Python基础教程

第9章 执行环境

9. Python执行环境

- 在当前脚本继续运行
  - 创建和管理子进程
  - 执行外部命令或程序
  - 执行需要输入的命令
- 通过网络来调用命令
- 执行命令来创建需要处理的输出
- 执行其他的Python脚本
- 执行一系列动态生成的Python语句
- 导入Python模块（和执行它顶层的代码）

9.1 可调用对象

Python有 4 种可调用对象:

- 函数
- 方法
- 类
- 类的实例

这些对象的任何引用或者别名都是可调用的。

9.1.1 函数

Python有三种不同类型函数对象:

- 内建函数 （内置函数，无需导入就可直接调用）
- 用户定义的函数 （用户自己编写的函数）
- lambda表达式 (用lambda关键字创建的匿名函数)

9.1.1 函数 - 内建函数(BIF)

内建函数用C/C++编写，编译后放入Python解释器，而后加载进系统。

这些函数在_bulitin_模块里，并作为__builtins__模块导入到解释器中。

表 14.1 内建函数属性

属　性	描　述
bif.__doc__	文档字符串（或 None）
bif.__name__	字符串类型的文档名字
bif.__self__	设置为 None（保留给内建方法）
bif.__module__	存放 bif 定义的模块名字(或 None)

```
>>> dir( type )
['__abstractmethods__', '__base__', '__bases__', '__basicsize__', '__call__', '__class__', '__
delattr__', '__dict__', '__dictoffset__', '__dir__', '__doc__', '__eq__', '__flags__', '__format
__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__inst
ancecheck__', '__itemsize__', '__le__', '__lt__', '__module__', '__mro__', '__name__', '__n
e__', '__new__', '__prepare__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '
__setattr__', '__sizeof__', '__str__', '__subclasscheck__', '__subclasses__', '__subclasshook
__', '__text_signature__', '__weakrefoffset__', 'mro']
>>> type( dir )
<class 'builtin_function_or_method'>
```

9.1.1 函数 - 用户定义的函数(UDF)

用户定义函数通常用Python编写，定义在模块的最高级，函数也可以在其他的函数体内定义。

表 14.2 用户自定义函数属性

属　性	描　述
udf.__doc__	文档字符串（也可以用 *udf*.func_doc)
udf.__name__	字符串类型的函数名字（也可以用 *udf*.func_name）
udf.func_code	字节编译的代码对象
udf.func_defaults	默认的参数元组
udf.func_globals	全局名称空间字典；和从函数内部调用 globals(x)一样
udf.func_dict	函数属性的名称空间
udf.func_doc	（见上面的 *udf*.__doc__）
udf.func_name	（见上面的 *udf*.__name__）
udf.func_closure	包含了自由变量的引用的单元对象元组（自用变量在 UDF 中使用，但在别处定义；参见《Python[语言]参考手册》）

9.1.1 函数 - lambda表达式

lambda表达式是用lambda关键字创建，返回一个函数对象。

lambda表达式不向命名绑定的代码提供基础结构，所以需要通过函数式编程接口，或者把表达式的引用赋值给一个变量来调用。

```
>>> lambdaFunc = lambda x : x * 2
>>> lambdaFunc( 100 )
200
>>> type( lambdaFunc )
<class 'function'>
```

9.1.2 方法 - 内建方法(BIM)

许多Python数据类型也有方法，被称为内建方法。

只有内建类型有内建方法。

表 14.3 内建方法属性

属　性	描　述
*bim.*__doc__	文档字串
*bim.*__name__	字符串类型的函数名字
*bim.*__self__	绑定的对象

```
>>> type( [ ].append )
<class 'builtin_function_or_method'>
>>> dir( [ ].append )
['__call__', '__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '_
_module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex
__', '__repr__', '__self__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__text
_signature__']
```

9.1.2 方法 - 用户自定义的方法(UDM)

用户自定义的方法包含在类定义中。

只是拥有标准函数的包装，仅有定义它们的类可以使用。

子类中如果不覆盖那么也可以使用。

表 14.4

属　性	描　述
udm.__doc__	文档字符串（与 udm.im_fuc.__doc__相同）
udm.__name__	字符串类型的方法名字（与 umd.im_func.__name__相同）
udm.__module__	定义 udm 的模块的名字(或 none)
udm.im_class	方法相关联的类（对于绑定的方法；如果是非绑定，那么为要求 udm 的类）
udm.im_func	方法的函数对象（见 UDF）
udm.im_self	如果绑定的话为相关联的实例，如果非绑定位为 none

