**目录**

[1. 创建类： 4](#_Toc3475218)

[2. 定义方法 4](#_Toc3475219)

[3. 内部变量与private变量(此处变量即属性) 4](#_Toc3475220)

[4. 程序入口 4](#_Toc3475221)

[5. super()函数/方法 4](#_Toc3475222)

[6. assert断言 4](#_Toc3475223)

[7. try...except…finally 5](#_Toc3475224)

[8. \_\_str\_\_与\_\_repr\_\_方法 6](#_Toc3475225)

[9. \_\_call\_\_方法 6](#_Toc3475226)

[10. \_\_dict\_\_属性 6](#_Toc3475227)

[11. Type动态创建类 7](#_Toc3475228)

[12. @property 7](#_Toc3475229)

[13. @staticmethod 8](#_Toc3475230)

[14. 查找模块下的方法(函数)、属性 9](#_Toc3475231)

[15. Import与from…import…的区别 9](#_Toc3475232)

[16. Tuple与()的使用区别 10](#_Toc3475233)

[17. 浅拷贝与深拷贝 10](#_Toc3475234)

[18. Python时间格式化输出 10](#_Toc3475235)

[19. 格式化输出 10](#_Toc3475236)

[20. Windows环境下文件路径表示 10](#_Toc3475237)

[21. Python接收命令行参数 11](#_Toc3475238)

[22. '\u'前缀字符串 11](#_Toc3475239)

[23. 定义1个元素的tuple 11](#_Toc3475240)

[24. 创建生成器(generator)的两种方式 11](#_Toc3475241)

[25. python代码规范 11](#_Toc3475242)

[26. 迭代器总结 11](#_Toc3475243)

[27. Map 函数总结 12](#_Toc3475244)

[28. Reduce函数总结 12](#_Toc3475245)

[29. 匿名函数(lambda)总结 12](#_Toc3475246)

[30. 全局变量的使用 13](#_Toc3475247)

[31. python 定义类时，内部方法的互相调用 13](#_Toc3475248)

[32. filter函数总结 13](#_Toc3475249)

[33. sorted函数总结 14](#_Toc3475250)

[34. 返回函数(闭包) 14](#_Toc3475251)

[35. 装饰器 14](#_Toc3475252)

[36. 函数参数 14](#_Toc3475253)

[37. 字符串里面的引号 15](#_Toc3475254)

[38. 偏函数 15](#_Toc3475255)

[39. 算法速度的定义 15](#_Toc3475256)

[40. 计算运行时间的一般法则 16](#_Toc3475257)

[41. 导入自定义模块 16](#_Toc3475258)

[42. 子类继承父类的属性问题 16](#_Toc3475259)

[43. 子类扩展父类属性 17](#_Toc3475260)

[44. list逆序 17](#_Toc3475261)

[45. i = false 与i == 0 18](#_Toc3475262)

[46. 多进程 18](#_Toc3475263)

[1. Linux/类UNIX下创建多进程(fork()系统调用) 18](#_Toc3475264)

[2. 跨平台的多进程模块(multiprocessing)创建多进程 18](#_Toc3475265)

[3. 创建子进程(此时子进程为外部进程，并非父进程的复制)暂未理解 19](#_Toc3475266)

[4. 进程间通信 19](#_Toc3475267)

[47. 多线程 21](#_Toc3475268)

[48. 附录 23](#_Toc3475269)

[5. 方法解析顺序（Method Resolution Order, MRO）列表 23](#_Toc3475270)

[6. 查询模块的帮助文档 23](#_Toc3475271)

[7. python中时间日期格式化符号 23](#_Toc3475272)

1. 创建类：
2. class Student(object):
3. def \_\_init\_\_(self, name, score):
4. self.name = name
5. self.score = score
6. bart = Student('Bart Simpson', 59)

类名首字母大写，基本类为object, \_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，在\_\_init\_\_方法内部，就可以把各种属性绑定到self有了\_\_init\_\_ 方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_ 方法匹配的参数，但self不需要传

1. 定义方法

在类的定义内：

1. **def** print\_score**(**self**):**
2. **print** '%s: %s' **%** **(**self**.**name**,** self**.**score**)**

要定义一个方法，除了第一个参数是self 外，其它和普通函数一样

1. 内部变量与private变量(此处变量即属性)

