**全栈Python 编程必备**

据说：

2019年， 浙江信息技术高考可以考python了；

2018年， Python 进入了小学生的教材；

2018年， 全国计算机等级考试，可以考python 了；

据外媒报道，微软正考虑添加 Python 为官方的一种 Excel 脚本语言

……

Python作为一种编程语言，被称为“胶水语言”，更被拥趸们誉为“最美丽”的编程语言，从云端到客户端，再到物联网终端，无所不在，同时还是人工智能优选的编程语言。

因此，从全栈的角度看， Python 是一门必备的语言，因为它是除了驱动和操作系统外，其他都可以做好。

不积跬步无以至千里，不积小流无以成江海。—— 荀子《劝学》

**语法**

Python使用空格或制表符缩进的方式分隔代码，Python 2 仅有31个保留字，而且没有分号、begin、end等标记。

|  |
| --- |
| >>> help("keywords")Here is a list of the Python keywords. Enter any keyword to get more help.and elif if printas else import raiseassert except in returnbreak exec is tryclass finally lambda whilecontinue for not withdef from or yielddel global pass >>> |

可以组织成打油诗， 更方便记忆：

Global is class，def not pass。

if eilf else， del as break。

raise in while，import from yield，

try for print，return and assert。

exec except with lambda,

finally or continue……

python中没有提供定义常量的保留字，可以自己定义一个常量类来实现常量的功能。python中有3种表示字符串类型的方式，即单引号、双引号、三引号。单引号和双引号的作用是相同的，python程序员更喜欢用单引号，C/Java程序员则习惯使用双引号表示字符串。三引号中可以输入单引号、双引号或换行等字符。python不支持自增运算符和自减运算符，其他运算符和表达式都是类似的，尤其是分支判断和循环。

Python的文件类型分为3种，即源代码、字节代码和优化代码。这些都可以直接运行，不需要进行预编译或连接。

**数据类型**

Python中的基本数据类型有布尔类型，整数，浮点数和字符串等。

Python 中的数据结构主要有元组（tuple），列表（list）和字典（dictionary）。元组、列表和字符串都属于序列,是具有索引和切片能力的集合。

元组初始化后不可修改，是写保护的。元组往往代表一行数据，而元组中的元素代表不同的数据项，可以把元组看做不可修改的数组。

|  |
| --- |
| tuple\_name=(“you”,”me”,”him”,”her”) |

列表可转换为元组，是传统意义上的数组，可以实现添加、删除和查找操作，元素的值可以被修改。

|  |
| --- |
| list\_name=[“you”,”me”,”him”,”her”] |

字典是键值对,相对于哈希表。

|  |
| --- |
| dict\_name={“y”:”you”, “m”:”me”, “hi”:”him”, “he”:”her”} |

列表推导（List Comprehensions）是构建列表的快捷方式, 可读性较好且效率更高. 运用列表生成式，可以快速生成list，例如 得到当前目录下的所有目录和文件：

|  |
| --- |
| >>> import os>>> [d for d in os.listdir('.')] |

也可以通过一个list推导出另一个list，代码简洁，例如 将一个列表中的元素都变成小写：

|  |
| --- |
| >>> L = ['Hello', 'World', 'IBM', 'Apple'] >>> [s.lower() for s in L] |

通过这些基本类型，可以组成更有针对性需求的数据结构，例如字典嵌套形成的树等， 针对更复杂的数据结构， Python 中提供了大量的库。

**类与继承**

python用class来定义一个类，当所需的数据结构不能用简单类型来表示时，就需要定义类，然后利用定义的类创建对象。当一个对象被创建后，包含了三方面的特性，即对象的句柄、属性和方法。创建对象的方法：

|  |
| --- |
| abel = Abel()Abel.do() |

类的方法同样分为公有方法和私有方法。私有函数不能被该类之外的函数调用，私有的方法也不能被外部的类或函数调用。python使用函数”staticmethod()“或”@ staticmethod“的方法把普通的函数转换为静态方法，相当于全局函数。python的构造函数名为**init**，析构函数名为**del**。继承的使用方法：

|  |
| --- |
| class AbelApp(abel): 　　 def … |

Python 中的变量名解析遵循LEGB原则，本地作用域（Local），上一层结构中的def或Lambda的本地作用域（Enclosing），全局作用域（Global），内置作用域（Builtin），按顺序查找。