9.1.2 方法 - 用户自定义的方法(UDM)

```
>>> class C( object ):
        def foo( self ):
            pass


>>> c = C( )
>>> type( C )
<class 'type'>
>>> type( c )
<class '__main__.C'>
>>> type( C.foo )
<class 'function'>
>>> type( c.foo )
<class 'method'>
>>> C.foo
<function C.foo at 0x03D66A98>
>>> c.foo
<bound method C.foo of <__main__.C object at 0x041D9DB0>>
>>> c
<__main__.C object at 0x041D9DB0>
```

9.1.3 类和类的实例

可调用对象：任何能通过函数操作符“( )”来调用的对象。

Python有4种可调用对象：函数，方法，类，实现了__call__( )方法的类的实例。

默认情况下__call__方法是没有实现的，只有定义类的时候实现了__call__方法，类的实例才能成为可调用的。

9.1.3 类和类的实例

```
>>> class C( object ):
        def __call__( self, *args ):
            print( 'I am callable! Called with args: \n', args)


>>> c = C( )
>>> c
<__main__.C object at 0x033070B0>
>>> callable( c )
True
>>> c( )
I am callable! Called with args:
 ()
>>> c( 3 )
I am callable! Called with args:
 (3,)
>>> c(3, 'no more, no less')
I am callable! Called with args:
 (3, 'no more, no less')
```

9.2 代码对象

可调用对象是Python执行环境里最重要的部分。

每个可调用物的核心都是代码对象，由语句、赋值、表达式和其他可调用物组成。

代码对象的一个重要属性就是字节码（由源代码编译得到一组虚拟机指令）。

代码对象可作为函数或者方法调用的一部分来执行，也可用内建函数exec( )或者eval( )来执行。

```
>>> def foo( a ):
        return a + 3

>>> foo              #函数对象
<function foo at 0x0376C4F8>
>>> foo.__code__          #代码对象
<code object foo at 0x03770CD8, file "<pyshell#12>", line 1>
>>> foo.__code__.co_varnames
('a',)
>>> foo.__code__.co_code          #字节码
b'|\x00d\x01\x17\x00S\x00'
>>> [i for i in foo.__code__.co_code]
[124, 0, 100, 1, 23, 0, 83, 0]
>>> import dis    #加载反汇编模块
>>> dis.dis( foo.__code__ )
  2           0 LOAD_FAST                0 (a)
              2 LOAD_CONST               1 (3)
              4 BINARY_ADD
              6 RETURN_VALUE
>>>
```

9.3 可执行的对象声明和内建函数

Python提供了大量的BIF来支持可调用/可执行对象。

- callable()
- eval()
- exec()
- compile()

callable()

callable( )确定一个对象是否可以通过函数操作符 “**( )**” 来调用。可调用返回**True**，否则返回**False**。

```
>>> callable( dir )          #内建函数
True
>>> callable( 1 )            #整形
False
>>> def foo( ): pass

>>> callable( foo )          #用户自定义函数
True
>>> callable( 'bar' )        #字符串
False
>>> class C( object ): pass

>>> callable( C )            #类
True
```

eval( )

eval( )接收参数字符串并把它作为**Python**表达式进行求值。

int( )接收代表整形字符串并把它转化为整型：

```
>>> eval( '932' )
932
>>> int( '932' )
932
>>> eval( '100 + 200' )
300
>>> int( '100 + 200' )
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    int( '100 + 200' )
ValueError: invalid literal for int() with base 10: '100 + 200'
```

exec( )

exec( )语句执行代码对象或字符串形式的Python代码，且可以接受有效的Python文件对象：

```
>>> exec("""
x = 0
print(' x is currently:', x )
while x < 5:
        x += 1
        print(' incrementing x to:', x )
""" )
 x is currently: 0
 incrementing x to: 1
 incrementing x to: 2
 incrementing x to: 3
 incrementing x to: 4
 incrementing x to: 5
```

```
>>> f = open('xcount.py')
>>> exec( f.read( ) )
 x is currently: 0
 incrementing x to: 1
 incrementing x to: 2
 incrementing x to: 3
 incrementing x to: 4
 incrementing x to: 5
>>> f.tell( )
96
>>> f.seek( 0, 0 )
0
```

exec读取文件数据后会停留在文件末尾，若需再次执行需要用seek( )定位。

compile( )