**私有变量/属性/方法：**属性的名称前加上两个下划线\_，就变成了一个私有变量（private）

**特殊变量/属性/方法：**变量名类似\_\_xxx\_\_的，以双下划线开头，以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量

一个下划线开头的实例变量名，比如\_name ，这样的实例变量外部可以访问，但按照约定俗成“请视为私有变量，不要随意访问”

1. 程序入口

if \_\_name\_\_ == '\_\_main\_\_'的意思是：当.py文件被直接运行时，if \_\_name\_\_ == '\_\_main\_\_'之下的代码块将被运行；当.py文件以模块形式被导入时，if \_\_name\_\_ == '\_\_main\_\_'之下的代码块不被运行

1. super()函数/方法

super() 函数是用于调用父类(超类)的一个方法。单继承：super 与直接用类名调用父类方法无异；多继承，会涉及到查找顺序（MRO）、重复调用（钻石继承）等问题，此时应区分super与直接类名调用父类。

MRO 就是类的方法解析顺序表, 其实也就是继承父类方法时的顺序表(见[附录1](#MRO))

1. assert断言
2. assert condition

用来让程序测试这个condition，如果condition为false，那么raise一个AssertionError出来。逻辑上等同于：

1. if not condition:
2. raise AssertionError()

assert断言语句添加异常参数，其实就是在断言表达式后添加字符串信息，用来解释断言并更好的知道是哪里出了问题。格式如下：

assert expression [, arguments]

assert 表达式 [, 参数]

1. >>> assert len(lists) >=5,'列表元素个数小于5'
2. Traceback (most recent call last):
3. File "D:/Data/Python/helloworld/helloworld.py", line 1, in <module>
4. assert 2>=5,'列表元素个数小于5'
5. AssertionError: 列表元素个数小于5
6. try...except…finally
7. a=10
8. b=0
9. try:
10. print(a/b)
11. except:
12. print("error")
13. finally:
14. print("always excute")

先执行try，如果出错执行except，无论如何都执行finally.

1. a=10
2. b=0
3. try:
4. print(a/b)
5. except:
6. print("error")
7. else:
8. print("always excute")

先执行try，如果出错执行except，如果不出错执行else

1. \_\_str\_\_与\_\_repr\_\_方法

\_\_repr\_\_：repr() 函数将对象转化为供解释器读取的形式，返回一个对象的string格式

\_\_str\_\_：str() 函数将对象转化为适于人阅读的形式，返回一个对象的string格式，

两个方法均用于返回对象供人阅读，\_\_str\_\_()用于显示给用户，而\_\_repr\_\_()用于显示给开发人员。print调用的是\_\_str\_\_方法，直接输出实例调用的是\_\_repr\_\_方法

1. >>> class Student(object):
2. ... def \_\_init\_\_(self, name):
3. ... self.name = name
4. ...
5. >>> print(Student('Michael'))
6. <\_\_main\_\_.Student object at 0x109afb190>
7. **>>>** s **=** Student**(**'Michael'**)**
8. >>> s
9. **<**\_\_main\_\_**.**Student object at 0x109afb310**>**
10. \_\_call\_\_方法

当实例以函数形式被调用，其调用的即为\_\_calll\_\_方法

1. In [35]: class A:
2. ....: def \_\_init\_\_(self):
3. ....: print "init"
4. ....: def \_\_call\_\_(self):
5. ....: print "call"
6. In [36]: a = A()
7. init
8. In [37]: a()
9. call
10. \_\_dict\_\_属性

\_\_dict\_\_属性是用来存储对象属性的一个字典，其键为属性名，值为属性的值；

1. class A(object):
2. class\_var = 1
3. a = A()
4. a.\_\_dict\_\_ #仅存储与该实例相关的实例属性
5. A.\_\_dict\_\_ #存储所有实例共享的变量和函数(类属性，方法等),但不包含父类属性
6. Type动态创建类

通过type()函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用type()函数创建出class。要创建一个class对象，type()函数依次传入3个参数：