和变量解析不同，Python 会按照特定的顺序遍历继承树，就是方法解析顺序（Method Resolution Order，MRO）。类都有一个名为mro 的属性，值是一个元组，按照方法解析顺序列出各个超类，从当前类一直向上，直到 object 类。

Python 中有一种特殊的类是元类（metaclass）。元类是由“type”衍生而出，所以父类需要传入type，元类的操作都在 **new**中完成。通过元类创建的类，第一个参数是父类，第二个参数是metaclass。

**包与模块**

python程序由包(package)、模块(module)和函数组成。包是由一系列模块组成的集合。包必须含有一个**init**.py文件，它用于标识当前文件夹是一个包。

模块是处理某一类问题的函数和类的集合。模块把一组相关的函数或代码组织到一个文件中，一个文件即是一个模块。模块由代码、函数和类组成。导入模块使用import语句，不过模块不限于此，还可以被 import 语句导入的模块共有以下四类:

* 使用Python写的程序( .py文件)
* C或C++扩展(已编译为共享库或DLL文件)
* 包(包含多个模块)
* 内建模块(使用C编写并已链接到Python解释器内)

Python 提供内建函数\_\_import\_\_动态加载 module,import 本质上是调用 \_\_import\_\_加载 module 的， 函数原型如下：

|  |
| --- |
| \_\_import\_\_(name, globals={}, locals={}, fromlist=[], level=-1) |

例如，加载名为 abel的目录下所有模块：

|  |
| --- |
| def loadModules():res = {}import oslst = os.listdir("abel")dir = []for d in lst:s = os.path.abspath("abel") + os.sep + dif os.path.isdir(s) and os.path.exists(s + os.sep + "\_\_init\_\_.py"):dir.append(d)# load the modulesfor d in dir:res[d] = \_\_import\_\_("abel." + d, fromlist = ["\*"])return res |

需要注意的是，如果输入的参数如果带有 “.”，采用 \_\_import\_\_直接导入 module 容易造成意想不到的结果。 OpenStack 的 oslo.utils 封装了 \_\_import\_\_，支持动态导入 class, object 等。

**命名规范**

Python 中的naming convention 以及 coding standard 有很多好的实践，例如Google 的Python 编程规范等。 就命名规范而言， 可以参见Python之父Guido推荐的规范，见下表：



**迭代器**

迭代是数据处理的基础， 采用一种惰性获取数据的方式, 即按需一次获取一个数据，这就是迭代器模式. 迭代器是一个带状态的对象，检查一个对象 a 是否是迭代对象, 最准确的方法是调用 iter(a) , 如果不可迭代, 则抛出 TypeError 异常.

标准的迭代器接口有两个方法:

* \_\_next\_\_: 返回下一个可用元素, 如没有, 抛出StopIteration 异常.
* \_\_iter\_\_: 返回self , 以便在应该使用可迭代对象的地方使用迭代器.

可迭代对象一定不能是自身的迭代器. 也就是说, 可迭代对象必须实现 \_\_iter\_\_方法, 但不能实现 \_\_next\_\_ 方法.

实现一个斐波那契数列的迭代器例子如下：

|  |
| --- |
| class Fibonacci:def \_\_init\_\_(self):self.prevous = 0self.current = 1def \_\_iter\_\_(self):return selfdef \_\_next\_\_(self):value = self.currentself.current = self.prevous + self.currentself.prevous = valuereturn value |

迭代器就是实现了工厂模式的对象，有很多关于迭代器的例子，比如itertools函数返回的都是迭代器对象。

**生成器**

生成器算得上是Python中最吸引人的特性之一，生成器其实是一种特殊的迭代器，但不需要写\_\_iter\_\_()和\_\_next\_\_()方法了，只需要一个yiled关键字即可。python中的 yield 关键字, 用于构建生成器(generator), 其作用与迭代器一样. 还以斐波那契数列为例：

|  |
| --- |
| def Fibonacci():prevous, current = 0, 1while True:yield currentprevous, current = current, current + prevous |

所有的生成器都是迭代器, 都实现了迭代器的接口。 一般地，只要python函数的定义体中使用了 yield 关键字, 该函数就是生成器函数. 调用生成器函数时, 会返回一个生成器对象。也就是说, 生成器函数是生成器工厂。

生成器函数会创建一个生成器对象, 包装生成器函数的定义体. 把生成器传给 next(…) 函数时, 生成器函数会向前执行函数体中下一个 yield 语句, 返回产出的值, 并在函数定义体的当前位置暂停.