返回一个代码对象，该对象可以传递给内建函数exec( )或eval( )来执行。compile( )函数的第三个参数表明代码对象的类型，有3个可能值：

- 'eval' 可求值的表达式
- 'single' 单一可执行语句
- 'exec' 可执行语句组

compile( )

```
>>> ecode = compile('100 + 200', ' ', 'eval')
>>> eval( ecode )
300
>>> ecode
<code object <module> at 0x039B4DE0, file " ", line 1>
>>> str1 = "print('Hello World')"
>>> str2 = '''for i in range(0, 5):
                    print( i )'''
>>> exec(compile(str1, ' ', 'single'))
Hello World
>>> exec(compile(str2, ' ','exec' ))
0
1
2
3
4
```

9.4 执行其他程序

在执行其他程序时，可以将它们分类为：

（1）Python程序

（2）其他非Python程序

9.4.1 导入

第一次导入模块会执行模块最高级的代码，只有属于模块最高级的代码才是全局变量、全局类和全局函数声明。

一旦导入Python模块后，就会执行该模块。

处理不想每次导入都执行的代码，应缩进它，并放入 **if__name__== '__main__'** 的内部。

9.4.1 导入

```
# import1.py
print( 'loaded import1' )
import import2

#这里是import2.py的内容:
#import2.py
print('loaded import2')
```

这是当我们导入import1时的输出:

```
>>>import import1
loaded import1
loaded import2
>>>
```

9.4.1 导入

检测 __name__ 值的迂回工作法:

```
# import1.py
import import2
if __name__ == '__main__':
    print( 'loaded import1' )
```

import1.py的内容

```
#import2.py
if __name__ == '__main__':
    print( 'loaded import2' )
```

import2.py的内容

输出:

```
>>>import import1
>>>
```

9.4.2 将模块作为脚本执行

Python允许从shell或DOS提示符，直接将模块作为脚本来运行，可以使用命令行从工作目录调用脚本:

```
$ python script.py          # Unix/Linux
```

或者

```
C:>python script.py         # Windows/DOS
```

9.4.2 将模块作为脚本执行

Python常用命令行选项：

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。

9.5 执行其他(非Python)程序

只要执行环境是有效的，就可以在Python程序里执行非Python程序。

针对不同的环境，Python为外部程序执行提供了各种os模块。

表 14.6　为外部程序执行提供的 os 模块（Ⓤ代表 Unix 下，Ⓦ代表 Windows 下）

模块函数	描　述
system(*cmd*)	执行程序 cmd（字符串），等待程序结束，返回退出代码（windows 下，始终为 0）
fork()	创建一个和父进程并行的子进程（通常来说和 exec*()一起使用）；返回两次....一次给父进程一次给子进程 U
execl(*file*, *arg0*, *arg1*,...)	用参数列表 arg0、arg1 等执行文件
execv(*file*, *arglist*)	除了使用参数向量列表，其他的和 execl()相同
execle(*file*, *arg0*, *arg1*,... *env*)	和 execl 相同，但提供了环境变量字典 env
execve(*file*, *arglist*, *env*)	除了带有参数向量列表，其他的和 execle()相同
execlp(*cmd*, *arg0*, *rarg1*,...)	与 execl()相同，但是在用户的搜索路径下搜索完全的文件路径名
execvp(*cmd*, *arglist*)	除了带有参数向量列表，与 execlp()相同
execlpe(*cmd*, *arg0*, *arg1*,... *env*)	和 execlp 相同，但提供了环境变量字典 env
execvpe(*cmd*, *arglist*, *env*)	和 execvp 相同，但提供了环境变量字典 env
spawn*[a](mode, file, args[, env])	spawn*()家族在一个新的进程中执行路径，args 作为参数，也许还有环境变量的字典 env；模式（mode）是个显示不同操作模式的魔术
wait()	等待子进程完成(通常和 fock 和 exec*()一起使用)Ⓤ
waitpid(*pid*, *options*)	等待指定的子进程完成[通常和 fock 和 exec*()一起使用]Ⓤ
popen(*cmd*, *mode*='r', *buffering*=-1)	执行字符串 cmd，返回一个类文件对象作为运行程序通信句柄，默认为读取模式和默认系统缓冲 startfileb(path)
startfile[b](path)	用关联的应用程序执行路径Ⓦ

9.5.1 os.system( )

system( )函数接收字符串形式的系统命令并执行它。

执行完成后，将会以system( )的返回值形式给出退出状态，Python的执行也会继续。

```
>>> import os
>>> os.system( 'dir' )
0
>>> os.system( 'notepad' )
0
```

