13.11 给简单脚本增加日志功能¶

问题¶

你希望在脚本和程序中将诊断信息写入日志文件。

解决方案¶

打印日志最简单方式是使用 `logging` 模块。例如：

```
import logging

def main():
    # Configure the logging system
    logging.basicConfig(
        filename='app.log',
        level=logging.ERROR
    )

    # Variables (to make the calls that follow work)
    hostname = 'www.python.org'
    item = 'spam'
    filename = 'data.csv'
    mode = 'r'

    # Example logging calls (insert into your program)
    logging.critical('Host %s unknown', hostname)
    logging.error("Couldn't find %r", item)
    logging.warning('Feature is deprecated')
    logging.info('Opening file %r, mode=%r', filename, mode)
    logging.debug('Got here')

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

上面五个日志调用（critical(), error(), warning(), info(), debug()）以降序方式表示不同的严重级别。 `basicConfig()` 的 `level` 参数是一个过滤器。 所有级别低于此级别的日志消息都会被忽略掉。 每个logging操作的参数是一个消息字符串，后面再跟一个或多个参数。 构造最终的日志消息的时候我们使用了%操作符来格式化消息字符串。

运行这个程序后，在文件 `app.log` 中的内容应该是下面这样：

```
CRITICAL:root:Host www.python.org unknown
ERROR:root:Could not find 'spam'
```

如果你想改变输出等级，你可以修改 `basicConfig()` 调用中的参数。例如：

```
logging.basicConfig(
     filename='app.log',
     level=logging.WARNING,
     format='%(levelname)s:%(asctime)s:%(message)s')
```

最后输出变成如下：

```
CRITICAL:2012-11-20 12:27:13,595:Host www.python.org unknown
ERROR:2012-11-20 12:27:13,595:Could not find 'spam'
WARNING:2012-11-20 12:27:13,595:Feature is deprecated
```

上面的日志配置都是硬编码到程序中的。如果你想使用配置文件， 可以像下面这样修改 `basicConfig()` 调用：

```
import logging
import logging.config

def main():
    # Configure the logging system
    logging.config.fileConfig('logconfig.ini')
```

```
    ...
```

创建一个下面这样的文件，名字叫 `logconfig.ini`：

```
[loggers]
keys=root

[handlers]
keys=defaultHandler

[formatters]
keys=defaultFormatter

[logger_root]
level=INFO
handlers=defaultHandler
qualname=root

[handler_defaultHandler]
class=FileHandler
formatter=defaultFormatter
args=('app.log', 'a')

[formatter_defaultFormatter]
format=%(levelname)s:%(name)s:%(message)s
```

如果你想修改配置，可以直接编辑文件logconfig.ini即可。

讨论¶

尽管对于 `logging` 模块而已有很多更高级的配置选项， 不过这里的方案对于简单的程序和脚本已经足够了。 只想在调用日志操作前先执行下basicConfig()函数方法，你的程序就能产生日志输出了。

如果你想要你的日志消息写到标准错误中，而不是日志文件中，调用 `basicConfig()` 时不传文件名参数即可。例如：

```
logging.basicConfig(level=logging.INFO)
```

`basicConfig()` 在程序中只能被执行一次。如果你稍后想改变日志配置， 就需要先获取 `root logger`，然后直接修改它。例如：

```
logging.getLogger().level = logging.DEBUG
```

需要强调的是本节只是演示了 `logging` 模块的一些基本用法。 它可以做更多更高级的定制。 关于日志定制化一个很好的资源是 Logging Cookbook

13.12 给函数库增加日志功能¶

问题¶

你想给某个函数库增加日志功能，但是又不能影响到那些不使用日志功能的程序。

解决方案¶

对于想要执行日志操作的函数库而已，你应该创建一个专属的 `logger` 对象，并且像下面这样初始化配置：

```
# somelib.py

import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

# Example function (for testing)
def func():
    log.critical('A Critical Error!')
    log.debug('A debug message')
```

使用这个配置，默认情况下不会打印日志。例如：

```
>>> import somelib
>>> somelib.func()
>>>
```

不过，如果配置过日志系统，那么日志消息打印就开始生效，例如：