1.class的名称；

2.继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；

3.class的方法名称与函数绑定，这里我们把函数fn绑定到方法名hello上。

1. >>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello class
2. >>> h = Hello()
3. >>> h.hello()
4. Hello, world.
5. >>> print(type(Hello))
6. <type 'type'>
7. >>> print(type(h))
8. <class '\_\_main\_\_.Hello'>
9. @property

提供如下功能：有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量(即把方法变成属性)

1. class Student(object):
2. def get\_score(self):
3. return self.\_score
4. def set\_score(self, value):
5. if not isinstance(value, int):
6. raise ValueError('score must be an integer!')
7. if value < 0 or value > 100:
8. raise ValueError('score must between 0 ~ 100!')
9. self.\_score = value
10. >>> s = Student()
11. >>> s.set\_score(60) # ok!
12. >>> s.get\_score()
13. 60
14. >>> s.set\_score(9999)
15. Traceback (most recent call last):
16. ...
17. ValueError: score must between 0 ~ 100!
18. # 通过如下方式解决上述问题
19. class Student(object):
20. @property
21. def score(self):
22. return self.\_score
23. @score.setter
24. def score(self, value):
25. if not isinstance(value, int):
26. raise ValueError('score must be an integer!')
27. if value < 0 or value > 100:
28. raise ValueError('score must between 0 ~ 100!')
29. self.\_score = value
30. >>> s = Student()
31. >>> s.score = 60 # OK，实际转化为s.set\_score(60)
32. >>> s.score # OK，实际转化为s.get\_score()
33. 60
34. >>> s.score = 9999
35. Traceback (most recent call last):
36. ...

ValueError: score must between 0 ~ 100!

1. @staticmethod

参考： <https://stackoverflow.com/questions/40834145/whats-the-point-of-staticmethod-in-python>

* instance methods: require the instance as the first argument
* class methods: require the class as the first argument
* static methods: require neither as the first argument

1. class TestClass:
2. weight = 200 # class attr
3. def \_\_init\_\_(self, size):
4. self.size = size # instance attr
5. def instance\_mthd(self, val):
6. print("Instance method, with 'self':", self.size\*val)
7. @classmethod
8. def class\_mthd(cls, val):
9. print("Class method, with `cls`:", cls.weight\*val)
10. @staticmethod
11. def static\_mthd(val):
12. print("Static method, with neither args:", val)
13. a = TestClass(1000)
14. a.instance\_mthd(2)
15. # Instance method, with 'self': 2000
16. TestClass.class\_mthd(2)
17. # Class method, with `cls`: 400
18. a.static\_mthd(2)
19. # Static method, with neither args: 2
20. 查找模块下的方法(函数)、属性

均在python解释器下查询，详见[附录2.查询模块的帮助文档](#查询帮助)

1. Import与from…import…的区别

import datetime是引入整个datetime包，如果使用datetime包中的datetime类,需要加上模块名的限定；from datetime import datetime是只引入datetime包里的datetime类,在使用时无需添加模块名的限定

1. Tuple与()的使用区别

tuple( seq )，seq -- 要转换为元组的序列。用法如下：

1. aList **=** **[**123**,** 'xyz'**,** 'zara'**,** 'abc'**];**
2. aTuple **=** tuple**(**aList**)**
3. **print** "Tuple elements : "**,** aTuple
4. **>>>**Tuple elements **:** **(**123**,** 'xyz'**,** 'zara'**,** 'abc'**)**

()则直接使用：

1. **aTuple =** **(**123**,** 'xyz'**,** 'zara'**,** 'abc'**)**
2. 浅拷贝与深拷贝
3. Python时间格式化输出

strftime()函数接收以时间元组(struct\_time对象)，并返回以可读字符串表示的当地时间，格式由参数format决定

1. t **=** time**.**time**()//**获得以秒为单位的时间
2. **print(**time**.**strftime**(**"%b %d %Y %H:%M:%S"**,** time**.**gmtime**(**t**)))//**gmtime获得struct\_time对象

格式参数format，详见[附录3.python中时间日期格式化符号](#日期格式)

1. 格式化输出

格式输出详见参考文档 [《python的格式化输出》](file:///C:\Users\Alta\Desktop\编程笔记\python的格式化输出.pdf)