9.5.2 os.popen()

os.system是简单的执行shell命令，但不能获取shell命令执行输出的内容，如果需要就需使用os.popen。

在system( )的基础上结合文件对象，popen( )建立一个指向程序的单向连接，使用完毕以后，应当用close( )关闭。

9.5.2 os.popen()

```
>>> a = os.popen('dir')
>>> d = a.read( )
>>> d
' 驱动器 C 中的卷没有标签。\n 卷的序列号是 7691-E82F\n\n C:\\Users\\reebox\\AppData\\Local\\Programs\\Python\\Python37-32 的目录\n\n2019/12/10  22:54    <DIR>          .\n2019/12/10  22:54    <DIR>          ..\n2019/11/11  19:25               199 7-2.py\n2019/11/11  19:31               310 7-3.py\n2019/11/11  19:52               122 7-5.py\n2019/09/03  18:45    <DIR>          DLLs\n2019/09/03  18:45    <DIR>          Doc\n2019/09/03  18:44    <DIR>          include\n2019/09/03  18:45    <DIR>          Lib\n2019/09/03  18:45    <DIR>          libs\n2019/07/08  19:33            30,188 LICENSE.txt\n2019/11/11  20:18               114 Mymodule.py\n2019/07/08  19:33           692,078 NEWS.txt\n2019/07/08  19:31            97,296 python.exe\n2019/07/08  19:30            58,896 python3.dll\n2019/07/08  19:29         3,606,032 python37.dll\n2019/07/08  19:31            95,760 pythonw.exe\n2019/09/03  18:45    <DIR>          Scripts\n2019/09/03  18:45    <DIR>          tcl\n2019/09/03  18:45    <DIR>          Tools\n2019/07/08  19:24            86,840 vcruntime140.dll\n2019/12/10  22:54                96 xcount.py\n2019/11/11  20:20    <DIR>          __pycache__\n              12 个文件      4,667,931 字节\n              11 个目录 48,897,646,592 可用字节\n'
>>> a.close( )
```

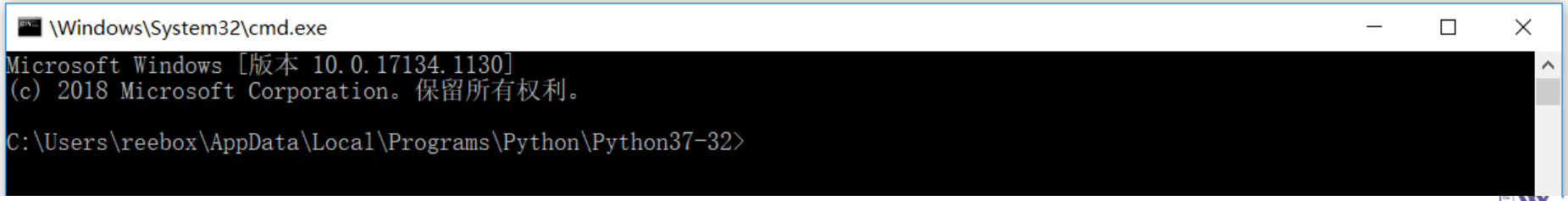
9.5.4 os.spawn*()

os.spawnv( ) 也是 **os** 模块提供的一个执行命令的内置功能函数。

不同于 **os.system( )** 只能执行 **shell** 命令，**os.spawnv( )** 可以执行任何“可执行”文件，包括C编译后的可执行文件，以及**Python**可以执行文件，当然也包括了**shell** 命令。

os.spawnv(mode, file, args)

```
>>> import os
>>> os.spawnv( os.P_WAIT, '\Windows\System32\cmd.exe', ['netstat', '-an'] )
```



9.5.4 os.spawn*()

os.spawnl(mode, path, ...)

os.spawnle(mode, path, ..., env)

os.spawnlp(mode, file, ...)

os.spawnlpe(mode, file, ..., env)

os.spawnv(mode, path, args)

os.spawnve(mode, path, args, env)

os.spawnvp(mode, file, args)

os.spawnvpe(mode, file, args, env)

在新进程中执行程序**path**。

9.6 subprocess模块

subprocess模块在Python 3.5中添加run()函数，用以替换os模块中的相应函数。

from subprocess import run

1）run('cmd.exe')

2）run('dir',shell=True)

3）运行命令并获得输出

a = run('dir',shell=True,capture_output=True,encoding='gbk')

a. stdout

```
>>> from subprocess import run
>>> run('cmd.exe')
CompletedProcess(args='cmd.exe', returncode=3221225786)
>>> run('dir', shell = True)
CompletedProcess(args='dir', returncode=0)
>>> a=run('dir',shell=True,capture_output=True,encoding='gbk')
```