需要注意的是， 在协程中, yield 通常出现在表达式的右边(data = yield), 可以产出值, 也可以不产出(如果yield后面没有表达式, 那么会出None)。 协程可能会从调用方接收数据, 调用方把数据提供给协程使用 通过的是 .send(data) 方法. 而不是 next(…) . 通常, 调用方会把值推送给协程.

生成器调用方是一直获取数据, 而协程调用方可以向它传入数据, 协程也不一定要产出数据。不管数据如何流动, yield 都是一种流程控制工具, 使用它可以实现写作式多任务即，协程可以把控制器让步给中心调度程序, 从而激活其他的协程.

**描述符**

描述符是一种创建托管属性的方法，托管属性还可用于保护属性不受修改，或自动更新某个依赖属性的值。描述符是一种在多个属性上重复利用同一个存取逻辑的方式，能劫持那些本应对于self.\_\_dict\_\_的操作。在其他编程语言中，描述符被称作 setter 和 getter，用于获得 (Get) 和设置 (Set) 一个私有变量。Python 没有私有变量的概念，而描述符可以作为一种 Python 的方式来实现与私有变量类似的功能。

静态方法、类方法、property都是构建描述符的类。创建描述符的方式主要有3种：

1.创建一个类并覆盖任意一个描述符方法：\_\_set\_\_、\_\_ get\_\_ 和 \_\_delete\_\_。当需要某个描述符跨多个不同的类和属性的时候，例如类型验证，则使用该方法，例如：

|  |
| --- |
| class MyNameDescriptor(object):def \_\_init\_\_(self):self.\_myname = ''def \_\_get\_\_(self, instance, owner): return self.\_mynamedef \_\_set\_\_(self, instance, myname):self.\_myname = myname.getText()def \_\_delete\_\_(self, instance):del self.\_myname |

2.使用属性类型可以更加简单、灵活地创建描述符。通过使用 property()，可以轻松地为任意属性创建可用的描述符。

|  |
| --- |
| class Student(object):def \_\_init\_\_(self):self.\_sname = ''def fget(self):return self.\_snamedef fset(self, value):self.\_sname = value.title()def fdel(self):del self.\_snamename = property(fget, fset, fdel, "This is the property.") |

3.使用属性描述符，它结合了属性类型方法和 Python装饰器。

|  |
| --- |
| class Student(object):def \_\_init\_\_(self):self.\_sname = ''@propertydef name(self):return self.\_sname@name.setterdef name(self, value):self.\_sname = value.title()@name.deleterdef name(self):del self.\_sname |

另外，还可以在运行时动态创建描述符。 描述符有很多经典的应用，例如Protobuf。

**装饰器**

装饰器(Decorator)是可调用的对象, 其参数是另一个函数(被装饰的函数). 装饰器可能会处理被装饰的函数, 然后把它返回, 或者将其替换成另一个函数或可调用对象.实际上装饰器就是一个高阶函数，它接收一个函数作为参数，然后返回一个新函数。

装饰器有两大特征:

* 把被装饰的函数替换成其他函数
* 装饰器在加载模块时立即执行

python内置了三个用于装饰方法的函数: property、classmethod 和 staticmethod. 当装饰器不关心被装饰函数的参数，或是被装饰函数的参数多种多样的时候，可变参数非常适合使用。

如果一个函数被多个装饰器修饰，其实应该是该函数先被最里面的装饰器修饰，变成另一个函数后，再次被装饰器修饰。例如：

|  |
| --- |
| def second(func):print "running 2nd decorator"def wrapper():func()return wrapperdef fisrt(func):print "running 1st decorator"def wrapper():func()return wrapper@second@firstdef myfunction():print "running myfunction" |