1. Windows环境下文件路径表示

Python代码里面，反斜杠“\”是转义符，例如“\n”表示回车，采用以下三种方式表示路径：

* 斜杠“/”，如“c:/test.txt”
* 两个反斜杠“\\”，如“c:\\test.txt”
* 字符串前面加上字母r，表示后面是一个原始字符串raw string，如“r“c:\\test.txt””

1. Python接收命令行参数

导入argv，结果即为参数列表

1. **from** sys **import** argv
2. **print(**argv**)**
3. **>>>**python xx**.**py xxx
4. **>>>[**'xx.py'**,**'xxx'**]**
5. '\u'前缀字符串

\u4f60十六进制代表对应汉字的utf-16编码

1. 定义1个元素的tuple

定义1个元素的tuple: t = (1,)，加上一个逗号，避免成为数学意义上的括号

1. 创建生成器(generator)的两种方式

方式1: g = (x \* x for x in range(10))，列表生成式的[]更改为()

方式2: 如果一个函数定义中包含yield关键字，那么函数就不再是一个普通函数，而是一个generator，函数是顺序执行，遇到return语句或者最后一行函数语句返回。而generator在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行

1. **def** fib**(**max**):**
2. n**,** a**,** b **=** 0**,** 0**,** 1
3. **while** n **<** max**:**
4. **yield** b
5. a**,** b **=** b**,** a **+** b
6. n **=** n **+** 1
7. **return** 'done'
8. python代码规范

请参考[《python代码规范》](file:///C:\Users\Alta\Desktop\编程笔记\python代码规范.pdf)

1. 迭代器总结

* 凡是可作用于for循环的对象都是Iterable类型；
* 凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；
* 集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。
* Python的for循环本质上就是通过不断调用next()函数实现的

1. Map 函数总结

map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，返回新的Iterator。

map(function, iterable1,iterable2 ...)

例如三个列表相乘：

1. list1 **=** **[**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**]**
2. list2 **=** **[**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**]**
3. list3 **=** **[**9**,**8**,**7**,**6**,**5**,**4**,**3**,**2**,**1**]**
4. **def** foo**(**l1**,**l2**,**l3**):**
5. **return** l1**\***l2**\***l3
6. **print(**list**(**map**(**foo**,**list1**,**list2**,**list3**)))**
7. **>>>[9,32,63,96,125,144,147,128,81]**
8. Reduce函数总结

reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算。

例如序列求和：

1. **>>>** **from** functools **import** reduce
2. **>>>** **def** add**(**x**,** y**):**
3. **...** **return** x **+** y
4. **...**
5. **>>>** reduce**(**add**,** **[**1**,** 3**,** 5**,** 7**,** 9**])**
6. 25
7. 匿名函数(lambda)总结

lambda parameters: expression

parameters：可选，如果提供，通常是逗号分隔的变量表达式形式，即位置参数。

expression：**不能包含分支或循环（但允许条件表达式）**，也不能包含return（或yield）函数。如果为元组，则应用圆括号将其包含起来。调用lambda函数，**返回的结果是对表达式计算产生的结果**

1. #根据参数是否为1 决定s为yes还是no
2. **>>>** s **=** **lambda** x**:**"yes" **if** x**==**1 **else** "no"
3. 全局变量的使用

使用到的全局变量只是作为引用，不在函数中修改它的值的话，不需要加global关键字

1. a **=** 1
2. **def** func**():**
3. **if** a **==** 1**:**
4. **print(**"a: %d" **%**a**)**

使用到的全局变量，需要在函数中修改的话，就涉及到歧义问题，因此，需要修改全局变量a，可以在"a = 2"之前加入global a声明

1. a **=** 1
2. **def** func**():**
3. **global** a
4. a **=** 2
5. **print(**"in func a:"**,** a)
6. python 定义类时，内部方法的互相调用

每次调用内部的方法时，方法前面加 self

1. class MyClass:
2. def \_\_init\_\_(self):
3. pass
4. def func1(self):
5. print('a')
6. self.common\_func()
7. def func2(self):
8. self.common\_func()
10. def common\_func(self):
11. pass
12. filter函数总结

filter()也接收一个函数和一个序列。filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素

1. sorted函数总结

sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序，key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序

1. **>>>** sorted**([**36**,** 5**,** **-**12**,** 9**,** **-**21**],** key**=**abs**)**
2. **[**5**,** 9**,** **-**12**,** **-**21**,** 36**]**
3. 返回函数(闭包)
4. 装饰器