9.7 结束执行

干净的执行表示当所有模块最高级的语句执行完毕后的退出，当遇到某种致命的错误或不满足继续执行的条件的时候，Python会提前退出。

可以通过异常和异常处理，或者建造一个“清扫器”方法，把代码的主要部分放在if语句中，在没有错误的情况下执行，因而可以让错误的情况“正常地”终结。

9.7.1 sys.exit() and SystemExit

当调用sys.exit( )时，会引发systemExit( )异常。

除非对异常进行监控，异常通常不会被捕捉到或处理的，解释器会用给定的状态参数退出，默认为0，exit( )的任何整型参数都会以退出状态返回给调用。

sys.exit( )经常用在命令调用的中途发现错误之后。

9.7.2 sys.exitfunc( )

sys.exitfunc( )默认是不可用的，可以修改它以提供额外的功能。

在调用exit( )退出解释器之前用到这个函数:

（1）如果sys.exitfunc已经被先前定义的exit函数覆盖了，则把这段代码作为exit( )函数的一部分执行。

（2）通常exit函数用于执行某些类型的关闭活动，比如关闭文件和网络连接，最好用于完成维护任务，比如释放先前保留的系统资源。

9.7.2 sys.exitfunc( )

```
import sys
prev_exit_func = getattr(sys, 'exitfunc', None)
def my_exit_func(old_exit = prev_exit_func):
        # ...
        # 进行清理
        # ...
        if old_exit is not None and callable(old_exit):
                old_exit( )

        sys.exitfunc = my_exit_func
```

9.7.3 os._exit( )函数

os模块中的_exit( )函数不是在一般应用中使用（平台相关，只是用特定的平台，比如基于Unix的平台，以及Win32平台）。

语法是os._exit(status)，不执行任何清理便立即退出Python，状态参数是必需的。

9.7.4 os.kill() Function

os模块的kill( )函数模拟传统的Unix函数来发送信号给进程。

kill( )参数是进程标识数（PID）和想要发送的进程的信号。

发送的典型信号为SIGINT、SIGQUIT、或更彻底地SIGKIL来使进程终结。

9.7.4 os.kill() Windows

```
1 import os
2 if __name__ == "__main__":
3 pid = 620
4 os.popen('taskkill.exe /pid:'+str(pid))
```

9.8 各种操作系统接口

表 14.8 各种 OS 模块属性（Ⓦ也适用于 win32）

模 块 属 性	描 述
uname()	获得系统信息（主机名、操作系统版本、补丁级别、系统构架等）
getuid()/setuid(*uid*)	获取/设置现在进程的真正的用户 ID
getpid()/getppid()	获取真正的现在/父进程 ID（PID）Ⓦ
getgid()/setgid(*gid*)	获取/设置现在进程的群组 ID
getsid()/setsid()	获取会话 ID（SID）或创建和返回新的 SID
umask(mask)	设置现在的数字 unmask，同时返回先前的那个（mask 用于文件许可）Ⓦ
getenv(*ev*)/ putenv(*ev*, *value*), environ	获取和设置 环境变量 ev 的值；os.envion 属性是描述当前所有环境变量的字典Ⓦ
geteuid()/setegid()	获取/设置当前进程的有效用户 ID（GID）
getegid()/setegid()	获取/设置当前进程的有效组 ID（GID）
getpgid(*pid*)/ setpgid(*pid*, *pgrp*)	获取和设置进程 GID 进程 PID；对于 get，如果 pid 为 0，便返回现在进程的进程 GID
getlogin()	返回运行现在进程的用户登录
times()	返回各种进程时期的元组Ⓦ
strerror(*code*)	返回和错误代码对应的错误信息Ⓦ
getloadavg()[a]	返回代表在过去 1，5，15 分钟内的系统平均负载值的元组

a. Python2.3 时加入。

9.9 相关模块

表 14.9 执行环境相关模块

模　块	描　述
atexit[a]	注册当 Python 解释器退出时的执行句柄
popen2	提供额外的在 os.popen 之上的功能：提供通过标准文件和其他的进程交互的能力；对于 Python2.4 和更新的版本，使用 subpross()
commands	提供额外的在 os.system 之上的功能：把所有的程序输出保存在返回的字符串中（与输出到屏幕的相反）；对于 Python2.4 和更新的版本，使用 subpross
getopt	在这样的应用程序中的处理选项和命令行参数
site	处理 site-specific 模块或包
platform[b]	底层平台和架构的属性
subprocess[c]	管理（计划替代旧的函数和模块，比如 os.system()、os.spawn*()、os.popen*()、 popen2.*和 command.*）

a. Python2.0 时加入。
b. Python2.3 时加入。
c. Python2.4 时加入。

思考

理解常用函数执行的用法