就扩展功能而言，装饰器模式比子类化更加灵活。

在设计模式中，具体的装饰器实例要包装具体组件的实例，即装饰器和所装饰的组件接口一致，对使用该组件的客户端透明，并将客户端的请求转发给该组件，并且可能在转发前后执行一些额外的操作，透明性使得可以递归嵌套多个装饰器，从而可以添加任意多个功能。装饰器模式和Python装饰器之间并不是一对一的等价关系，Python装饰器函数更为强大，不仅仅可以实现装饰器模式。

**Lambda**

Python 不是纯萃的函数式编程语言，但本身提供了一些函数式编程的特性，像 map、reduce、filter等都支持函数作为参数，lambda 函数函数则是函数式编程中的翘楚。

Lambda 函数又称匿名函数，在某种意义上，return语句隐含在lambda中。和其他很多语言相比，Python 的 lambda 限制很多，最严重的是它只能由一条表达式组成。lambda规范必须包含只有一个表达式，表达式必须返回一个值，由lambda创建一个匿名函数隐式地返回表达式的返回值。

在PySpark 中经常会用到使用Lambda 的操作，例如：

|  |
| --- |
| li = [1, 2, 3, 4, 5]### 列表中国年的每个元素加5map(lambda x: x+5, li) ### 返回其中的偶数filter(lambda x: x % 2 == 0, li) # [2, 4]### 返回所有元素的乘积reduce(lambda x, y: x \* y, li) |

lambda 可以接收任意多个参数 (包括可选参数) 并且返回单个表达式的值。

本质上，Lambda 函数是一个只与输入参数有关的抽象代码树片段。在很多语言里，lambda 函数的调用会被套上一层接口，还会形成闭包，在 lambda 函数构造的同时就可以完成，之后 lambda 函数内部就是完全静态的。而一般的函数还要加上存储局部变量的区域，对外部环境的操作，以及命名，大部分语言强制了一般函数必须与名字绑定。

**线程**

python是支持多线程的, python的线程就是C语言的一个pthread，并通过操作系统调度算法进行调度。 python 的thread模块是轻量级的，而threading模块是对thread做了一些封装，方便使用。threading 经常和Queue结合使用,Queue模块中提供了同步的、线程安全的队列类，包括FIFO队列，LIFO队列，和优先级队列等。这些队列都实现了锁，能够在多线程中直接使用，可以使用队列来实现线程间的同步。

运行线程(线程中包含name属性)的两种常用方式如下:

* 在构造函数中传入用于线程运行的函数
* 在子类中重写threading.Thread基类中run()方法(只需重写**init**()和run()方法)

实现一个守护线程的简单例子如下：

|  |
| --- |
| class MyThread(threading.Thread):def run(self):time.sleep(30)print 'thread %s finished.' % self.namedef MyDaemons():print 'start thread:'for i in range(5):t = MyThread()t.setDaemon(1)t.start()print 'end thread.'if \_\_name\_\_ == '\_\_main\_\_':MyDaemons() |

为了避免线程不同步造成数据不同步，可以对资源进行加锁，也就是访问资源的线程需要获得锁，才能访问。threading 模块中提供了一个 Lock 功能。从Python3.X开始，标准库为提供了concurrent.futures模块，其中的ThreadPoolExecutor和ProcessPoolExecutor两个类，实现了对threading和multiprocessing的进一步抽象，对编写线程池提供了直接支持。

线程在python 被诟病的是，由于GIL的机制致使多线程不能利用机器多核的特性。其实，GIL并不是Python的特性，只是在实现Python解析器(CPython)的时侯所引入的。尽管Python完全支持多线程编程， 但解释器的C语言实现部分在完全并行执行时并不是线程安全的，解释器被一个全局锁即GIL保护着，它确保任何时候都只有一个Python线程执行。

在多线程环境中，Python 虚拟机按以下方式执行:

1. 设置GIL
2. 切换到一个线程去执行
3. 运行指定的字节码指令集合
4. 线程主动让出控制
5. 把线程设置完睡眠状态
6. 解锁GIL
7. 再次重复以上步骤