```
>>> import logging
>>> logging.basicConfig()
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
>>>
```

讨论¶

通常来讲，你不应该在函数库代码中自己配置日志系统，或者是已经假定有个已经存在的日志配置了。

调用 `getLogger(__name__)` 创建一个和调用模块同名的logger模块。 由于模块都是唯一的，因此创建的logger也将是唯一的。

`log.addHandler(logging.NullHandler())` 操作将一个空处理器绑定到刚刚已经创建好的logger对象上。 一个空处理器默认会忽略调用所有的日志消息。 因此，如果使用该函数库的时候还没有配置日志，那么将不会有消息或警告出现。

还有一点就是对于各个函数库的日志配置可以是相互独立的，不影响其他库的日志配置。 例如，对于如下的代码：

```
>>> import logging
>>> logging.basicConfig(level=logging.ERROR)

>>> import somelib
>>> somelib.func()
CRITICAL:somelib:A Critical Error!

>>> # Change the logging level for 'somelib' only
>>> logging.getLogger('somelib').level=logging.DEBUG
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
DEBUG:somelib:A debug message
>>>
```

在这里，根日志被配置成仅仅输出ERROR或更高级别的消息。 不过， `somelib` 的日志级别被单独配置成可以输出debug级别的消息，它的优先级比全局配置高。 像这样更改单独模块的日志配置对于调试来讲是很方便的， 因为你无

需去更改任何的全局日志配置——只需要修改你想要更多输出的模块的日志等级。

Logging HOWTO 详细介绍了如何配置日志模块和其他有用技巧，可以参阅下。

13.13 实现一个计时器¶

问题¶

你想记录程序执行多个任务所花费的时间

解决方案¶

`time` 模块包含很多函数来执行跟时间有关的函数。 尽管如此，通常我们会在此基础之上构造一个更高级的接口来模拟一个计时器。例如：

```
import time

class Timer:
    def __init__(self, func=time.perf_counter):
        self.elapsed = 0.0
        self._func = func
        self._start = None

    def start(self):
        if self._start is not None:
            raise RuntimeError('Already started')
        self._start = self._func()

    def stop(self):
        if self._start is None:
            raise RuntimeError('Not started')
        end = self._func()
        self.elapsed += end - self._start
        self._start = None

    def reset(self):
        self.elapsed = 0.0

    @property
    def running(self):
        return self._start is not None

    def __enter__(self):
        self.start()
        return self

    def __exit__(self, *args):
        self.stop()
```

这个类定义了一个可以被用户根据需要启动、停止和重置的计时器。 它会在 `elapsed` 属性中记录整个消耗时间。 下面是一个例子来演示怎样使用它：

```
def countdown(n):
    while n > 0:
        n -= 1

# Use 1: Explicit start/stop
t = Timer()
t.start()
countdown(1000000)
t.stop()
print(t.elapsed)

# Use 2: As a context manager
with t:
    countdown(1000000)

print(t.elapsed)

with Timer() as t2:
```

```
    countdown(1000000)
print(t2.elapsed)
```

讨论¶

本节提供了一个简单而实用的类来实现时间记录以及耗时计算。 同时也是对使用with语句以及上下文管理器协议的一个很好的演示。

在计时中要考虑一个底层的时间函数问题。一般来说， 使用 `time.time()` 或 `time.clock()` 计算的时间精度因操作系统的不同会有所不同。 而使用 `time.perf_counter()` 函数可以确保使用系统上面最精确的计时器。

上述代码中由 `Timer` 类记录的时间是钟表时间，并包含了所有休眠时间。 如果你只想计算该进程所花费的CPU时间，应该使用 `time.process_time()` 来代替：

```
t = Timer(time.process_time)
with t:
    countdown(1000000)
print(t.elapsed)
```

`time.perf_counter()` 和 `time.process_time()` 都会返回小数形式的秒数时间。 实际的时间值没有任何意义，为了得到有意义的结果，你得执行两次函数然后计算它们的差值。

更多关于计时和性能分析的例子请参考14.13小节。

13.14 限制内存和CPU的使用量¶

问题¶

你想对在Unix系统上面运行的程序设置内存或CPU的使用限制。

解决方案¶

`resource` 模块能同时执行这两个任务。例如，要限制CPU时间，可以像下面这样做：