这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。本质上，decorator就是一个返回函数的高阶函数。

1. **def** log**(**func**):**
2. **def** wrapper**(\***args**,** **\*\***kw**):**
3. **print(**'call %s():' **%** func**.**\_\_name\_\_**)**
4. **return** func**(\***args**,** **\*\***kw**)**
5. **return** wrapper

如果将上述的log函数作为装饰器，则在其装饰的函数前添加@log

1. @log
2. **def** now**():**
3. **print(**'2015-3-25'**)**
4. >>> now()
5. call now():
6. 2015-3-25

装饰器原理及引入机制较为复杂，详情可参考[《python装饰器解释》](file:///C:\Users\Alta\Desktop\编程笔记\python装饰器解释.pdf)

1. 函数参数

* 必选参数(位置参数)：
* 默认参数：如def power(x, n=2)，x即为位置参数，n为默认参数，必选参数在前，默认参数在后
* 可变参数：传入的参数个数是可变的，不必自行将所有参数组装成一个list或tuple，如def calc(\*numbers)。定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个\*号。在函数内部，参数numbers接收到的是一个tuple

1. **def** calc**(\***numbers**):**
2. sum **=** 0
3. **for** n **in** numbers**:**
4. sum **=** sum **+** n **\*** n
5. **return** sum

* 关键字参数:可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict:

1. **def** person**(**name**,** age**,** **\*\***kw**):**
2. **print(**'name:'**,** name**,** 'age:'**,** age**,** 'other:'**,** kw**)**
4. **>>>** person**(**'Michael'**,** 30**)**
5. name**:** Michael age**:** 30 other**:** **{}**
6. **>>>** person**(**'Bob'**,** 35**,** city**=**'Beijing'**)**
7. name**:** Bob age**:** 35 other**:** **{**'city'**:** 'Beijing'**}**
8. 字符串里面的引号

单引号' '定义字符串的时候，它就会认为你字符串里面的双引号" "是普通字符，从而不需要转义。反之当你用双引号定义字符串的时候，就会认为你字符串里面的单引号是普通字符无需转义

1. Str1 = "We all know that 'A' and 'B' are two capital letters."
2. Str2 = 'The teacher said: "Practice makes perfect" is a very famous proverb.'
3. 偏函数

functools.partial的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单

1. **>>>** **import** functools
2. **>>>** int2 **=** functools**.**partial**(**int**,** base**=**2**)**
3. **>>>** int2**(**'1000000'**)**
4. 64
5. 算法速度的定义

* 大O：T(N) = O(f(N))，T增长率小于等于f
* **Ω**：T(N) =Ω (f(N))，T增长率大于f
* **Θ**：T(N) =Θ (f(N))，T增长率等于f
* 小o：T(N) =o (f(N))，T增长率小于f

1. 计算运行时间的一般法则

* 法则1：for循环：一个for循环的运行时间至多是该循环内语句的运行时间乘以迭代次数
* 法则2：嵌套for循环：嵌套循环内部的一条语句总运行时间为该语句运行时间乘以所有for循环的大小，如下程序片段运行时间为O(N2)

1. **for** i **in** range**(**N**):**
2. **for** j **in** range**(**N**):**
3. j **+=** 1

* 法则3：顺序语句：各个语句运行时间求和
* 法则4：if/else语句：运行时间至多是判断时间+S1或S2中运行时间长者

1. **if** **(**condition**):**
2. S1
3. **else:**
4. S2
5. 导入自定义模块
6. 子类继承父类的属性问题

更加细致的继承中的属性和方法问题，请参考《[python类的继承、属性总结和方法总结](file:///C:\Users\Alta\Desktop\编程笔记\python类的继承、属性总结和方法总结.pdf)》

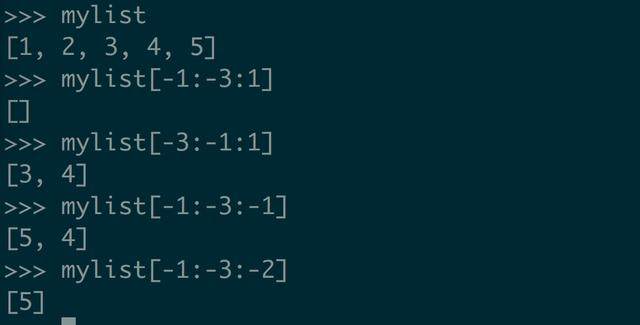
如果子类自己定义了\_\_init\_\_方法，那么父类的属性是不能调用的，如下：