因此，Python的多线程在多核CPU上，只对于IO密集型计算产生正面效果；而当有至少有一个CPU密集型线程存在，那么多线程效率会由于GIL而大幅下降。

**GC**

Python 中的GC为可配置的垃圾回收器提供了一个接口。通过它可以禁用回收器、调整回收频率以及设置debug选项，也为用户能够查看那些无法回收的对象。

需要了解GC 的两个重要函数是gc.collect（） 和 gc.set\_threshold（）。

gc.collect([generation])触发回收行为，返回unreachable object的数量。generation可选参数，用于指定回收第几代垃圾回收，由此也可看出python使用的是分代垃圾回收。如果不提供参数，表示对整个堆进行回收，即Full GC。

gc.set\_threshold(threshold0[,threshold1[,threshold2)设置不同代的回收频率，GC会把生命周期不同的对象分别放到3种代去管理回收，generation 0即传说中的年轻代，generation 1为老年代等。

一般地，通过比较上次回收之后，比较分配的资源数和释放的资源数来决定是否启动回收，比如，当分配的资源减去释放的资源数超过阈值0时，回收年轻代的对象。相应的，可以通过gc.get\_referents(\*objs)得到对objs任一对象引用的所有对象列表。

在要求极限性能的情况下，并确保程序不会造成对象循环引用的时候，可以禁掉垃圾回收器。通过使用gc.disable()，可以禁掉自动垃圾回收器。

|  |
| --- |
| 1. gc.enable()：激活GC2. gc.disable()：禁用GC3. gc.isenabled():检查是否激活 |

同时，可以用gc.set\_debug(gc.DEBUG\_LEAK)来调试有内存泄露的程序。除此之外，还有DEBUG\_SAVEALL，该选项能够让被回收的对象保存在gc.garbage里面，以便检查。

**调试**

iPDB是一个不错的工具，通过 pip install ipdb 安装该工具，然后在你的代码中import ipdb; ipdb.set\_trace()，然后在程序运行时，会获得一个交互式提示，每次执行程序的一行并且检查变量。示例代码如下：

|  |
| --- |
| import ipdbipdb.set\_trace()ipdb.set\_trace(context=5) # will show five lines of code# instead of the default three linesipdb.pm()ipdb.run('x[0] = 3')result = ipdb.runcall(function, arg0, arg1, kwarg='foo')result = ipdb.runeval('f(1,2) - 3') |

另外，python内置了一个很好的追踪模块，当希望搞清其他程序的内部构造的时候，这个功能非常有用。

|  |
| --- |
| python -m trace --trace tracing.py |

在一些场合，可以使用pycallgraph来追踪性能问题，它可以创建函数调用时间和次数的图表。同时，objgraph对于查找内存泄露非常有用。

当然， 在Python 程序员八荣八耻中谈到“以打印日志为荣 , 以单步跟踪为耻“，日志在很多时候都是调试的不二法门。

**性能优化中的雕虫小技**

从时空的角度看，优化通常包含两方面的内容：减小代码的体积，提高代码的运行效率。

一个良好的算法往往对性能起到关键作用，因此性能改进的首要点是对算法的改进。在算法的时间复杂度排序上依次是：

|  |
| --- |
| O(1) -> O(log n) -> O(n) -> O(n log n) -> O(n^2) -> O(n^3) -> O(n^k) -> O(k^n) -> O(n!) |

因此能在时间复杂度上对算法进行一定的改进，对性能的提高不言而喻。

Python 字典中查找操作的复杂度为O(1)，而list 实际是个数组，在list 中查找需要遍历整个表，其复杂度为O(n)，因此对成员的读操作字典要比列表 更快。在需要多数据成员进行频繁访问的时候，字典是一个较好的选择。set的union， intersection，difference操作要比list的迭代要快。因此如果涉及到求list交集，并集或者差的问题可以转换为set来操作。

对循环的优化所遵循的原则是尽量减少循环过程中的计算量，有多重循环的尽量将内层的计算提到上一层。 在循环的时候使用 xrange 而不是 range，因为 xrange() 在序列中每次调用只产生一个整数元素。而 range() 将直接返回完整的元素列表，用于循环时会有不必要的开销。另外，while 1 要比 while True 更快。另外，要充分利用Lazy if-evaluation的特性，也就是说如果存在条件表达式if x and y，在 x 为false的情况下y表达式的值将不再计算。

python中的字符串对象是不可改变的，因此对任何字符串的操作如拼接，修改等都将产生一个新的字符串对象，而不是基于原字符串，因此这种持续的copy会在一定程度上影响python的性能。因此，在字符串连接的使用尽量使用join()而不是+，当对字符串处理的时候，首选内置函数，对字符进行格式化比直接串联读取要快，尽量使用列表推导和生成器表达式。