```
import signal
import resource
import os

def time_exceeded(signo, frame):
    print("Time's up!")
    raise SystemExit(1)

def set_max_runtime(seconds):
    # Install the signal handler and set a resource limit
    soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
    resource.setrlimit(resource.RLIMIT_CPU, (seconds, hard))
    signal.signal(signal.SIGXCPU, time_exceeded)

if __name__ == '__main__':
    set_max_runtime(15)
    while True:
        pass
```

程序运行时，`SIGXCPU` 信号在时间过期时被生成，然后执行清理并退出。

要限制内存使用，设置可使用的总内存值即可，如下：

```
import resource

def limit_memory(maxsize):
    soft, hard = resource.getrlimit(resource.RLIMIT_AS)
    resource.setrlimit(resource.RLIMIT_AS, (maxsize, hard))
```

像这样设置了内存限制后，程序运行到没有多余内存时会抛出 `MemoryError` 异常。

讨论¶

在本节例子中，`setrlimit()` 函数被用来设置特定资源上面的软限制和硬限制。 软限制是一个值，当超过这个值的时候操作系统通常会发送一个信号来限制或通知该进程。 硬限制是用来指定软限制能设定的最大值。通常来讲，这个由系统管理员通过设置系统级参数来决定。 尽管硬限制可以改小一点，但是最好不要使用用户进程去修改。

`setrlimit()` 函数还能被用来设置子进程数量、打开文件数以及类似系统资源的限制。 更多详情请参考 `resource` 模块的文档。

需要注意的是本节内容只能适用于Unix系统，并且不保证所有系统都能如期工作。 比如我们在测试的时候，它能在Linux上面正常运行，但是在OS X上却不能。

13.15 启动一个WEB浏览器¶

问题¶

你想通过脚本启动浏览器并打开指定的URL网页

解决方案¶

`webbrowser` 模块能被用来启动一个浏览器，并且与平台无关。例如：

```
>>> import webbrowser
>>> webbrowser.open('http://www.python.org')
True
>>>
```

它会使用默认浏览器打开指定网页。如果你还想对网页打开方式做更多控制，还可以使用下面这些函数：

```
>>> # Open the page in a new browser window
>>> webbrowser.open_new('http://www.python.org')
True
>>>

>>> # Open the page in a new browser tab
>>> webbrowser.open_new_tab('http://www.python.org')
True
>>>
```

这样就可以打开一个新的浏览器窗口或者标签，只要浏览器支持就行。

如果你想指定浏览器类型，可以使用 `webbrowser.get()` 函数来指定某个特定浏览器。例如：

```
>>> c = webbrowser.get('firefox')
>>> c.open('http://www.python.org')
True
>>> c.open_new_tab('http://docs.python.org')
True
>>>
```

对于支持的浏览器名称列表可查阅`Python文档 <http://docs.python.org/3/library/webbrowser.html>`_

讨论¶

在脚本中打开浏览器有时候会很有用。例如，某个脚本执行某个服务器发布任务， 你想快速打开一个浏览器来确保它已经正常运行了。 或者是某个程序以HTML网页格式输出数据，你想打开浏览器查看结果。 不管是上面哪种情况，使用 `webbrowser` 模块都是一个简单实用的解决方案。

13.2 终止程序并给出错误信息¶

问题¶

你想向标准错误打印一条消息并返回某个非零状态码来终止程序运行

解决方案¶

你有一个程序像下面这样终止，抛出一个 `SystemExit` 异常，使用错误消息作为参数。例如：

```
raise SystemExit('It failed!')
```

它会将消息在 `sys.stderr` 中打印，然后程序以状态码1退出。

讨论¶

本节虽然很短小，但是它能解决在写脚本时的一个常见问题。 也就是说，当你想要终止某个程序时，你可能会像下面这样写：

```
import sys
sys.stderr.write('It failed!\n')
raise SystemExit(1)
```

如果你直接将消息作为参数传给 `SystemExit()` ，那么你可以省略其他步骤， 比如import语句或将错误消息写入 `sys.stderr`

13.3 解析命令行选项¶

问题¶

你的程序如何能够解析命令行选项（位于sys.argv中）

解决方案¶

`argparse` 模块可被用来解析命令行选项。下面一个简单例子演示了最基本的用法：