1. class Animal:
2. def \_\_init\_\_(self):
3. self.a = 'aaa'
4. class Cat(Animal):
5. def \_\_init\_\_(self):
6. pass
7. cat = Cat()
8. print(cat.a)
9. >>>AttributeError: 'Cat' object has no attribute 'a'

可以在子类的 \_\_init\_\_中调用一下父类的 \_\_init\_\_ 方法,这样就可以调用父类的属性

1. class Animal:
2. def \_\_init\_\_(self):
3. self.a = 'aaa'
4. class Cat(Animal):
5. def \_\_init\_\_(self):
6. super().\_\_init\_\_()
7. cat = Cat()
8. print(cat.a)
9. >>>aaa
10. 子类扩展父类属性
11. list逆序

* list.reverse()会直接在原来的列表里面将元素进行逆序排列，不需要创建新的副本用于存储结果，调用 list.reverse() 的返回值是 None
* 使用切片 [::-1]，mylist[start:end:step]：上面的操作表示取 mylist 的第 start 个（列表索引从 0 开始）到第 end 个元素（不包括第 end 个），其中每隔 step 个（默认 1 ）取一个。



* 使用 reversed() 方法，reversed 方法会将列表逆序的结果存储到迭代器里面，这种方式不会改变原来的列表，也不会创建原来列表的完整副本

1. i = false 与i == 0

在python3中，如果将i赋值False，则i == 0为True

1. 多进程
2. Linux/类UNIX下创建多进程(fork()系统调用)
3. import os
4. print('Process (%s) start...' % os.getpid())
5. pid = os.fork()
6. if pid == 0:
7. print('I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid()))
8. else:
9. print('I (%s) just created a child process (%s).' % (os.getpid(), pid))
10. >>>Process (876) start...
11. >>>I (876) just created a child process (877).
12. >>>I am child process (877) and my parent is 876.

普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后分别在父进程和子进程内返回。子进程永远返回0，而父进程返回子进程的ID.

1. 跨平台的多进程模块(multiprocessing)创建多进程
2. from multiprocessing import Process
3. import os
4. def run\_proc(name):
5. print('Run child process %s (%s)...' % (name, os.getpid()))
6. if \_\_name\_\_=='\_\_main\_\_':
7. print('Parent process %s.' % os.getpid())
8. p = Process(target=run\_proc, args=('test',))
9. print('Child process will start.')
10. p.start()
11. p.join()
12. print('Child process end.')

multiprocessing模块提供了一个Process类来代表一个进程对象，创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

1. 创建子进程(此时子进程为外部进程，并非父进程的复制)暂未理解
2. 进程间通信

* Queue方式(相当于创建了一个临时交换区)

Queue有两个方法：A.Put方法：以插入数据到队列中，两个可选参数：blocked和timeout. B.Get方法：从队列读取并且删除一个元素。两个可选参数：blocked和timeout

1. #- \* -coding: utf - 8 - \* -
2. from multiprocessing import Process, Queue
3. import os, time, random
4. # 写数据进程执行的代码:
5. def write(q):
6. print('Process to write: %s' % os.getpid())
7. for value in ['A', 'B', 'C']:
8. print('Put %s to queue...' % value)
9. q.put(value)
10. time.sleep(random.random())
11. # 读数据进程执行的代码:
12. def read(q):
13. print('Process to read: %s' % os.getpid())
14. while True:
15. value = q.get()
16. print('Get %s from queue.' % value)
17. if \_\_name\_\_ == '\_\_main\_\_': #父进程创建Queue， 并传给各个子进程：
18. q = Queue()
19. pw = Process(target = write, args = (q, ))
20. pr = Process(target = read, args = (q, ))# 启动子进程pw， 写入:
21. pw.start()# 启动子进程pr， 读取:
22. pr.start()# 等待pw结束:
23. pw.join()# pr进程里是死循环， 无法等待其结束， 只能强行终止:
24. pr.terminate()

输出如下：

1. Process to read: 5836
2. Process to write: 6472
3. Put A to queue...
4. Put B to queue...
5. Get A from queue.
6. Put C to queue...
7. Get B from queue.
8. Get C from queue.
9. Process finished with exit code 0

* pipe方式

1. #- \* -coding: utf - 8 - \* -
2. from multiprocessing import Process, Pipe
3. def f(conn):
4. conn.send([42, None, 'hello'])
5. while True:
6. print(conn.recv())
7. if \_\_name\_\_ == '\_\_main\_\_':
8. parent\_conn, child\_conn = Pipe()
9. p = Process(target = f, args = (child\_conn, ))
10. p.start()
11. print parent\_conn.recv()# prints "[42, None, 'hello']"
12. parent\_conn.send('666')
13. p.terminate()

输出如下：

1. [42, None, 'hello']
2. 666
3. Process finished with exit code 0

Pipe（）函数返回一对由管道连接的连接对象（Connection（conn1，conn2）对象），默认情况下是双工（双向）。Pipe（）返回的两个连接对象代表管道的两端。每个连接对象都有send（）和recv（）方法（等等）。conn1发出(send)的消息传递到了conn2的接收端(recv)，反之亦然。

请注意，如果两个进程（或线程）尝试同时读取或写入管道的同一端，管道中的数据可能会损坏。当然，同时使用管道不同端的过程也不会有风险。如果duplex为True（默认），则管道是双向的。如果duplex是False，那么管道是单向的：conn1只能用于接收消息，conn2只能用于发送消息.

1. 多线程

创建一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行：

1. import time, threading
2. # 新线程执行的代码:
3. def loop():
4. print('thread %s is running...' % threading.current\_thread().name)
5. n = 0
6. while n < 5:
7. n = n + 1
8. print('thread %s >>> %s' % (threading.current\_thread().name, n))
9. time.sleep(1)
10. print('thread %s ended.' % threading.current\_thread().name)
11. print('thread %s is running...' % threading.current\_thread().name)
12. t = threading.Thread(target=loop, name='LoopThread')
13. t.start()
14. t.join()
15. print('thread %s ended.' % threading.current\_thread().name)

Threadlocal

多进程模式的缺点是创建进程的代价大，另外，操作系统能同时运行的进程数也是有限的(受内存和CPU的限制)

多线程模式通常比多进程快一点，致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存.

1. 附录
2. 方法解析顺序（Method Resolution Order, MRO）列表

MRO 列表的顺序遵循以下三条原则：

* 子类永远在父类前面
* 如果有多个父类，会根据它们在列表中的顺序被检查
* 如果对下一个类存在两个合法的选择，选择第一个父类

1. 查询模块的帮助文档

* 先导入模块，再查询普通模块的使用方法：help(module\_name)，例：help(math)
* 先导入sys，再查询系统内置模块的使用方法：sys.bultin\_modulenames
* 查看模块下所有函数：dir(module\_name)，例：dir(math)
* 查看模块下特定函数：help(module\_name.func\_name)，例：help(math.sin)
* 查看函数信息的另一种方法：print(func\_name.\_\_doc\_\_)，例：print(sin.\_\_doc\_\_)
* 查看python关键字：help(‘keywords’)

1. python中时间日期格式化符号

* %y 两位数的年份表示（00-99）
* %Y 四位数的年份表示（000-9999）
* %m 月份（01-12）
* %d 月内中的一天（0-31）
* %H 24小时制小时数（0-23）
* %I 12小时制小时数（01-12）
* %M 分钟数（00=59）
* %S 秒（00-59）
* %a 本地简化星期名称
* %A 本地完整星期名称
* %b 本地简化的月份名称
* %B 本地完整的月份名称
* %c 本地相应的日期表示和时间表示
* %j 年内的一天（001-366）
* %p 本地A.M.或P.M.的等价符
* %U 一年中的星期数（00-53）星期天为星期的开始
* %w 星期（0-6），星期天为星期的开始
* %W 一年中的星期数（00-53）星期一为星期的开始
* %x 本地相应的日期表示
* %X 本地相应的时间表示
* %Z 当前时区的名称
* %% %号本身