目录

[Python编程基础 2](#_Toc477790637)

[变量 2](#_Toc477790638)

[字符串 2](#_Toc477790639)

[find 查找字符串是否含有子串 2](#_Toc477790640)

[replace 字符串替换 3](#_Toc477790641)

[列表 3](#_Toc477790642)

[遍历列表时获取每个元素的index 3](#_Toc477790643)

[字典 3](#_Toc477790644)

[元组 3](#_Toc477790645)

[函数式编程 3](#_Toc477790646)

[map 3](#_Toc477790647)

[filter 4](#_Toc477790648)

[reduce 4](#_Toc477790649)

[lambda 4](#_Toc477790650)

[闭包 5](#_Toc477790651)

[概念 5](#_Toc477790652)

[存在意义 5](#_Toc477790653)

[与普通函数不同 5](#_Toc477790654)

[使用场景 5](#_Toc477790655)

[面向对象 6](#_Toc477790656)

[类 6](#_Toc477790657)

[对象 6](#_Toc477790658)

[classmethod VS staticmethod 7](#_Toc477790659)

[内省 7](#_Toc477790660)

[inspect 7](#_Toc477790661)

[getmembers 7](#_Toc477790662)

[设计模式 8](#_Toc477790663)

[单例模式 8](#_Toc477790664)

[装饰器实现 8](#_Toc477790665)

[多线程 8](#_Toc477790666)

[threading.Thread 模块 8](#_Toc477790667)

[队列 9](#_Toc477790668)

[task\_done(): 9](#_Toc477790669)

[join(): 9](#_Toc477790670)

[内置库 9](#_Toc477790671)

[sys 9](#_Toc477790672)

[argv 9](#_Toc477790673)

[exc\_info() 9](#_Toc477790674)

[os 10](#_Toc477790675)

[argparse 10](#_Toc477790676)

[定义一个解析器 10](#_Toc477790677)

[创建定位参数(positional arguments) 10](#_Toc477790678)

[创建可选参数(optional arguments) 10](#_Toc477790679)

[第三方库 12](#_Toc477790680)

[日志记录 12](#_Toc477790681)

[日志等级 12](#_Toc477790682)

[控制器 14](#_Toc477790683)

[模块级别函数（重要） 15](#_Toc477790684)

[配置信息 15](#_Toc477790685)

[通过ini 15](#_Toc477790686)

[分析工具 16](#_Toc477790687)

[profilehook 16](#_Toc477790688)

[异步队列 17](#_Toc477790689)

[Celery 17](#_Toc477790690)

[安装: 17](#_Toc477790691)

[配置并运行 17](#_Toc477790692)

作者：梁毅强

邮箱：398504533@qq.com

# Python编程基础

## 变量

## 字符串

### find 查找字符串是否含有子串

s = “this is python string”

print s.find(‘this’)

#如果存在则返回子串在字符串的开始位置，这里是0，如果不存在则返回-1。

print ‘this’ in s

#如果存在则返回True,否则返回False

### replace 字符串替换

字符串替换

s = “a.cn,b.com” #将逗号替换为空格

new\_s = s.replace(“,”, “ ”)

## 列表

### 常用方法

list.append(*x*)

在列表最末尾添加元素; 等于a[len(a):] = [x].

list.extend(*L*)

将一个列表的所有元素添加到另一个列表中， 等于a[len(a):] = L.

list.insert(*i*, *x*)

在给定的位置插入一个元素，i是位置，x是元素，最常用的有a.insert(0, x)，从开头插入，a.inert(len(a),x) 相当于a.append(x)

list.remove(*x*)

从列表中移除一个元素，如果这个元素的值是x,如果没有这个元素，则会产生一个ValueError错误。

list.pop([*i*])

以给定的位置移除列表中的一个元素，并返回这个元素，如果不给定位置，则a.pop()移除最后一个元素。

list.index(*x*)

返回列表中以x为值的元素位置，如果没有，则抛出ValueError异常

list.count(*x*)

统计列表里x的个数

list.sort(*cmp=None*, *key=None*, *reverse=False*)

元素排序

list.reverse()