```
# search.py
'''
Hypothetical command-line tool for searching a collection of
files for one or more text patterns.
'''
import argparse
parser = argparse.ArgumentParser(description='Search some files')

parser.add_argument(dest='filenames',metavar='filename', nargs='*')

parser.add_argument('-p', '--pat',metavar='pattern', required=True,
                    dest='patterns', action='append',
                    help='text pattern to search for')

parser.add_argument('-v', dest='verbose', action='store_true',
                    help='verbose mode')

parser.add_argument('-o', dest='outfile', action='store',
                    help='output file')

parser.add_argument('--speed', dest='speed', action='store',
                    choices={'slow','fast'}, default='slow',
                    help='search speed')

args = parser.parse_args()

# Output the collected arguments
print(args.filenames)
print(args.patterns)
print(args.verbose)
print(args.outfile)
print(args.speed)
```

该程序定义了一个如下使用的命令行解析器：

```
bash % python3 search.py -h
usage: search.py [-h] [-p pattern] [-v] [-o OUTFILE] [--speed {slow,fast}]
                 [filename [filename ...]]

Search some files

positional arguments:
  filename

optional arguments:
  -h, --help            show this help message and exit
  -p pattern, --pat pattern
                        text pattern to search for
  -v                    verbose mode
  -o OUTFILE            output file
  --speed {slow,fast}   search speed
```

下面的部分演示了程序中的数据部分。仔细观察print()语句的打印输出。

```
bash % python3 search.py foo.txt bar.txt
usage: search.py [-h] -p pattern [-v] [-o OUTFILE] [--speed {fast,slow}]
```

```
                [filename [filename ...]]
search.py: error: the following arguments are required: -p/--pat

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt
filenames = ['foo.txt', 'bar.txt']
patterns  = ['spam', 'eggs']
verbose   = True
outfile   = None
speed     = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -o results
filenames = ['foo.txt', 'bar.txt']
patterns  = ['spam', 'eggs']
verbose   = True
outfile   = results
speed     = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -o results \
           --speed=fast
filenames = ['foo.txt', 'bar.txt']
patterns  = ['spam', 'eggs']
verbose   = True
outfile   = results
speed     = fast
```

对于选项值的进一步处理由程序来决定，用你自己的逻辑来替代 `print()` 函数。

讨论¶

`argparse` 模块是标准库中最大的模块之一，拥有大量的配置选项。 本节只是演示了其中最基础的一些特性，帮助你入门。

为了解析命令行选项，你首先要创建一个 `ArgumentParser` 实例， 并使用 `add_argument()` 方法声明你想要支持的选项。 在每个 `add_argument()` 调用中，`dest` 参数指定解析结果被指派给属性的名字。 `metavar` 参数被用来生成帮助信息。`action` 参数指定跟属性对应的处理逻辑， 通常的值为 `store` ,被用来存储某个值或将多个参数值收集到一个列表中。 下面的参数收集所有剩余的命令行参数到一个列表中。在本例中它被用来构造一个文件名列表：

```
parser.add_argument(dest='filenames',metavar='filename', nargs='*')
```

下面的参数根据参数是否存在来设置一个 `Boolean` 标志：

```
parser.add_argument('-v', dest='verbose', action='store_true',
                    help='verbose mode')
```

下面的参数接受一个单独值并将其存储为一个字符串：

```
parser.add_argument('-o', dest='outfile', action='store',
                    help='output file')
```

下面的参数说明允许某个参数重复出现多次，并将它们追加到一个列表中去。 `required` 标志表示该参数至少要有一个。`-p` 和 `--pat` 表示两个参数名形式都可使用。

```
parser.add_argument('-p', '--pat',metavar='pattern', required=True,
                    dest='patterns', action='append',
                    help='text pattern to search for')
```

最后，下面的参数说明接受一个值，但是会将其和可能的选择值做比较，以检测其合法性：