优化的前提是需要了解性能瓶颈在什么地方，对于比较复杂的代码可以借助一些工具来定位，如profile。profile的使用非常简单，只需要在使用之前进行import即可。对于profile的剖析数据，如果以二进制文件的时候保存结果的时候，可以通过pstats模块进行文本报表分析，它支持多种形式的报表输出，是文本界面下一个较为实用的工具。

Python性能优化除了改进算法，选用合适的数据结构之外，还可以将关键python代码部分重写成C扩展模块，或者选用在性能上更为优化的解释器等。

**强大的库**

Python最棒的地方之一，就是大量的第三方库，覆盖之广，令人惊叹。Python 库有一个缺陷就是默认会进行全局安装。为了使每个项目都有一个独立的环境，需要使用工具virtualenv，再用包管理工具pip和virtualenv配合工作。

尽管都可以求助于google或者baidu，但还要不自量力，按照个人认知给出一个列表，如下：

|  |  |  |
| --- | --- | --- |
| **领域** | **简要说明** | **示例库** |
| **包管理** | **管理包和依赖的工具** | **pip，conda 等** |
| **分发与安装** | **打包为可执行文件** | **PyInstaller 等** |
| **构建** | **将源码编译成软件** | **BitBake，PlatformIO 等** |
| **解释器** | **交互式 Python 解析器** | **IPython 等** |
| **编辑器** | **Python 代码编辑器** | **Anaconda，Python-mode 等** |
| **IDE** | **集成开发环境** | **pydev，Spyder 等** |
| **进程** | **操作系统进程启动及通信库** | **envoy，sh 等** |
| **并发** | **用以进行并发和并行操作的库** | **gevent，eventlet 等** |
| **网络** | **用于网络编程的库** | **Twisted，pyzmq 等** |
| **WebSocket** | **用于网络编程的库** | **AutobahnPython，Crossbar 等** |
| **RPC** | **兼容 RPC 的服务器** | **SimpleJSONRPCServer，zeroRPC 等** |
| **软件定义网络** | **网络可视化和SDN的工具和库** | **Pyretic，POX 等** |
| **硬件** | **对硬件进行编程的库** | **ino，Pyro 等** |
| **GUI** | **创建图形用户界面程序的库** | **wxPython，PyQt，PySide 等** |
|  |  |  |
| **文件** | **文件管理和 MIME类型检测** | **mimetypes，watchdog 等** |
| **文本处理** | **用于解析和操作文本的库** | **chardet，simplejson，pyparsing 等** |
| **特殊文本格式** | **一些用来解析和操作特殊文本格式的库** | **python-docx，PDFMiner，PyYAML 等** |
| **文档** | **用以生成项目文档的库** | **Sphinx 等** |
| **配置文件** | **用来保存和解析配置文件的库** | **ConfigParser 等** |
| **图像处理** | **用来操作图像的库** | **PIL，ImageMagic，python-qrcode 等** |
| **音频** | **用来操作音频的库** | **eyeD3，audioread 等** |
| **视频** | **用来操作视频和GIF的库** | **moviepy，scikit-video 等** |
| **地理信息** | **地理编码地址以及用来处理经纬度的库** | **GeoIP，GeoDjango 等** |
| **密码学** | **各种加解密工具库** | **cryptography，PyCrypto 等** |
| **算法** | **Python 实现的算法和设计模式** | **algorithms，python-patterns 等** |
| **游戏开发** | **游戏开发库** | **Cocos2d，Pygame，Panda3D 等** |
| **日志** | **游戏开发库** | **Sentry，logbook 等** |
| **数据库驱动** | **用来连接和操作数据库的库** | **PyMySQL，psycopg2 等** |
| **关系型ORM** | **实现关系型数据映射的库** | **SQLAlchemy 等** |
| **NoSQL驱动** | **用来连接和操作NoSQL的库** | **PyMongo，redis-py，py2neo，HappyBase 等** |
| **NoSQL ORM** | **实现NoSQL数据映射的库** | **MongoEngine，Hot-redis 等** |
| **HTTP** | **HTTP协议的工具库** | **requests，urllib3 等** |
| **Restful API** | **用来开发RESTful API的库** | **flask-restful，falcon 等** |
| **URL 处理** | **解析url的库** | **webargs，furl 等** |
| **HTML处理** | **处理 HTML和XML的库** | **BeautifulSoup，cssutils，html5lib 等** |