反转列表里的元素

### 像Stack堆栈一样使用list

stack 特性：last-in first-out

>>> stack = [3, 4, 5]

>>> stack.append(6)

>>> stack.append(7)

>>> stack

[3, 4, 5, 6, 7]

>>> stack.pop()

7

>>> stack

[3, 4, 5, 6]

>>> stack.pop()

6

>>> stack.pop()

5

>>> stack

[3, 4]

### 像Queue队列一样使用list

队列特性：first-in first-out

>>> from collections import deque

>>> queue = deque(["Eric", "John", "Michael"])

>>> queue.append("Terry") # Terry arrives

>>> queue.append("Graham") # Graham arrives

>>> queue.popleft() # The first to arrive now leaves

'Eric'

>>> queue.popleft() # The second to arrive now leaves

'John'

>>> queue # Remaining queue in order of arrival

deque(['Michael', 'Terry', 'Graham'])

### 遍历列表时获取每个元素的index

例：

jk = [1,2,3]

sni = ['aa','','bb']

pt = ['cc','dd','']

def get\_total():

totals = []

for idx,val in enumerate(jk):

#print 'idx:',idx, ' val:',val

totals.append({'jk':val, 'sni':sni[idx], 'pt':pt[idx]})

return totals

print get\_total()

[{'sni': 'aa', 'pt': 'cc', 'jk': 1}, {'sni': '', 'pt': 'dd', 'jk': 2}, {'sni': 'bb', 'pt': '', 'jk': 3}]

更为pythonic的写法

def get\_total2():

return [{'jk':val, 'sni':sni[idx], 'pt':pt[idx]} for idx, val in enumerate(jk)]

## 列表生成器

### 作用域

列表生成器有自己的作用域，与闭包不同，其作用域与外界是隔离的。

class A(object):

x = 1

gen = (x for \_ in xrange(10)) # gen=(x for \_ in range(10))

if \_\_name\_\_ == "\_\_main\_\_":

print(list(A.gen))

会抛NameError,因为gen有自己的作用域

## 字典

## 元组

## 函数式编程

### map

用一个函数处理列表里的每个元素,这个函数是处理函数

例子：

sq = [1,2,4,5,6]

def add10(ls):

return map(lambda x:x+10, ls)

def add100(ls):

return map(lambda x:x+100, ls)

ls\_10 = add10(sq)

ls\_100 = add100(sq)

[11, 12, 14, 15, 16]

[101, 102, 104, 105, 106]

上例里 lambda x:x+10 和 lambda x:x+100就是处理函数

### filter

用一个函数对列表的元素进行过滤，这个函数只返回True,False

例子：

sq = range(20)

def is\_odd(x): #判断x是否是奇数

return x % 2 != 0

rs = filter(is\_odd, sq)

#[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

比较pythonic的做法：

filter(lambda x:x%2 != 0, range(20)) #一句话搞定

重点是lambda 语法

### reduce

用一个处理函数每次处理序列的前两个元素

例子：

ls = [1,2,3,4,5,6]

reduce(lambda x,y:x+y, ls) #21

处理方式：(((((1+2)+3)+4)+5)+6)

### lambda

函数原型：lambda [arguments]: expression

lambda 可以定义一个匿名简单函数

例子：

取出列表中是负数的数

sq = [1,3,-4,5,66,10,-8]

print filter(lambda x:x < 0, sq)

lambda x:x<0 转换成函数为

def isNegative(x):

if x < 0:return x

lambda 的表达式包含了返回值。

## 闭包

### 概念

在一些语言中，在函数中可以（嵌套）定义另一个函数时，如果内部的函数引用了外部的函数的变量，则可能产生闭包。闭包可以用来在一个函数与一组“私有”变量之间创建关联关系。在给定函数被多次调用的过程中，这些私有变量能够保持其持久性。