```
parser.add_argument('--speed', dest='speed', action='store',
                    choices={'slow','fast'}, default='slow',
                    help='search speed')
```

一旦参数选项被指定，你就可以执行 `parser.parse()` 方法了。 它会处理 `sys.argv` 的值并返回一个结果实例。 每个参数值会被设置成该实例中 `add_argument()` 方法的 `dest` 参数指定的属性值。

还很多种其他方法解析命令行选项。 例如，你可能会手动的处理 `sys.argv` 或者使用 `getopt` 模块。 但是，如果你采用

本节的方式，将会减少很多冗余代码，底层细节 `argparse` 模块已经帮你处理了。 你可能还会碰到使用 `optparse` 库解析选项的代码。 尽管 `optparse` 和 `argparse` 很像，但是后者更先进，因此在新的程序中你应该使用它。

13.4 运行时弹出密码输入提示¶

问题¶

你写了个脚本，运行时需要一个密码。此脚本是交互式的，因此不能将密码在脚本中硬编码， 而是需要弹出一个密码输入提示，让用户自己输入。

解决方案¶

这时候Python的 `getpass` 模块正是你所需要的。你可以让你很轻松的弹出密码输入提示， 并且不会在用户终端回显密码。下面是具体代码：

```
import getpass

user = getpass.getuser()
passwd = getpass.getpass()

if svc_login(user, passwd):    # You must write svc_login()
   print('Yay!')
else:
   print('Boo!')
```

在此代码中，`svc_login()` 是你要实现的处理密码的函数，具体的处理过程你自己决定。

讨论¶

注意在前面代码中 `getpass.getuser()` 不会弹出用户名的输入提示。 它会根据该用户的shell环境或者会依据本地系统的密码库（支持 *pwd* 模块的平台）来使用当前用户的登录名，

如果你想显示的弹出用户名输入提示，使用内置的 `input` 函数：

```
user = input('Enter your username: ')
```

还有一点很重要，有些系统可能不支持 `getpass()` 方法隐藏输入密码。 这种情况下，Python会提前警告你这些问题（例如它会警告你说密码会以明文形式显示）

13.5 获取终端的大小¶

问题¶

你需要知道当前终端的大小以便正确的格式化输出。

解决方案¶

使用 `os.get_terminal_size()` 函数来做到这一点。

代码示例：

```
>>> import os
>>> sz = os.get_terminal_size()
>>> sz
os.terminal_size(columns=80, lines=24)
>>> sz.columns
80
>>> sz.lines
24
>>>
```

讨论¶

有太多方式来得知终端大小了，从读取环境变量到执行底层的 `ioctl()` 函数等等。 不过，为什么要去研究这些复杂的办法而不是仅仅调用一个简单的函数呢？

13.6 执行外部命令并获取它的输出¶

问题¶

你想执行一个外部命令并以Python字符串的形式获取执行结果。

解决方案¶

使用 `subprocess.check_output()` 函数。例如：

```
import subprocess
out_bytes = subprocess.check_output(['netstat','-a'])
```

这段代码执行一个指定的命令并将执行结果以一个字节字符串的形式返回。 如果你需要文本形式返回，加一个解码步骤即可。例如：

```
out_text = out_bytes.decode('utf-8')
```

如果被执行的命令以非零码返回，就会抛出异常。 下面的例子捕获到错误并获取返回码：

```
try:
    out_bytes = subprocess.check_output(['cmd','arg1','arg2'])
except subprocess.CalledProcessError as e:
    out_bytes = e.output       # Output generated before error
    code      = e.returncode   # Return code
```

默认情况下，`check_output()` 仅仅返回输入到标准输出的值。 如果你需要同时收集标准输出和错误输出，使用 `stderr` 参数：

```
out_bytes = subprocess.check_output(['cmd','arg1','arg2'],
                                    stderr=subprocess.STDOUT)
```

如果你需要用一个超时机制来执行命令，使用 `timeout` 参数：

```
try:
    out_bytes = subprocess.check_output(['cmd','arg1','arg2'], timeout=5)
except subprocess.TimeoutExpired as e:
    ...
```

通常来讲，命令的执行不需要使用到底层shell环境（比如sh、bash）。 一个字符串列表会被传递给一个低级系统命令，比如 `os.execve()` 。 如果你想让命令被一个shell执行，传递一个字符串参数，并设置参数 `shell=True` . 有时候你想要Python去执行一个复杂的shell命令的时候这个就很有用了，比如管道流、I/O重定向和其他特性。例如：

```
out_bytes = subprocess.check_output('grep python | wc > out', shell=True)
```

需要注意的是在shell中执行命令会存在一定的安全风险，特别是当参数来自于用户输入时。 这时候可以使用 `shlex.quote()` 函数来将参数正确的用双引用引起来。

讨论¶

使用 `check_output()` 函数是执行外部命令并获取其返回值的最简单方式。 但是，如果你需要对子进程做更复杂的交互，比如给它发送输入，你得采用另外一种方法。 这时候可直接使用 `subprocess.Popen` 类。例如：