| **网页处理** | **用于进行网页内容提取的库** | **opengraph，Haul 等** |
| **网页处理** | **用于进行网页内容提取的库** | **opengraph，Haul 等** |
| **网页生成** | **用于进行网页内容提取的库** | **Pelican，Hyde 等** |
| **表单处理** | **爬取网络站点的库** | **Deform，WTForms 等** |
| **数据验证** | **数据验证库，可用于表单验证** | **Cerberus，schema 等** |
| **管理面板** | **数据验证库，可用于表单验证** | **Ajenti，flask-admin 等** |
| **授权验证** | **实现验证方案的库** | **OAuthLib，python-oauth2 等** |
| **模版引擎** | **模板生成和词法解析的库和工具** | **Jinja2，Mako 等** |
| **队列** | **处理事件以及任务队列的库** | **celery，mrq 等** |
| **搜索引擎** | **对数据进行索引和执行搜索查询的库** | **elasticsearch-py，solrpy 等** |
| **Feed 消息** | **用来创建用户活动的库** | **Stream-Framework 等** |
| **Web框架** | **兼容 WSGI 的 web 服务器** | **gunicorn，uwsgi等** |
| **WSGI** | **丰富的互联网应用** | **Django，Flask，Tornado等** |
| **资源管理** | **丰富的互联网应用** | **fanstatic，jinja-assets-compressor等** |
| **缓存** | **缓存数据的库** | **django-cache-machine，django-cacheop等** |
| **CMS** | **内容管理系统** | **django-cms等** |
| **电子商务** | **用于电子商务以及支付的框架和库** | **django-shop，merchant等** |
| **电子邮件** | **用来发送和解析电子邮件的库** | **envelopes，inbox等** |
| **国际化** | **用来进行国际化的库** | **Babel等** |
| **测试框架** | **单元测试库** | **nose，pytest，Robot Framework等** |
| **Web测试** | **web应用测试库** | **Selenium，sixpack等** |
| **mock测试** | **Mock测试库** | **mock，httppretty等** |
| **测试数据** | **生成测试数据的库** | **mixer，faker等** |
| **代码分析** | **用于代码分析及可视化的库** | **pycallgraph，pysonar2，coverage等** |
| **Lint工具** | **用于静态代码分析的库** | **Flake8，pylint等** |
| **调试工具** | **用于debug的库** | **ipdb，wdb等** |
| **性能工具** | **辅助确定性能瓶颈的库** | **profiling，Memory Profiler等** |
| **高性能** | **让 Python 更快的库** | **cython，pypy等** |
| **devops** | **辅用于 DevOps 的软件和库** | **Ansible，Fabric，pexpect等** |
| **CI** | **持续集成工具库** | **CircleCI，Wercker等** |
| **任务调度** | **任务调度库** | **APScheduler，TaskFlow等** |
| **科学计算** | **科学计算的库** | **numpy，pandas，blaze，scipy等** |
| **学科专属** | **天文，化学，生物学等的库** | **astropy，cclib，Biopython等** |
| **数据可视化** | **进行数据可视化的库** | **matplotlib，ggplot，boker等** |
| **OCR** | **光学字符识别库** | **pytesseract 等** |
| **计算机视觉** | **计算机视觉库** | **OpenCV，SimpleCV 等** |
| **自然语言处理** | **NLP相关的python库** | **NLTK，Jieba 等** |
| **机器学习** | **机器学习库** | **scikit-learn，tensorflow，theano等** |
| **大数据** | **MapReduce 框架和库** | **PySpark，streamparse等** |
| **云端工具** | **云服务相关的python库** | **aws-cli，apache-libcloud等** |
| **云构建** | **用于构建私有和公有云的库** | **OpenStack，Docker Compose等** |

虽然罗列很多，但终归是沧海一粟，重要的是，这些都是开源的。

**不是小结的小结**

语法数据，类与继承；

包与模块，规范命名；

描述装饰，迭代生成；

Lambda GC， 并发线程；

调试优化，类库无穷；

人生苦短，Python 编程。