### 存在意义

同一个函数，根据参数的不同而带了不同的外部变量，实现不同的功能，可以理解为轻量级的接口封装。

### 与普通函数不同

拥有\_\_closure\_\_ 属性，里面定义了一个元组以保存所有的cell对象，每个cell对象保存了这个闭包中所有的外部变量。

def make\_printer(msg1, msg2):

def printer():

print msg1, '----',msg2

return printer

printer = make\_printer("foo", "bar")

printer.\_\_closure\_\_

printer.\_\_closure\_\_[0].cell\_contents

printer.\_\_closure\_\_[1].cell\_contents

### 使用场景

1. 装饰器
2. 对函数进行提前赋值

用functools.parial侧可，也可以用闭包实现

## 面向对象

### 类

新式类：

class MyClass(object):pass

继承自object

旧式类

无继承

class MyClass():pass

新式类与旧式类\_\_dict\_\_上的不同

**class MyOldTest**()**:pass  
  
class MyTest**(object)**:pass  
  
print** MyTest.\_\_dict\_\_  
**print** MyOldTest.\_\_dict\_\_

{'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'MyTest' objects>, '\_\_module\_\_': '\_\_main\_\_', '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'MyTest' objects>, '\_\_doc\_\_': None}

{'\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': None}

\_\_dict\_\_在这里是类属性，打印的是类变量

### 对象

\_\_dict\_\_ 对象的这个属性可以打印实例变量，显示在\_\_init\_\_里定义的属性和动态定义的属性

**class MyClass**(object)**:** ivar **=** '22'  
 **def \_\_init\_\_**(self)**:** self.name **=** 'cat'  
  
v **=** MyClass()  
**print** v.\_\_dict\_\_  
v.age **=** 20 #动态定义的属性  
**print** v.\_\_dict\_\_

输出

{'name': 'cat'}

{'age': 20, 'name': 'cat'}

#### classmethod VS staticmethod

### 内省

#### inspect

##### getmembers

*# -\*- coding=utf-8 -\*-***class** MyCar(object):  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 **def** get\_name(self):  
 **return** self.name  
  
 **def** get\_age(self):  
 **return** self.age  
  
  
**import** inspect  
  
car = MyCar(**'aaa'**, 22)  
methods = inspect.getmembers(car, predicate=inspect.ismethod)  
**for** method **in** methods:  
 **if** method[0].startswith(**'get'**):  
 **print** method[0], **' '**, method[1]()

get\_age 22

get\_name aaa

## 设计模式

### 单例模式

#### 装饰器实现

**def singleton**(*cls*)**:** instances **=** {}  
  
 **def wrapper**(*\*args*, *\*\*kwargs*)**:  
 if not** instances.has\_key(cls)**:** instances[cls] **=** cls(**\****args*, **\*\****kwargs*)  
 **return** instances[cls]  
  
 **return** wrapper  
  
**@singleton  
class MyClass**(object)**:  
 def \_\_init\_\_**(self, *n*)**:** *#装饰器作用在init之前，执行装饰器的wrapper方法*

self.n **=** *n* **print** 'n:',*n*p **=** MyClass(10)  
  
p2 **=** MyClass(11)

## 多线程

### threading.Thread 模块

start

getName()

setName()

isDaemon()

setDaemon()

join(timeout)

run()

join等待直到线程结束

## 命令行操作

1.像shell脚本一样运行python脚本

如果脚本并不知道python解析器地址，要运行python脚本

1. 在脚本开头加上#!/usr/bin/env python
2. chmod +x script.py 赋执行权限
3. ./script.py 运行脚本

2.指定源文件编码

默认会以ASCII编码保存源文件要改变编码，需要在#!/usr/bin/env python下一行加上

# -\*- coding: encoding -\*-

3.-m 参数的用法

像运行脚本一样运行一个库模块

python -m SimpleHTTPServer 8080

再看SimpleHTTPServer 的代码

def test(HandlerClass = SimpleHTTPRequestHandler,

ServerClass = BaseHTTPServer.HTTPServer):

BaseHTTPServer.test(HandlerClass, ServerClass)

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

test()

文档提示，需要有\_\_main\_\_ 才能让模块运行于-m 参数下，也就是上述是运行了test()函数

## 队列

### task\_done():

指示前面get出来队列元素已经处理完成，

### join():

等待队列为空再执行别的动作

# 内置函数、库

## sys

### argv

argv[0] 脚本名称，是否显示全路径要看操作系统，根据验证，win7和centos6.5只显示文件名，不显示全路径，要显示全路径可以用os.path.realpath(argv[0])

argv[1..n] 脚本的参数，因为是通过标准输入，所以所传递的类型全都是str。

### exc\_info()

当程序产生异常时，此函数能返回含有三个值的元组，(type, value, traceback)

type: 异常对象类型

value:传递给异常对象的参数

traceback: traceback object

## os

## locals()

局部变量，不可以修改

def f(arg):

x = 1

print locals()

f(333)

结果：{'x': 1, 'arg': 333}

## globals()

## argparse

详细用法参考：<http://www.jianshu.com/p/fef2d215b91d>

命令行选项、参数和子命令的解析器

### 定义一个解析器

parser = argparse.ArgumentParser(description='Example with long option names')

### 创建定位参数(positional arguments)

定位参数的意思是,不用带-的参数

简单示例：

保存文件为job.py

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("echo")

args = parser.parse\_args()

print args.echo #args.echo 为一个变量，即命令行传递过来的第一个参数

如：python job.py foo

则args.echo 的值为foo

### 创建可选参数(optional arguments)

两种方式：

1. 用 – 来指定短参数， 如-h
2. 用 - - 来指定长参数，如 - - help

示例：

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("-v", "--verbosity", help="increase output verbosity")

args = parser.parse\_args()

if args.verbosity:

print "verbosity turned on"

调用例子：

python prog.py -v 1

如果不给-v指定参数，会报错，如果要像-h这样不用指定参数，就要用action功能

例子二：

import argparse

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

parser = argparse.ArgumentParser(prog="uop\_jobs")

parser.add\_argument("-v", "--verbose", help="increate output verbose.")

parser.add\_argument("-g", "--group\_type", help="select dba|other sla group.")

#parser.add\_argument('echo')

args = parser.parse\_args()

#print args.echo

print 'verbose:',args.verbose

if args.group\_type == 'dba':

print 'select dba'

elif args.group\_type == 'other':

print 'select other'

else:

raise Exception("select group error.")

命令行调用：

[root@mytest scripts]# python learn\_arg.py -g other

verbose: None

select other

action 的用法

如果option arguments不需要加参数，像-h这样，则可以用action来指定

示例：

import argparse

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

parser = argparse.ArgumentParser()

parser.add\_argument("-g", "--group", action="store\_true", help="this is group select")

args = parser.parse\_args()

print "select group:",args.group

调用：

[root@mytest scripts]# python learn\_arg2.py

select group: False

[root@mytest scripts]# python learn\_arg2.py -g

select group: True

[root@mytest scripts]#

以上例子用了action=”store\_true”，意思是如果加了-g 则args.group的值为True,否则为False

# 第三方库

## 日志记录

### 日志等级

| **Level** |  |  | **Numeric value** |  |
| --- | --- | --- | --- | --- |
| CRITICAL |  |  | 50 | 特别糟糕的事情发生时，如内存耗尽、磁盘满 |
| ERROR |  |  | 40 | 有错误发生时，  def get\_user\_by\_id(user\_id):      user = db.read\_user(user\_id)      if user is None:          logger.error('Cannot find user with user\_id=%s', user\_id)          return user      return user |
| WARNING |  |  | 30 | 有重要的事件发生时，但并不是错误  def authenticate(user\_name, password, ip\_address):      if user\_name != USER\_NAME and password != PASSWORD:          logger.warn('Login attempt to %s from IP %s', user\_name, ip\_address)          return False      # do authentication here |
| INFO |  |  | 20 | 在处理请求或者服务器状态变化等日常事务中，我会使用 INFO 等级  def handle\_request(request):      logger.info('Handling request %s', request)      # handle request here        result = 'result'      logger.info('Return result: %s', result)    def start\_service():      logger.info('Starting service at port %s ...', port)      service.start()      logger.info('Service is started') |
| DEBUG |  |  | 10 | 大多数的情况下，你都不想阅读日志中的太多细节。因此，只有你在调试过程中才会使用 DEBUG 等级。我只使用 DEBUG 获取详细的调试信息，特别是当数据量很大或者频率很高的时候，比如算法内部每个循环的中间状态。  def complex\_algorithm(items):      for i, item in enumerate(items):          # do some complex algorithm computation            logger.debug('%s iteration, item=%s', i, item) |
| NOTSET |  |  | 0 |  |

一般情况最后的notset 不会使用

设置日志等级的方法

**import** logging  
logging.basicConfig(level=logging.DEBUG)

根据等级向上冒泡原则，等级数值越大，权限越高

设定等级的好处：

1. 控制消息级别，过滤掉不重要的信息，只关注重点信息

### 控制器

|  |  |  |
| --- | --- | --- |
| 控制器 | 解释 | 方法 |
| StreamHandler | 流控制器，类似于--stdout --stderr | emit()  flush() |
| FileHandler | 文件控制器，保存到文件进行持久化 | close()  emit() |
| RotatingFileHandler | 放置文件控制器  RotatingFileHandler(filename, mode='a', maxBytes=0, backupCount=0, encoding=None, delay=0)  可控制日志文件大小，保存日志数量 | doRollover()  emit() |
| TimedRotatingFileHandler |  |  |
| SocketHandler | 基于TCP,保存到网络 | close()  emit()  handleError()  makeSocket()  makePickle()  send()  createSocket() |
| DatagramHandler | 继承自Sockethandler,基于UDP | emit()  makeSocket()  send() |
| SysLogHandler | 参数linux的SysLog | close()  emit()  encodePriority()  mapPriority() |

可以定制日志的等级、日志保存方式、日志保存格式(format)等

handler = logging.FileHandler(**"./tlog.log"**) #保存在文件里  
handler.setLevel(logging.INFO) #等级为INFO  
logger.addHandler(handler) #添加

### 模块级别函数（重要）

logging.getLogger([name])

如果不设定name,则返回logger层级中的根logger(root logger),一般名称是点分隔的层级名，如”a”, “a.b”, “a.b.c.d”，以相同名字调用的总返回同一个logger

logging.getLoggerClass()

返回标准的Logger类，或者最后一次传给setLoggerClass()的类

logging.setLoggerClass(klass)

logging.debug(msg [,\*args[, \*\*kwargs]])

kwargs是两个关键词参数：

1. exc\_info 默认为False,设置为True后，异常信息会被加入到日志消息中,也可以传递一个tuple(以sys.exc\_info返回的形式)，将使用元组信息。
2. extra 字典形式，用以填充LogRecord的\_\_dict\_\_

### 配置信息

#### 通过ini

**import** logging  
**import** logging.config

*# load the logging configuration*logging.config.fileConfig(**'logging.ini',** disable\_existing\_loggers=False

)

##################配置示例#################

[loggers]  
keys=root  
  
[handlers]  
keys=fileHandler  
  
[formatters]  
keys=simpleFormatter  
  
[logger\_root]  
level=DEBUG  
handlers=fileHandler  
  
[handler\_fileHandler]  
class=FileHandler  
level=INFO  
formatter=simpleFormatter  
args=("/tmp/filelog.log",)  
  
[handler\_consoleHandler]  
class=StreamHandler  
level=DEBUG  
formatter=simpleFormatter  
args=(sys.stdout,)  
  
[formatter\_simpleFormatter]  
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s  
datefmt=

####################################################

重要参数：disable\_existing\_loggers=False

表示如果模块里存在logger,则让其生效，默认值为True,会关闭已经存在的logger

## 分析工具

### profilehook

三个主要的装饰器

profile() 对函数进行性能分析

coverage() 计算函数被调用的次数

timecall() 简单统计函数处理时间

使用例子：

@profile  
**def** check\_fun():  
 **for** i **in** range(100000):  
 **pass**

## 异步队列

### Celery

#### 安装:

pip install –U celery flower

flower是一个专门监控celery性能的前端界面应用

#### 配置并运行

默认使用redis作为broker

示例：

tasks.py

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

from celery import Celery

app = Celery("tasks", backend="redis://localhost", broker='redis://localhost//')

@app.task()

def add(x, y):

return x + y

命令行下运行：celery -A tasks worker

计算：

>>> from task import add

>>> res = add.delay(1,2)

>>> res.get()