```
import subprocess

# Some text to send
text = b'''
hello world
this is a test
goodbye
'''
```

```
# Launch a command with pipes
p = subprocess.Popen(['wc'],
          stdout = subprocess.PIPE,
          stdin = subprocess.PIPE)

# Send the data and get the output
stdout, stderr = p.communicate(text)

# To interpret as text, decode
out = stdout.decode('utf-8')
err = stderr.decode('utf-8')
```

`subprocess` 模块对于依赖TTY的外部命令不合适用。 例如，你不能使用它来自动化一个用户输入密码的任务（比如一个ssh会话）。 这时候，你需要使用到第三方模块了，比如基于著名的 `expect` 家族的工具（pexpect或类似的）

13.7 复制或者移动文件和目录¶

问题¶

你想要复制或移动文件和目录，但是又不想调用shell命令。

解决方案¶

`shutil` 模块有很多便捷的函数可以复制文件和目录。使用起来非常简单，比如：

```
import shutil

# Copy src to dst. (cp src dst)
shutil.copy(src, dst)

# Copy files, but preserve metadata (cp -p src dst)
shutil.copy2(src, dst)

# Copy directory tree (cp -R src dst)
shutil.copytree(src, dst)

# Move src to dst (mv src dst)
shutil.move(src, dst)
```

这些函数的参数都是字符串形式的文件或目录名。 底层语义模拟了类似的Unix命令，如上面的注释部分。

默认情况下，对于符号链接而已这些命令处理的是它指向的东西。 例如，如果源文件是一个符号链接，那么目标文件将会是符号链接指向的文件。 如果你只想复制符号链接本身，那么需要指定关键字参数 `follow_symlinks` ,如下：

如果你想保留被复制目录中的符号链接，像这样做：

```
shutil.copytree(src, dst, symlinks=True)
```

`copytree()` 可以让你在复制过程中选择性的忽略某些文件或目录。 你可以提供一个忽略函数，接受一个目录名和文件名列表作为输入，返回一个忽略的名称列表。例如：

```
def ignore_pyc_files(dirname, filenames):
    return [name in filenames if name.endswith('.pyc')]

shutil.copytree(src, dst, ignore=ignore_pyc_files)
```

由于忽略某种模式的文件名是很常见的，因此一个便捷的函数 `ignore_patterns()` 已经包含在里面了。例如：

```
shutil.copytree(src, dst, ignore=shutil.ignore_patterns('*~', '*.pyc'))
```

讨论¶

使用 `shutil` 复制文件和目录也忒简单了点吧。 不过，对于文件元数据信息，`copy2()` 这样的函数只能尽自己最大能力来保留它。 访问时间、创建时间和权限这些基本信息会被保留， 但是对于所有者、ACLs、资源fork和其他更深层次的文件元信息就说不准了， 这个还得依赖于底层操作系统类型和用户所拥有的访问权限。 你通常不会去使用 `shutil.copytree()` 函数来执行系统备份。 当处理文件名的时候，最好使用 `os.path` 中的函数来确保最大的可移植性（特别是同时要适用于Unix和Windows）。 例如：

```
>>> filename = '/Users/guido/programs/spam.py'
>>> import os.path
>>> os.path.basename(filename)
'spam.py'
>>> os.path.dirname(filename)
'/Users/guido/programs'
>>> os.path.split(filename)
('/Users/guido/programs', 'spam.py')
>>> os.path.join('/new/dir', os.path.basename(filename))
'/new/dir/spam.py'
```

```
>>> os.path.expanduser('~/guido/programs/spam.py')
'/Users/guido/programs/spam.py'
>>>
```

使用 `copytree()` 复制文件夹的一个棘手的问题是对于错误的处理。 例如，在复制过程中，函数可能会碰到损坏的符号链接，因为权限无法访问文件的问题等等。 为了解决这个问题，所有碰到的问题会被收集到一个列表中并打包为一个单独的异常，到了最后再抛出。 下面是一个例子：

```
try:
    shutil.copytree(src, dst)
except shutil.Error as e:
    for src, dst, msg in e.args[0]:
         # src is source name
         # dst is destination name
         # msg is error message from exception
         print(dst, src, msg)
```

如果你提供关键字参数 `ignore_dangling_symlinks=True`， 这时候 `copytree()` 会忽略掉无效符号链接。

本节演示的这些函数都是最常见的。不过，`shutil` 还有更多的和复制数据相关的操作。 它的文档很值得一看，参考 Python documentation

13.8 创建和解压归档文件¶

问题¶

你需要创建或解压常见格式的归档文件（比如.tar, .tgz或.zip）

解决方案¶

`shutil` 模块拥有两个函数——`make_archive()` 和 `unpack_archive()` 可派上用场。 例如：

```
>>> import shutil
>>> shutil.unpack_archive('Python-3.3.0.tgz')

>>> shutil.make_archive('py33','zip','Python-3.3.0')
'/Users/beazley/Downloads/py33.zip'
>>>
```

`make_archive()` 的第二个参数是期望的输出格式。 可以使用 `get_archive_formats()` 获取所有支持的归档格式列表。 例如：

```
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"), ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'), ('zip', 'ZIP file')]
>>>
```

讨论¶

Python还有其他的模块可用来处理多种归档格式（比如tarfile, zipfile, gzip, bz2）的底层细节。 不过，如果你仅仅只是要创建或提取某个归档，就没有必要使用底层库了。 可以直接使用 `shutil` 中的这些高层函数。

这些函数还有很多其他选项，用于日志打印、预检、文件权限等等。 参考 shutil文档

13.9 通过文件名查找文件¶

问题¶

你需要写一个涉及到文件查找操作的脚本，比如对日志归档文件的重命名工具， 你不想在Python脚本中调用shell，或者你要实现一些shell不能做的功能。

解决方案¶

查找文件，可使用 `os.walk()` 函数，传一个顶级目录名给它。 下面是一个例子，查找特定的文件名并答应所有符合条件的文件全路径：

```
#!/usr/bin/env python3.3
import os

def findfile(start, name):
    for relpath, dirs, files in os.walk(start):
        if name in files:
            full_path = os.path.join(start, relpath, name)
            print(os.path.normpath(os.path.abspath(full_path)))

if __name__ == '__main__':
    findfile(sys.argv[1], sys.argv[2])
```

保存脚本为文件findfile.py，然后在命令行中执行它。 指定初始查找目录以及名字作为位置参数，如下：

讨论¶

`os.walk()` 方法为我们遍历目录树， 每次进入一个目录，它会返回一个三元组，包含相对于查找目录的相对路径，一个该目录下的目录名列表， 以及那个目录下面的文件名列表。

对于每个元组，只需检测一下目标文件名是否在文件列表中。如果是就使用 `os.path.join()` 合并路径。 为了避免奇怪的路径名比如 `././foo//bar` ，使用了另外两个函数来修正结果。 第一个是 `os.path.abspath()` ,它接受一个路径，可能是相对路径，最后返回绝对路径。 第二个是 `os.path.normpath()` ，用来返回正常路径，可以解决双斜杆、对目录的多重引用的问题等。

尽管这个脚本相对于UNIX平台上面的很多查找来讲要简单很多，它还有跨平台的优势。 并且，还能很轻松的加入其他的功能。 我们再演示一个例子，下面的函数打印所有最近被修改过的文件：

```
#!/usr/bin/env python3.3

import os
import time

def modified_within(top, seconds):
    now = time.time()
    for path, dirs, files in os.walk(top):
        for name in files:
            fullpath = os.path.join(path, name)
            if os.path.exists(fullpath):
                mtime = os.path.getmtime(fullpath)
                if mtime > (now - seconds):
                    print(fullpath)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: {} dir seconds'.format(sys.argv[0]))
        raise SystemExit(1)

    modified_within(sys.argv[1], float(sys.argv[2]))
```

在此函数的基础之上，使用os,os.path,glob等类似模块，你就能实现更加复杂的操作了。 可参考5.11小节和5.13小节等相

关章节。

第十三章：脚本编程与系统管理¶

许多人使用Python作为一个shell脚本的替代，用来实现常用系统任务的自动化，如文件的操作，系统的配置等。本章的主要目标是描述关于编写脚本时候经常遇到的一些功能。例如，解析命令行选项、获取有用的系统配置数据等等。第5章也包含了与文件和目录相关的一般信息。

Contents: