

# Python简介

2019年8月16日 13:43

**本笔记以廖雪峰的Python 3教程为主题。**

## 特点：

- “脚本语言”
- 简单易用
- 运行速度慢
- 代码不能加密
- 不适用于操作系统/手机应用/3D游戏

## 简介：

由荷兰人Guido van Rossum于1989年设计。

基础代码库完善，有大量第三方库。

适用于网络应用（网站，后台服务）、任务脚本、作为“胶水语言”

# Python解释器

2019年8月16日 13:51

Python代码以.py作为扩展名，运行时使用Python解释器执行.py文件。

**Python的官方解释器是CPython，使用C语言开发。**

PyPy是另一个Python解释器，采用JIT技术进行**动态编译（不是解释）**，显著提高了代码执行速度，但与CPython略有不同。

在命令行模式运行.py文件和在Python交互式环境下直接运行Python代码有所不同。Python交互式环境会**把每一行Python代码的结果自动打印出来**（因为是输入一行执行一行），但是，直接运行Python代码却不会。

在Linux环境中，可以通过在.py文件的开头添加

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

的方式使文件可以被直接执行。

可能还需要使用

```
$ chmod a+x hello.py
```

来授予权限。

# Python基础

2019年8月16日 14:10

以#开头的语句作为注释。

Python的语法采用**缩进方式**，没有一般语言中用花括号限定的块，当语句以冒号:结尾时，缩进的语句视为代码块。

缩进方式好处：

- 代码更加简洁
- 能迫使人写出格式化的代码
- 能迫使人将长代码拆分成若干函数以减少缩进

坏处：

- 复制粘贴时需要仔细排版，检查缩进是否正确
- 无法用IDE格式化代码，需要自己注意


**Python是大小写敏感的。**

# 输入和输出

2019年8月16日 14:44

## 输出：

`print()` 函数用于输出指定的文字。

- 可以接受多个字符串，用逗号， 隔开，连成一串输出。
- **遇到逗号时，会输出一个空格。**
- 如果不想输出空格，可以采用 `str()` 函数把非字符串转为字符串，然后用 `+` 连接起来输出。
- 也可以打印整数、计算结果
- `print` 函数中有 `end` 参数，默认为 `"\n"`，即输出完换行。若不想换行，可以写 `print(xxx, end="")`，将 `end` 参数定义为空。

## 输入：

`input()` 函数用于输入**字符串**并存放到变量里，故返回的是 `str` 类型。

# 数据类型和变量

2019年8月16日 14:53

- 整数

可以处理**任意大小**的整数。

- 浮点数

没有大小限制，但超出一定范围会直接表示为inf（无限大）。

- 字符串

用单引号和双引号都是合法的，三引号用于表示多行内容

r' '表示' '内部不转义。

- 布尔值

用True/False表示布尔值（**注意首字母大写**）

可以使用and/or/not运算。

- 空值

用None表示，与0不等同。

- 变量

Python中可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量。

- 常量

通常用**全部大写**的变量名表示常量。

但实际上**没有任何机制保证常量不被修改**，全大写只是一种习惯。

/除法计算结果是浮点数，即使整除结果也是浮点数（保留一个0）。

//地板除只取结果的整数部分。

%余数运算，取两整数相除的余数。

# 字符串和编码

2019年8月16日 15:43

## 编码、字符串：

Python字符串使用Unicode进行编码，支持多语言。

Unicode一般用2个字节表示一个字符，非常偏僻的字符可能需要4个字节。

`ord()` 获取字符的整数表示

`chr()` 将编码转换为对应字符

但实际上为了节约空间，Unicode在被存储和传输时，使用的是压缩的UTF-8编码，这种编码下常用的英文字符只需1个字节，汉字通常是3字节，生僻的字符在4-6字节。

英文字符较多的情况下，用UTF-8编码可以大大减少使用空间。而且ASCII是UTF-8的一部分，故可以完全兼容ASCII编码的软件。

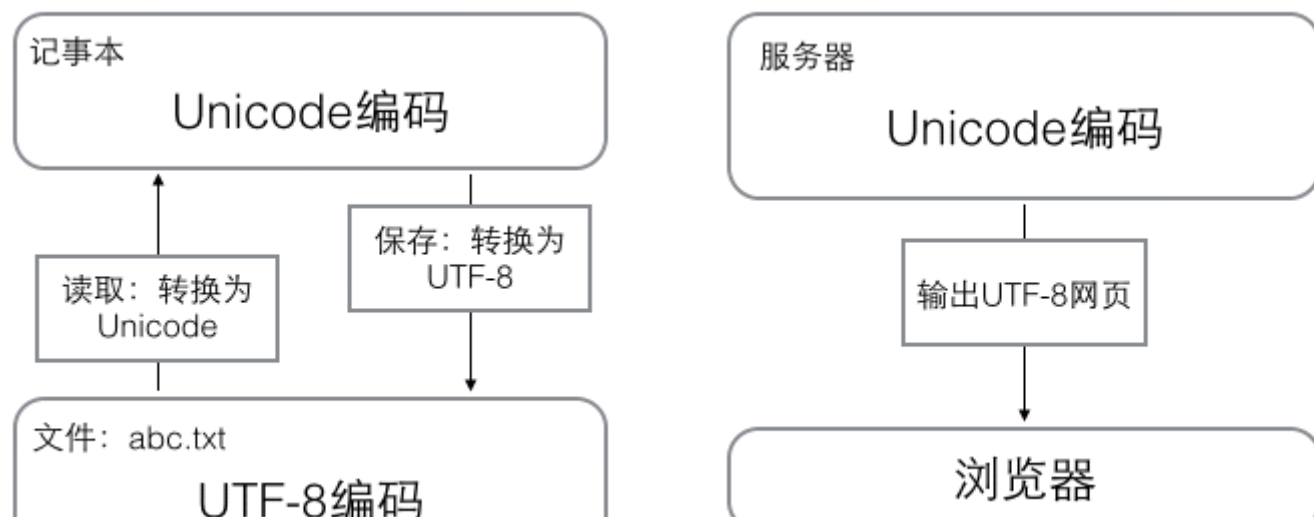
故在传输或保存到磁盘上时，需要使用`encode()`方法将str编码为指定的bytes，参数是编码方式（如'utf-8','ascii'等）。

反过来，从网络或磁盘读取字节流时，读取的是UTF-8编码的bytes，需要用`decode()`方法转为str字符串。

在.py文件的开头，可以用`# -*- coding: utf-8 -*-`让解释器以utf-8编码读取源代码，以保证中文不会出现乱码。

即内存中的字符串是Unicode的str，存储传输用的是UTF-8的bytes！

```
1 >>> 'ABC'.encode('ascii')
2 b'ABC'
3 >>> '中文'.encode('utf-8')
4 b'\xe4\xbb\xad\xe6\x96\x87'
5
6 >>> b'ABC'.decode('ascii')
7 'ABC'
8 >>> b'\xe4\xbb\xad\xe6\x96\x87'.decode('utf-8')
9 '中文'
```



len()函数用于计算str的**字符数**，如果换成bytes，则计算的是字节数。

## 格式化:

格式化的方法和C语言一致，使用%实现。

%d	整数	%f	浮点数
%s	字符串	%x	十六进制整数

如要转义字符串中的'%'，用'%%'。

```
1 #格式化
2 >>> 'Hello, %s' % 'world'
3 'Hello, world'
4 >>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)
5 'Hi, Michael, you have $1000000.'
6
7 #转义
8 >>> 'growth rate: %d %%' % 7
9 'growth rate: 7 %'
10
11 #宽度, 补零, 精度
12 print('%10.6f' % 3.1415926)           # 3.141593
13 print('%010.6f' % 3.1415926)         #003.141593
14 print('%-10.6f%-10.6f' % (3.1415926, 3.1415926)) #3.141593 3.141593
```

另一种格式化方法是对字符串使用format()方法，该方法用传入的参数依次替换字符串中的占位符{0}、{1}、{2}.....

用.format是比较好的方法，建议使用。

# 使用list和tuple

2019年8月16日 20:58

## 列表list:

列表是有序的集合，可以随时添加/删除其中的元素。

类似于数组，但不规定大小，元素类型也可以不同。

### 定义:

```
1 >>> classmates = ['Michael', 'Bob', 'Tracy'] #用方括号
2 >>> classmates
3 ['Michael', 'Bob', 'Tracy']
```

len()函数用于获取list中元素的个数，由于索引从0开始，最后一个元素的索引是

len(list)-1!

还可以用负数索引获取倒数位置的元素。

### 方法:

1. append(elem) 追加元素到末尾
2. insert(n,elem) 插入元素到指定位置
3. pop(n) 弹出指定位置元素，缺省为**末尾元素**

## 元组tuple:

类似于list，但是初始化后元素不能修改（**指向不变**，即如果在tuple中嵌入list而修改list，还是可以变的），因此没有append、insert等方法。

tuple更安全，如果可以应尽量用tuple替代list。

### 定义:

```
1 >>> classmates = ('Michael', 'Bob', 'Tracy') #用圆括号
```

注意如果要定义只有一个元素的tuple，应该在**唯一的元素后加逗号**！（否则就成了赋值）



# 分支和循环

2019年8月17日 15:12

## 分支：

if, else, elif。

判断语句以冒号结尾。

**布尔值True、非零数值、非空字符串、非空list等都判断为True。**

注意用input()输入的整数返回的是str字符串，不能直接作为判断条件（否则非空字符串一律判断为True），应该先用int()转换。

## 循环：

### for...in循环：

这种循环依次迭代list或tuple中的每个元素。for x in ... 即是将每个元素依次代入x，然后执行缩进块的语句。

```
1 sum = 0
2 for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
3     sum = sum + x
4 print(sum)
```

Python提供range()函数，用于生成整数列表，这在for循环中比较有用。

range(start, stop[, step])

- start: 计数从 start 开始。默认是从 0 开始。例如range(5)等价于range(0, 5);
- stop: 计数到 stop 结束，**但不包括 stop**。例如：range(0, 5) 是[0, 1, 2, 3, 4]，没有5;
- step: 步长，默认为1。例如：range(0, 5) 等价于 range(0, 5, 1)。

```
1 >>>x = 'runoob'
2 >>> for i in range(len(x)) :
3     ...     print(x[i])
4 ...
5 r
6 u
7 n
8 o
9 o
10 b
```

### while循环：

break语句可提前结束循环。

continue语句可跳过当前循环的后面部分，直接开始下一次循环。

**不要滥用！**

死循环时，Ctrl+C退出。

? python为何没有C语言中的自增、自减运算？

原因：**python中的数字类型是不可变数据。**

数字类型的数据在内存中不会发生改变，每次变量值改变时都是申请一块新内存并赋值，然后将变量指向新地址，原地址的值没有变。

```
1 >>> a = 10
2 >>> id(a)
3 140530470127960
4 >>> a += 1
5 >>> id(a)
6 140530470129080
```

可以看到当 a 的值发生改变时，a 指向的内存地址也发生了改变。而在C语言中，a 指向的内存地址并不会发生改变，而是改变内存的内容。

`+=` 是改变变量，相当于重新生成一个变量，把操作后的结果赋予这个新生成的变量。

`++` 是改变了对象本身，而不是变量本身，即改变数据地址所指向的内存中的内容。

既然 Python 中的数字类型是不可变的，那何来的“自增”这么一说呢？

**另外：**

int 理论上是每次赋值都创建一个新对象的。但是由于使用频繁，为了提升性能避免浪费，所有 python 有个 整数池，默认 `[-5, 256]` 的数字都属于这个整数池，这些每次赋值的时候，是取得池中的整数对象。但是其他的除外，如下：

```
1 >>> a = 5
2 >>> b = 5
3 >>> id(a)
4 140530470128360
5 >>> id(b)
6 140530470128360
7 >>> a is b
8 True
9
10 >>> c = 257
11 >>> d = 257
12 >>> id(c)
13 140530470136432
14 >>> id(d)
15 140530470136408
```

**但：**

有时候在 Python 中看到存在 `++i` 这种形式，这其实不是自增，只是简单的表示正负数的正号而已。正正得正，负负得正，所以 `++i` 和 `--i` 都是 `i`。



<https://juejin.im/post/5cb425aef265da03867e4421>

# 使用dict和set

2019年8月17日 16:52

## 字典dict:

dict在其它语言中又称为**map**，使用的是键-值（key-value）存储，查找速度不随容量增大而降低。事实上，字典是使用哈希表实现的，对key使用哈希函数得到的即是value的地址（索引）。由于使用了哈希表，占用内存也多得多。

### 定义:

```
1 >>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}           #用花括号
2 >>> d['Michael']
3 95
```

每一项的前半段作为key，后半段作为value。

由于要对key进行哈希函数运算，故key必须是**不可变对象**，如字符串，整数等。

**除了list、dict、set和内部至少带有上述三种类型之一的tuple之外，其余的对象都能当key。**

除了初始化时指定，还可以通过key放入新的key-value，所以dict是可变的。

要删除一个key，使用pop(key)方法，对应的value也会被删除。

用in可以判断某个key是否在dict中，如'Thomas' in d 就会返回一个bool值。

还可对dict使用get(key)方法，返回的是参数key对应的value。如果没有对应的value或者对应的value是None，则会默认返回None。**返回None时Python交互环境不显示结果。**

## 集合set:

集合相当于没有value只有key的dict。

和数学上的集合一样，set中的元素不能重复（如同key不能重复）。

### 定义:

```
1 >>> s = set([1, 1, 2, 2, 3, 3])      #输入一个list来创建set
2 >>> s
3 {1, 2, 3}      #list中重复的元素被过滤
```

add(key)方法可以向set中添加元素，重复添加的元素无效；

remove(key)方法用于移除一个key，不存在则报错；discard(key)则不会报错。

pop()则是**随机删除**一个元素。

set和数学上的集合一样具有互异性和无序性，故可以作交&并|等操作。

由于相当于没有value的dict，故set中也不允许放入list等可变对象（否则可能违反互异性）。

## 不可变对象：

不可变对象**本身**是不可变的。

如果将不可变对象赋值给变量，对变量进行操作可能会创建新的不可变对象，但原本的不可变对象仍然不变（倒是可能因为没有被引用而被自动垃圾回收）。

```
1 >>> a = 'abc'
2 >>> b = a.replace('a', 'A')    #replace实际上作用于字符串'abc'
3 >>> b
4 'Abc'
5 >>> a
6 'abc'    #a仍然没变
```

## 小结

	类型限制	能否重复	是否有序	能否修改
列表list	无	能	是	能
元组tuple	无	能	是	否
字典dict	不可变对象	否	否	能
集合set	不可变对象	否	否	能

# 函数

2019年8月17日 20:15

**函数是最基本的一种代码抽象的方式。**

# 调用函数

2019年8月17日 20:17

python内置了许多有用的函数供调用。在交互式命令行用`help(func)`可查看函数的帮助信息。  
传入参数数量/类型不对时，报错`TypeError`。

函数名其实是对一个函数对象的引用，可以把函数名赋给一个变量，即相当于给函数起了一个“别名”。

# 定义函数

2019年8月17日 20:51

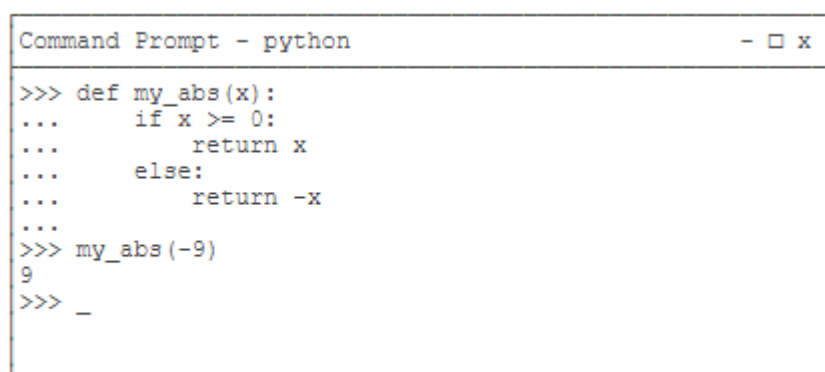
定义一个函数使用 `def` 语句。

依次写出函数名，括号，括号中的参数，最后是冒号 `:`，然后在缩进体中编写函数体。

```
1 def my_abs(x):
2     if x >= 0:
3         return x
4     else:
5         return -x
```

不写 `return` 语句时，函数执行完毕默认返回 `None`。

在 Python 交互环境中定义函数时，注意 Python 会出现 `...` 的提示。函数定义结束后需要按两次回车重新回到 `>>>` 提示符下。



```
Command Prompt - python - □ x

>>> def my_abs(x):
...     if x >= 0:
...         return x
...     else:
...         return -x
...
>>> my_abs(-9)
9
>>> _
```

可以从当前目录的其他 `.py` 文件中导入函数：

```
1 >>> from abctest import my_abs    #从abctest.py中导入my_abs函数
```

## 空函数：

什么也不做的空函数里可以写一句 `pass`。

`pass` 只用作占位符，满足语法要求。

## 返回多值：

`return n1, n2` 可以返回多值。

但实际上，返回多值时，返回的是一个 `tuple` 元组 `(n1, n2)`。

# 函数的参数

2019年8月17日 21:27

除了普通的“位置参数”，python的函数还支持以下几种特殊参数：

## 默认参数：

可以给函数的某个参数设置默认值。

当然，必选参数应该放在前，默认参数放在后，否则解释器会报错。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

```
1 def power(x, n=2):
2     s = 1
3     while n > 0:
4         n = n - 1
5         s = s * x
6     return s
```

## 一个重点：默认参数必须指向不变对象！

原因：若默认参数指向的对象发生变化，默认参数就跟着发生了变化，这会导致错误。

如在以下函数中：

```
1 def add_end(L=[]):
2     L.append('END')
3     return L
4
5 >>> add_end()
6 ['END']
7 >>> add_end()
8 ['END', 'END']    #默认参数[]由于可变，已经被append成了['END']
9 >>> add_end()
10 ['END', 'END', 'END']
11
12 #正确的定义
13 def add_end_right(L=None):    #将默认参数指向不变对象None，保证默认参数不变
14     if L is None:
15         L = []    #临时将L改为指向[]
16     L.append('END')    #[]进行append
17     return L
18
```

## 可变参数：

对于可变参数，传入参数的个数是可变的，甚至可以是0个。

定义：

```
1 def calc(*numbers):    #用*表示可变参数
```



```

2         sum = 0
3         for n in numbers:    #函数内接收到的是一个tuple, 进行迭代
4             sum = sum + n * n
5         return sum

```

支持在现成的list或tuple前加\*作为可变参数传入。\*nums表示把nums这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

## 关键字参数：

关键字参数是一个dict，允许传入若干带参数名的参数，以参数名为key，参数值为value。

关键字参数用于扩展函数功能，即是说就算**没有传入也可以接受**。

**定义：**

```

1 def person(name, age, **kw):    #用**表示关键字参数
2     print('name:', name, 'age:', age, 'other:', kw)
3
4 >>> person('Adam', 45, gender='M', job='Engineer')    #两对键值组成一个dict
5 name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}

```

当然，也支持在现成的dict前加\*\*作为关键字参数传入。传入的dict是原dict的一份拷贝，函数内的变化不会影响到函数外的原dict。

## 命名关键字参数：

顾名思义，命名关键字参数是在关键字参数的基础上，对参数名进行限制。

命名关键字参数需要一个特殊分隔符\*，\*后面的参数被视为命名关键字参数。

**定义：**

```

1 def person(name, age, *, city, job):    #city和job被认为是命名关键字参数
2     print(name, age, city, job)
3
4 >>> person('Jack', 24, city='Beijing', job='Engineer')    #调用时，需要写参数名
5 Jack 24 Beijing Engineer
6
7 def person(name, age, *args, city, job):    #可变参数前的*可以作为分隔符。
8     print(name, age, args, city, job)

```

## 参数组合：

在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用。

但是请注意，参数定义的顺序必须是：

**必选参数、默认参数、可变参数、命名关键字参数和关键字参数。**

# 递归函数

2019年8月18日 0:23

使用递归函数时需要注意防止栈溢出。

由于函数调用需要用到栈，每进入一个函数调用栈就会增加一层栈帧，而栈的容量是有限的，所以**递归次数过多时会导致栈溢出**。

## 尾递归优化：

尾递归是指在函数返回的时候调用自身，并且return语句不能包含表达式（即只能有函数调用本身）。

在这种情况下，编译器/解释器可以对尾递归作优化，使无论递归多少次都只占用一个栈帧。

然而，python解释器**并没有针对尾递归的优化**，所以以上策略并没有什么卵用。

理论上，任何递归都可以写成循环，只是循环的逻辑可能会复杂一些。

所以在可能导致栈溢出的情况下，不如用循环。

## 递归实现的汉诺塔：

```
1 # -*- coding: utf-8 -*-
2 def move(n, a, b, c):
3     if n == 1:
4         print(a, '-->', c) #a只剩一个时直接移动到c
5     else:
6         move(n-1, a, c, b) #要先把上面的(n-1)个挪到b上，故把b作为原来的c
7         print(a, '-->', c) #然后把最下面的挪到c，问题规模变为(n-1)
8         move(n-1, b, a, c) #再把b上的(n-1)个挪到c上，故把b作为原来的a
9
10 # 期待输出：
11 # A --> C
12 # A --> B
13 # C --> B
14 # A --> C
15 # B --> A
16 # B --> C
17 # A --> C
18 move(3, 'A', 'B', 'C')
```

# 高级特性

2019年8月18日 0:56

python中，代码越少越好，越简单越好。

为了实现极致的简介，引入python中非常有用的高级特性。

1行代码能实现的功能，决不写5行代码。

**代码越少，开发效率越高！**

# 切片slice

2019年8月18日 0:58

如何取一个list/tuple乃至str的部分元素？

可以用索引，数量多时要用到循环.....但都太繁琐。

python提供了切片(slice)操作符用于取指定索引范围。

切片操作：[a:b:c]，c默认为1。没有参数的[:]相当于直接原样复制。

从a开始，到b结束，每c个元素取一个。也支持倒数切片。

切什么类型，获得的就是什么类型。

这个操作相当于用“for x in range(a,b,c):”的循环取元素。

```
1 >>> L = list(range(100))    #先用range产生0-99的list
2 >>> L
3 [0, 1, 2, 3, ..., 99]
4
5 >>> L[:10:2]
6 [0, 2, 4, 6, 8]    #第一个参数默认0
```

# 迭代

2019年8月18日 10:57

Python中用for...in进行迭代，遍历一个list或tuple。

相对于C语言采用下标递增for循环的方式进行迭代，Python迭代的抽象程度更高。

迭代还可以用在dict和set等无下标的对象上，但由于它们没有顺序，结果的顺序可能不同。

示例：

```
1 >>> d = {'a': 1, 'b': 2, 'c': 3}
2 >>> for key in d:
3 ...     print(key)
4 ...
5 a
6 c
7 b
```

要判断一个对象能否迭代，需要导入collections模块的Iterable类型：

```
1 >>> from collections import Iterable
2 >>> isinstance('abc', Iterable) # str是否可迭代
3 True
4 >>> isinstance([1, 2, 3], Iterable) # list是否可迭代
5 True
6 >>> isinstance(123, Iterable) # 整数是否可迭代
7 False
```

要实现下标循环，可以使用enumerate()函数将可迭代对象变为枚举对象。

```
1 >>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
2 >>> list(enumerate(seasons))
3 [(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
4 >>> list(enumerate(seasons, start=1)) # 下标从 1 开始
5 [(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

# 列表生成式

2019年8月18日 19:20

List Comprehensions, 是Python内置的用于创建list的生成式。  
之前已经了解的range( )函数就是生成list的一种方法。

在列表的[]内写入表达式, 加上for循环, 就可依照循环计算生成列表元素:

```
1 >>> [x * x for x in range(1, 11)]    #从1到10的x**2
2 [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
3
4 >>> [x * x for x in range(1, 11) if x % 2 == 0]    #还可结合if判断
5 [4, 16, 36, 64, 100]
6
7 >>> [m + n for m in 'ABC' for n in 'XYZ']    #多层循环进行全排列
8 ['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

用列表生成式可以写出非常简洁的代码。

# 生成器

2019年8月19日 14:22

只要将列表生成式的[]改为(), 就是一个生成器 (Generator)。

有时需要生成的列表很大, 但需要访问的元素其实不多, 那么如果使用列表生成式先生成列表, 又慢又占用多余空间。这种情况下, 可以使用生成器。

生成器也是可迭代的对象, 可以使用for循环获取生成的元素。

也可以使用next(generator)函数获取下一个生成的元素。

generator保存的是算法, 不是生成的元素, 只有调用时才进行生成。

## 定义:

### 1. 简单的生成器, 将列表生成式的[]改为():

```
1 >>> L = [x * x for x in range(10)]    #列表生成式
2 >>> L
3 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
4 >>> g = (x * x for x in range(10))    #生成器
5 >>> g
6 <generator object <genexpr> at 0x1022ef630>    #保存的是算法
7
8 >>> next(g)    #next()函数获取元素
9 0
10 >>> next(g)
11 1
12
13 >>> g = (x * x for x in range(10))    #用for规定生成规则
14 >>> for n in g:    #用for迭代获取元素
15 ...     print(n)
16 ...
17 0
18 1
19 4
20 9
21 ...
```

### 2. 复杂的生成器, 在函数中包含yield关键字:

**包含yield关键字的函数是generator。**

函数是顺序执行, 遇到return语句或者最后一行函数语句就返回。而变成generator的函数, 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    #到这里就返回b的值
5         a, b = b, a + b
```

```
6             n = n + 1
7         return 'done'      #return实际上会抛出StopIteration错误
8                             #因此要拿到返回值需要捕获StopIteration错误
9
10 >>> for n in fib(6):      #for循环迭代
11     ...     print(n)
12 ...
13 1
14 1
15 2
16 3
17 5
18
```



# 迭代器

2019年8月19日 23:32

能直接作用于for循环的数据类型，分为**集合数据类型**和**生成器**这两种。这些对象统称为可迭代对象Iterable。

可以被next( )函数调用并不断返回下一个值的对象被称为迭代器Iterator。  
生成器就是一种Iterator。

list、dict、str等**集合数据类型**是Iterable但不是Iterator，可以用iter( )函数将它们变为Iterator。

**iterator对象表示的是一个数据流**，这个数据流只能通过next( )计算下一个数据，直到没有数据是抛出StopIteration。

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，但可以通过iter( )函数获得一个对应的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。非迭代器的可迭代对象在使用for循环时，实际是先调用iter( )方法转换为迭代器，再循环调用next( )并返回值。

# 函数式编程

2019年8月20日 0:21

python对于函数式编程提供部分支持。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！这种编程范式的抽象度很高，但python不是纯函数式编程语言，因为真正的函数式编程语言中甚至不允许存在变量。

# 高阶函数

2019年8月20日 0:23

1. 变量可以指向函数。

2. **函数名**本身也是变量

之前提到，可以把函数名赋给一个变量，相当于给函数起了一个别名。

这里我认为：

所有的变量（包括函数名）都是类似于“**标签**”的东西，函数名只是一个预先贴在函数体这个对象上的标签。

既然如此，将新的标签贴在函数体上、将函数体原本的标签撕下来贴在其他对象上，在python中都是合法的。

3. 传入函数

既然变量可以指向函数，函数的参数能接收变量：

那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

示例：

```
1 def add(x, y, f):    #这里的f就是一个作为参数的函数
2     return f(x) + f(y)
3
4 #x = -5
5 #y = 6
6 #f = abs    #f是绝对值函数
7 #f(x) + f(y) ==> abs(-5) + abs(6) ==> 11
8 #return 11
```

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

# map/reduce

2019年8月20日 0:35

## map( )函数:

其它语言中的map在Python中是dict, 而Python中的map是这样一种函数:

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, 并把结果作为新的Iterator返回。

简单地说, 生成一个迭代器, 其产生的元素是**对Iterable中的元素依次进行函数处理**后的返回值。

```
1 >>> def f(x):
2 ...     return x * x
3 ...
4 >>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])    #对每个元素取平方
5 >>> list(r)    #将map产生的Iterator转为list, 全部计算出来
6 [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

map( )传入的第一个参数是f, 即函数对象本身。由于结果r是一个Iterator, Iterator是惰性序列, 因此通过list( )函数让它把整个序列都计算出来并返回一个list。

## reduce( )函数:

reduce(f, [x1, x2, x3, ...])把一个函数f作用在一个序列[x1, x2, x3, ...]上, 这个函数f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, 其效果就是:

$\text{reduce}(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)$

```
1 >>> from functools import reduce
2 >>> def fn(x, y):
3 ...     return x * 10 + y
4 ...
5 >>> reduce(fn, [1, 3, 5, 7, 9])
6 13579
```

# filter

2019年8月20日 12:19

filter( )用于过滤序列。

与map类似，将传入的函数作用于每个元素，然后根据返回值是True还是False来决定保留/丢弃该元素。

**filter返回的也是一个惰性序列Iterator**，要完成计算结果的话需要以list( )列出所有结果。

```
1 def not_empty(s):
2     return s and s.strip()    #非空字符串视为True
3
4 list(filter(not_empty, ['A', '', 'B', None, 'C', ' ']))
5 # 结果: ['A', 'B', 'C']
```

值得注意的是，**布尔值True、非零数值、非空字符串、非空list等都判断为True。**

# sorted

2019年8月20日 22:10

## 排序算法：

sorted() 函数可以对list进行**从小到大**的排序。

作为高阶函数，sorted() 还可以接受一个key函数来实现自定义的排序。key指定的函数作用于list的每一个元素上，sorted函数会根据key函数返回的结果进行排序。

例如按绝对值大小进行排序：

```
1 >>> sorted([36, 5, -12, 9, -21], key=abs)    #key函数为abs
2 [5, 9, -12, -21, 36]    #依照abs返回的5, 9, 12, 21, 36进排序
```

关于排序的规则：

sorted函数中含有默认参数reverse，该参数默认为False（升序）。

可以在调用sorted函数时传入reverse = True，即可实现降序排序。

# 返回函数

2019年8月21日 21:24

## 函数作为返回值：

如果在函数中嵌套定义一个函数并将其作为返回值，则调用外部函数时返回的是内部嵌套的函数本身，而不是内部函数的返回值。

举例：

```
1 def lazy_sum(*args):
2     def sum(): #嵌套定义的内部函数
3         ax = 0
4         for n in args:
5             ax = ax + n
6         return ax
7     return sum #将内部函数本身作为返回值
8
9 >>> f = lazy_sum(1, 3, 5, 7, 9)
10 >>> f #f被赋为lazy_sum的返回函数的别名
11 <function lazy_sum.<locals>.sum at 0x101c6ed90>
12
13 >>> f()
14 25
```

这种相关参数、变量都保存在返回的函数中的程序结构称为“闭包”。

//相当于是传入不同的参数就产生了不同的函数。（?）

## 闭包：

当一个函数返回了一个函数后，**其内部的局部变量还被新函数引用**。这导致如果内部的局部变量改变了，而之前返回的函数尚未执行，就会产生错误。

**你**

**返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量！**

# 匿名函数

2019年8月22日 12:30

匿名函数即**lambda表达式**，是一种不显示定义函数名的函数。

```
1 lambda x: x * x    #冒号前的是参数
2
3 #等价于:
4 def f(x):
5     return x*x
```

作为函数，当然也可以将其赋给一个变量，此时这个变量相当于成了其“函数名”。  
也可以把匿名函数作为返回值返回。



# 装饰器

2019年8月22日 20:09

## 前方高难!

如何在函数调用运行期间不修改函数定义也能动态增加功能?

使用装饰器 (Decorator) 。

本质上, decorator是一个高阶函数, 其以原函数为参数, 返回值是被“装饰”后的函数。

使用装饰器时, 借助Python的@语法, 将装饰器置于函数的定义处:

```
1 def use_logging(func):
2
3     def wrapper():      #装饰器中的wrapper函数对原函数进行装饰
4         logging.warn("%s is running" % func.__name__)
5         return func()
6     return wrapper      #返回装饰后的函数
7
8 @use_logging
9 def foo():
10     print("i am foo")
11
12 foo()
```

把@log放到now()函数的定义处, 相当于执行了语句:

```
1 now = log(now)
```

这里log(now)的返回值就是now经过装饰后得到的函数。

对于有参数的原函数, 可以在定义wrapper函数时, 指定wrapper的参数为\*args (可变参数), \*\*kw (关键字参数) 参数, 这样就可以接收任意参数, 将原函数的所有参数传入。

```
1 def log(func):
2     def wrapper(*args, **kw):      #接收任意参数
3         print('call %s():' % func.__name__)
4         return func(*args, **kw)   #接收到的参数传入内层返回的func
5     return wrapper
```

如果decorater本身还需要传入参数, 则需要编写一个返回decorator的高阶函数:

```
1 def log(text):
2     def decorator(func):
3         def wrapper(*args, **kw):
4             print('%s %s():' % (text, func.__name__))
5             return func(*args, **kw)
6         return wrapper
7     return decorator
8
9 @log('execute')      #为decorator传入参数'execute'
10 def now():
11     print('2015-3-25')
```

```
12
13 >>> now()      #执行结果
14 execute now():
15 2015-3-25
```

# 偏函数

2019年8月22日 23:45

`functools.partial`用于创建一个偏函数。

Python中的偏函数指在原函数的基础上，**预先固定了某些参数**的衍生函数。

这可以使参数太多的函数调用起来更加简单。

```
1 >>> import functools    #要先导入functools模块
2 >>> int2 = functools.partial(int, base=2)    #原函数int, 指定参数base=2
3 >>> int2('1000000')      #base=2指将参数作为2进制处理
4 64
5 >>> int2('1010101')
6 85
```

当传入：

```
1 max2 = functools.partial(max, 10)
```

实际上会把10作为\*args的一部分自动加到左边，也就是：

```
1 max2(5, 6, 7)
```

相当于：

```
1 args = (10, 5, 6, 7)
2 max(*args)
```

结果为10。

# 模块

2019年8月23日 0:30

Python中，每个.py文件都被称为一个模块(Module)。

相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。

## 如果模块名冲突？

引入包(Package)机制，和Java中的类似。

在每个包目录下**需要有一个“\_\_init\_\_.py”文件**（可以为空），这个文件对应的模块名就是包名本身。如果没有这个文件，Python就会把该目录当做是一个普通目录而不是一个包。

只要顶层包名不冲突，包内的模块就不会与其他包的内容冲突。这和文件系统中的重名文件解决方案是类似的。

为了不与系统模块冲突，创建自己的模块是最好先在Python交互环境里尝试import abc，若成功则说明abc是系统存在的模块名。

# 使用模块

2019年8月23日 16:02

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 ' a test module '      #第一个字符串被视为模块文档注释
5
6 __author__ = 'Michael Liao'    #模块作者署名
7
8 import sys
9
10 def test():
11     args = sys.argv
12     if len(args)==1:
13         print('Hello, world!')
14     elif len(args)==2:
15         print('Hello, %s!' % args[1])
16     else:
17         print('Too many arguments!')
18
19 if __name__ == '__main__':
20     test()
```

要使用模块，第一步是用 **import abc** 导入模块。

## 作用域：

Python中，使用 `_` 前缀来实现作用域的规定。

无前缀的函数、变量是公开(public)的，可以被直接引用。

模块中类似 `__xxx__` 的变量是特殊变量，也可以被直接引用，但有特殊用途，如 `__author__` 和 `__name__`（直接运行模块文件的入口，可以用于测试）等。

类似 `_xxx` 和 `__xxx` 的函数和变量是非公开(private)的，**不应该被直接引用**（Python并没有阻止其被引用的机制）。

Python没有protected这种权限设定。

# 安装使用第三方模块

2019年8月23日 23:30

安装第三方模块使用的是**包管理工具pip**。

Anaconda内置了许多非常有用的第三方库。Anaconda会把系统Path中的python指向自己自带的Python，并且，Anaconda安装的第三方模块会安装在Anaconda自己的路径下，不影响系统已安装的Python目录。

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在sys模块的path变量中。

要添加搜索目录，可以设置**环境变量PYTHONPATH**。

# 面向对象编程

2019年8月23日 23:34

在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student。

所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

## 面向对象的三大特点：

- 数据封装
- 继承
- 多态

# 类和实例

2019年8月23日 23:38

类是抽象的模板，实例是根据类创造出来的一个个具体“对象”。  
同一类的所有实例都拥有一系列相同的方法，但各自数据可能不同。

## 定义类：（不知道继承什么就继承object类）

```
1 class Student(object):    #class关键字定义类，括号内是继承的父类
2     pass                  #object是所有类都会继承的类
```

```
1 class Student(object):    #带有自定义构造方法的类
2
3     def __init__(self, name, score):    #第一个参数self指向创建实例本身
4         self.name = name
5         self.score = score
```

## 创建实例：

```
1 >>> bart = Student('Bart Simpson', 59)    #调用类的构造方法创建实例
2 >>> bart.name
3 'Bart Simpson'
4 >>> bart.score
5 59
```

## 数据封装：

在类中定义方法就是一种将数据“封装”的措施。

定义方法时，除了**第一个参数是 self**，其他和普通函数一样。

调用方法时，不需要传入 self 参数，其他参数正常传入。



# 访问限制

2019年8月24日 16:01

在内部属性前加两个下划线 `__` 就可以使一个属性成为私有(private)属性，只有对象内部可以访问。如果要从外部读取私有属性，可以在对象内部添加公有的如`get_xxx`的方法间接访问。同样的，也可以设置`set_xxx`等方法间接写入私有属性。

## Python的访问限制原理：

对于private属性，如Student类下bart实例的`__name`属性，Python解释器会将其名称从`bart.__name`改为`bart._Student__name`，阻止调用。

但是如果使用这个新名称，**依然可以直接调用private属性**。Python没有完全强制性的措施阻止调用private属性！

**也因此，在外部看来，bart内部实际上已经没有了一个叫`__name`的变量，如果直接从外部调用，反而会增加一个叫`__name`的新变量。**

# 继承和多态

2019年8月24日 16:18

## 继承：

Python中通过定义类时在类名后的括号中写入父类名的方式进行继承。

如：

```
1 class Dog(Animal):    #继承了Animal类
2     pass
3
4 class Cat(Animal):
5     pass
```

**继承时，子类获得了父类的全部功能。**

在子类中可以**重写**父类的方法，重写后在子类中父类的方法会被**覆盖**。

要调用父类方法，可以使用super( )函数。

## 多态：

在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是反过来就不行。

“开闭”原则：

- 对扩展开放（Open for extension）：允许子类重写方法函数
- 对修改封闭（Closed for modification）：不重写，直接继承父类方法函数

## 静态语言 vs 动态语言：

在Java等静态语言中，如果需要传入Animal类型，则只能传入Animal或其子类；

但在Python这种动态语言中，只要传入的对象有Animal类对象所具有的方法，那就可以被当做是Animal。只要实现了某些方法，就可以假装是某种对象。

这种特点被称为“鸭子类型”——“看起来像鸭子，走起路来像鸭子”，那就可以被看做是鸭子。

# 获取对象信息

2019年8月24日 17:39

## 使用type():

type()函数返回对象对应的Class类型。(Python中一切都是对象)

对于指向函数、类的变量,也可使用type():

```
1 >>> type(123)
2 <class 'int'>
3 >>> type('str')
4 <class 'str'>
5 >>> type(None)
6 <type(None) 'NoneType'>
7 >>> type(abs)
8 <class 'builtin_function_or_method'>
9 >>> type(a)
10 <class '__main__.Animal'>
```

## 使用isinstance():

isinstance可以告诉我们一个对象是否是某种类型。返回的是布尔值。

可以判断一个变量是否是某些类型中的一种,比如下面的代码就可以判断是否是list或者tuple:

```
1 >>> isinstance([1, 2, 3], (list, tuple))
2 True
3 >>> isinstance((1, 2, 3), (list, tuple))
4 False
```

## 使用dir():

要获得一个对象的所有属性和方法,可以使用dir()函数,它返回一个包含字符串的list。

比如,获得一个str对象的所有属性和方法:

```
1 >>> dir('ABC')
2 ['__add__', '__class__', ..., '__subclasshook__', 'capitalize', 'casefold', ..., 'zfill']
```

获取到属性、方法列表后,还可以用hasattr, getattr, setattr函数来操作一个对象的状态。

# 实例属性和类属性

2019年9月5日 10:44

在定义类时可以写入类属性，该属性虽然归类所有，但类的所有实例都可以访问到。

（在Java中，称为类成员变量/静态变量）

如果在实例中定义了类属性的名字，则在实例中类属性会被覆盖（即被重写了）。相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

## 访问方式：

- 类变量：  
在类内部和外部都使用 **类名.类变量** 的形式访问。
- 实例变量：  
类内部，实例变量用 **self.实例变量** 的形式访问；  
类外部，实例变量用 **实例名.实例变量** 的形式访问。

# 面向对象高级编程

2019年9月5日 17:07

面向对象的高级特性：

- 多重继承
- 定制类
- 元类
- ...

# 使用\_\_slots\_\_

2019年9月5日 17:09

动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。如：

```
1 >>> def set_score(self, score):
2 ...     self.score = score
3 ...
4 >>> Student.set_score = set_score    #运行时给类绑定新的方法
5
6 >>> s.set_score(100)    #类的所有实例都可使用新绑定的方法
7 >>> s.score
8 100
9 >>> s2.set_score(99)
10 >>> s2.score
11 99
```

如果我们想要限制实例的属性怎么办？比如，只允许对Student实例添加name和age属性。为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，其内容是一个tuple，来限制该class实例能添加的属性。

```
1 class Student(object):
2     __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称
3
4 >>> s = Student() # 创建新的实例
5 >>> s.name = 'Michael' # 绑定属性' name'
6 >>> s.age = 25 # 绑定属性' age'
7 >>> s.score = 99 # 绑定属性' score'
8 Traceback (most recent call last):
9     File "<stdin>", line 1, in <module>
10 AttributeError: 'Student' object has no attribute 'score'
```

试图绑定slots中没有的属性名时，会报AttributeError错误。

**\_\_slots\_\_ 定义的属性仅对当前类实例起作用，对继承的子类不起作用**，除非在子类中也定义\_\_slots\_\_，此时子类实例允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

# 使用@property

2019年9月5日 17:33

前面提到，使用\_\_slots\_\_可以限定属性的名称。

如果要**限定属性的取值范围**，就不得不使用setter和getter方法，在方法中对参数进行检查。为了方便地在set方法中限定属性的取值范围，可以使用@property装饰器。

由于类的setter方法也是一种函数，装饰器对其也有效。

@property装饰器负责把一个方法变成属性调用（相当于getter），并且连带创建了一个@xxx.setter装饰器作为getter。这样例如私有属性\_score的读写就可以不用get\_score和set\_score，而是用装饰器和重载函数（同一范围内声明多个函数名相同、参数表不同的函数，借此完成不同的功能）写成如下形式：

```
1 class Student(object):
2     @property      #property装饰器
3     def score(self):
4         return self._score
5
6     @score.setter   #伴生的setter装饰器
7     def score(self, value):    #重载函数，参数表不同
8         if not isinstance(value, int):    #参数检查部分
9             raise ValueError('score must be an integer!')
10        if value < 0 or value > 100:
11            raise ValueError('score must between 0 ~ 100!')
12        self._score = value
```

# 多重继承

2019年9月10日 18:11

多继承的定义是在定义类时，在class 类名(继承类)中写入多个父类名，这样生成的子类就同时获得多个父类的所有功能。

```
1 class Bat(Mammal, Flyable):      #蝙蝠继承了哺乳、能飞
2     pass
```

设计类的继承关系时，通常，主线都是单一继承下来的，而用多继承“混入”的额外功能通常称为Mixin。通过组合适当的Mixin，就可以创造出合适的服务。



# 定制类

2019年9月10日 18:21

类中前后都有下划线的属性、方法都有特殊用途。许多特殊的方法都可以帮助我们对类进行定制。以下给出一些用于定制类的常用特殊方法。

## `__str__`:

如果在类中实现一个 `__str__(self)` 方法，就会在用`print`打印类的时候打印出该方法的返回值。但注意，在交互界面直接显示变量调用的不是`__str__`而是`__repr__`，这个是为调试服务的。

## `__iter__`:

实现这个方法的类是`Iterable`，类似于`list`和`tuple`，可以用 `for ... in` 进行循环，循环会不断调用类的 `__next__()`方法，直到`StopIteration`使循环退出。

```
1 class Fib(object):
2     def __init__(self):
3         self.a, self.b = 0, 1 # 初始化两个计数器a, b
4
5     def __iter__(self):
6         return self # 实例本身就是迭代对象，故返回自己
7
8     def __next__(self):
9         self.a, self.b = self.b, self.a + self.b # 计算下一个值
10        if self.a > 100000: # 退出循环的条件
11            raise StopIteration()
12        return self.a # 返回下一个值
13
14 >>> for n in Fib():
15 ...     print(n)
16 ...
17 1
18 1
19 2
20 3
21 5
22 ...
23 46368
24
```

## `__getitem__`:

对于实现了`__iter__`的类，虽然是`Iterable`但并不像`list`和`tuple`一样适用下标（`Subscriptable`），需要实现一个`__getitem__()`方法。

在`__getitem__`方法要里单独实现对切片的支持。因为切片`[a:b:c]`也是一种对象（`slice`对象）。另外有`__setitem__`方法，把对象视作`list`或`dict`来对集合赋值。

`__delitem__()`方法，用于删除某个元素。

## `__getattr__`:

对于类中不存在的属性，如果没有实现`__getattr__`方法就会返回一个 `AttributeError` 错误。但是如果实现了该方法，遇到不存在的属性xxx就会试图调用`__getattr__(self, 'xxx')`来尝试获得属性。`__getattr__`方法默认返回的是None，这使得如果实现了`__getattr__`又希望只对特定的几个属性进行响应，就**应该在该方法内依然没有匹配属性时抛出 `AttributeError` 错误。**

```
1 class Student(object):
2
3     def __getattr__(self, attr):
4         if attr=='age':
5             return lambda: 25
6         raise AttributeError('%s\'' object has no attribute \'%s\'' % attr)
```

## `__call__`:

只要定义一个`__call__()`方法，就可以直接对实例进行调用。调用实例时实际上调用实例的`__call__`方法（定义可以带有参数）。

```
1 class Student(object):
2     def __init__(self, name):
3         self.name = name
4
5     def __call__(self):
6         print('My name is %s.' % self.name)
7
8 #调用方法:
9 >>> s = Student('Michael')
10 >>> s() # self参数不要传入
11 My name is Michael.
```

定义了`__call__`方法的类实例可以被`callable()`函数判断为True。

# 使用枚举类

2019年9月11日 22:58

Python原生类型中并没有枚举类，Python 3.4中才添加了enum标准库。

使用枚举类（Enum）有什么好处？

Python中的枚举类型是**不可变类型**，又可进行迭代，所以**使用起来比较安全**，而且占用内存可以更少。这是一种**定义多个同类型常量有限集合的好方法**。

使用枚举类需在定义类时继承Enum类，由于该类定义在enum模块中，所以要用import导入。

```
1 from enum import Enum, unique    #注意大小写，Enum是类名，enum是包名
2
3 @unique    #保证没有重复值的装饰器
4 class Weekday(Enum):
5     Sun = 0 # Sun是一个name，其value被设定为0
6     Mon = 1
7     Tue = 2
8     Wed = 3
9     Thu = 4
10    Fri = 5
11    Sat = 6
12
```

要访问枚举类型，可以用成员名称（name）引用枚举常量，又可以直接根据value的值获得枚举常量。

```
1 >>> print(Weekday.Tue)
2 Weekday.Tue
3 >>> print(Weekday['Tue'])
4 Weekday.Tue
5
6 #枚举类名.成员名称.value用来调用枚举常量的值
7 >>> print(Weekday.Tue.value)
8 2
9
10 #枚举类名(value的值)用来调用枚举常量本身
11 >>> print(Weekday(1))
12 Weekday.Mon
13 >>> Weekday(7)
14 Traceback (most recent call last):
15     ...
16 ValueError: 7 is not a valid Weekday
17
18 #迭代
19 >>> for name, member in Weekday.__members__.items():
20     ...     print(name, '=>', member)
21 ...
22 Sun => Weekday.Sun
23 Mon => Weekday.Mon
24 Tue => Weekday.Tue
```

```
25 Wed => Weekday.Wed
26 Thu => Weekday.Thu
27 Fri => Weekday.Fri
28 Sat => Weekday.Sat
```

枚举类不可实例化，也不可更改，定义之后就是只读的。

# 使用元类(暂缺)

2019年9月12日 11:16

据说很难也很少用到，暂时搁置。

# 错误、调试和测试

2019年9月12日 11:19

在程序运行过程中，总会遇到各种各样的错误。

有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串，这种错误我们通常称之为bug，bug是必须修复的。

有的错误是用户输入造成的，比如让用户输入email地址，结果得到一个空字符串，这种错误可以通过**检查用户输入**来做相应的处理。

还有一类错误是完全**无法在程序运行过程中预测**的，比如写入文件的时候，磁盘满了，写不进去了，或者从网络抓取数据，网络突然断掉了。这类错误也称为**异常**，在程序中通常是必须处理的，否则，程序会因为各种问题终止并退出。

Python内置了一套异常处理机制，来帮助我们进行错误处理。

此外，我们也需要跟踪程序的执行，查看变量的值是否正确，这个过程称为调试。Python的pdb可以让我们以单步方式执行代码。

最后，编写测试也很重要。有了良好的测试，就可以在程序修改后反复运行，确保程序输出符合我们编写的测试。

# 错误处理

2019年9月12日 11:22

高级语言通常都会内置一套try ... except ... finally ... 的错误处理机制。Python的错误处理也是如此。

## try:

当认为某段代码可能会出错时，可以用 try 来运行这段代码。

如果执行过程中出错，则后续代码不会继续执行，而是跳转到错误处理代码（except语句块）。

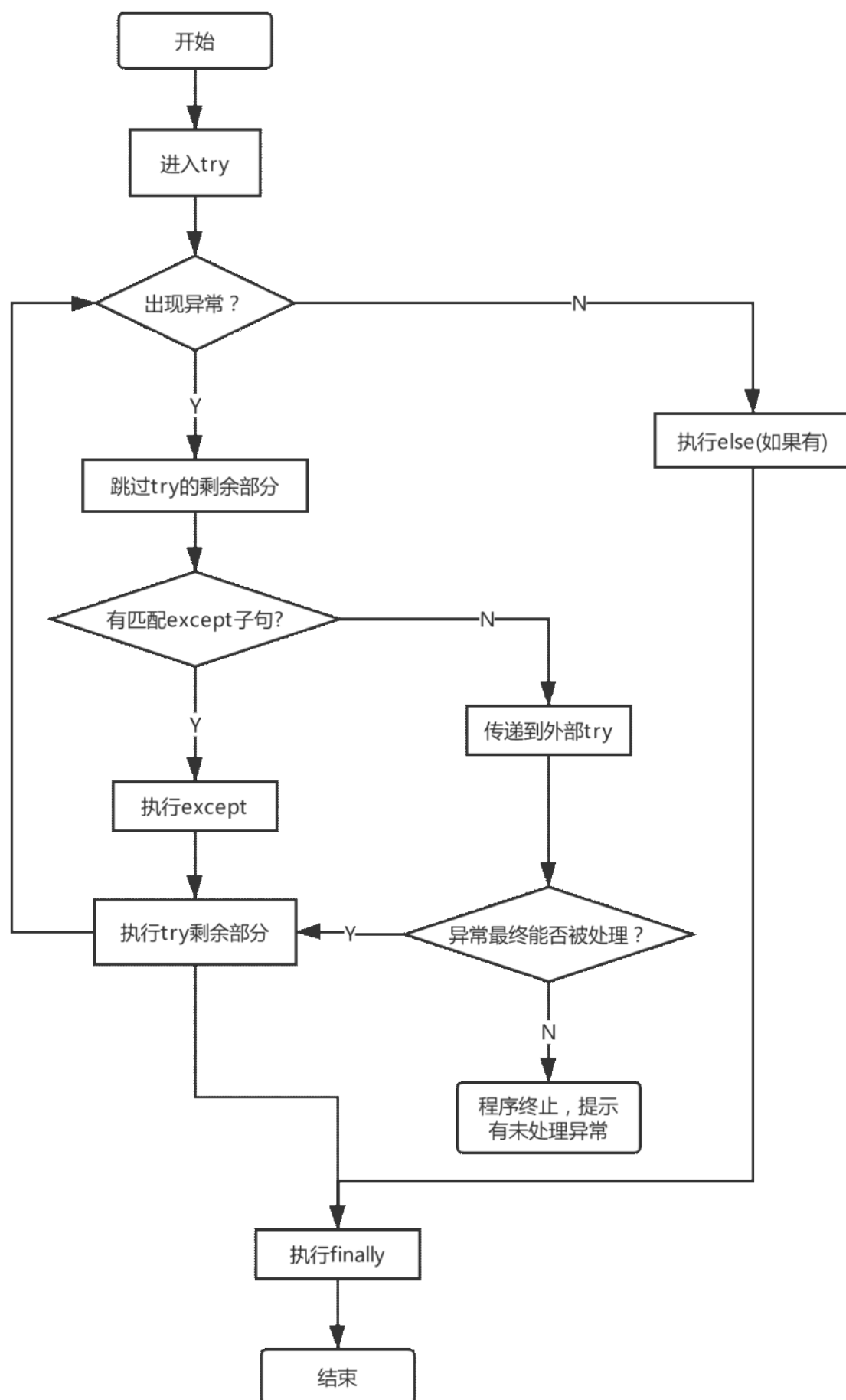
如果有finally语句块，在任何情况下，会在最后执行finally语句块。

```
1 try:
2     print('try...')
3     r = 10 / 0
4     print('result:', r)
5 except ZeroDivisionError as e:    #捕获ZeroDivisionError错误
6     print('except:', e)
7 finally:
8     print('finally...')
9 print('END')
10
11
12 '''
13 输出
14 try...
15 except: division by zero
16 finally...
17 END
18 '''
```

except语句块可以定义多个，以捕获不同类型的错误。

如果没有错误发生，可以在except语句块后面加一个else，当没有错误发生时，会自动执行else语句。

用一个流程图来描述Python的异常处理流程：



Python的错误也是一种class，所有的错误类型都继承自**BaseException**。



捕获错误类的时候，**捕获父类会涵盖其所有的子类**。

## 调用栈：

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。如以下的错误信息：

```
1 $ python3 err.py
2 Traceback (most recent call last):
3   File "err.py", line 11, in <module>      #11行出错导致程序退出
4     main()                                #出错原因是main()调用
5   File "err.py", line 9, in main          #9行的错误导致11行出错
6     bar('0')
7   File "err.py", line 6, in bar
8     return foo(s) * 2
9   File "err.py", line 3, in foo           #错误源头在第3行
10    return 10 / int(s)                    #该语句出错
11 ZeroDivisionError: division by zero      #错误类型是ZeroDivisionError
```

**出错的时候，一定要分析错误的调用栈信息，才能定位错误的位置。**

## 记录错误：

如果捕获了错误但不想在程序执行过程中处理错误，可以先将错误堆栈记录下来，同时让程序继续执行下去，直到运行完成，或有没能被捕获的错误使程序强制结束。

这一流程可以用Python内置的 logging 模块实现。

使用 logging.exception(e) 方法，可以让程序**打印完错误信息后**继续执行并正常退出。

还可以通过配置将错误信息记录到日志文件中。

## 抛出错误：

捕获一个错误，就是捕获到该错误类class的一个实例。

错误实例不会凭空产生，而需要在错误发生时有意创建一个实例并抛出。

除了Python内置的函数已定义的抛出错误，我们也可以用 raise 语句手动编写错误的抛出。

```
1 # err_raise.py
2 class FooError(ValueError):
3     pass
4
5 def foo(s):
6     n = int(s)
7     if n==0:
8         raise FooError('invalid value: %s' % s)    #raise语句抛出错误
9     return 10 / n
10
11 foo('0')
```

# 调试

2019年9月12日 14:22

调试一个程序，最重要的就是知道可能有问题的变量在某个时候的值。最简单粗暴的方法就是在关键位置写上 `print()`，把可能有问题的变量打印出来（我确实是这么干的……）。

这个方法虽然简单，但是最大的坏处是调试完之后还得删掉，如果`print`太多，输出信息就很复杂，删起来也不容易。

因此，应当使用**断言**进行调试。

## 断言(assert):

断言的概念在命令式计算中讲过，通过下断言，可以表达“在该处，如果不是断言中所描述的这样，之后必定会出错”。

```
1 def foo(s):
2     n = int(s)
3     assert n != 0, 'n is zero!'    #断言此处n不等于0，否则提示信息
4     return 10 / n
5
6 def main():
7     foo('0')
```

断言失败时，就会抛出 `AssertionError` 错误。

调试完成后要关闭断言不必一个个删去，只需要在启动Python解释器时用`-O`（大写字母O）参数即可关闭`assert`。

## logging:

与断言相比，`logging`不会抛出错误，且可以将错误信息输出到文件。

## pdb:

`pdb`是Python Debugger。

启动`pdb`的方式非常简单，只需要在启动解释器时以参数 `-m pdb` 运行即可（非侵入式方法）。

命令	解释
<b>break</b> 或 <b>b</b> 设置断点	设置断点
<b>continue</b> 或 <b>c</b>	继续执行程序
<b>list</b> 或 <b>l</b>	查看当前行的代码段
<b>step</b> 或 <b>s</b>	进入函数
<b>return</b> 或 <b>r</b>	执行代码直到从当前函数返回
<b>exit</b> 或 <b>q</b>	中止并退出
<b>next</b> 或 <b>n</b>	执行下一行
<b>pp</b>	打印变量的值
<b>help</b>	帮助

 [10分钟教程掌握Python调试器pdb - 知乎](#)

IDE：

调试当然还是用IDE比较方便。

比较好用的Python IDE就是VS Code和PyCharm。

虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

# 单元测试

2019年9月12日 16:57

## 编写单元测试：

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

把针对一个单元的一些测试用例放到一个测试模块里，就是一个完整的单元测试。

为了编写单元测试，我们需要引入Python自带的unittest模块。（import unittest）

编写一个类，继承自unittest.TestCase类：

```
1 import unittest
2
3 from mydict import Dict
4
5 class TestDict(unittest.TestCase):
6
7     def test_init(self):
8         d = Dict(a=1, b='test')
9         self.assertEqual(d.a, 1)
10        self.assertEqual(d.b, 'test')
11        self.assertTrue(isinstance(d, dict))
12
13    def test_key(self):
14        d = Dict()
15        d['key'] = 'value'
16        self.assertEqual(d.key, 'value')
17
18    def test_attr(self):
19        d = Dict()
20        d.key = 'value'
21        self.assertTrue('key' in d)
22        self.assertEqual(d['key'], 'value')
23
24    def test_keyerror(self):
25        d = Dict()
26        with self.assertRaises(KeyError):
27            value = d['empty']
28
29    def test_attrerror(self):
30        d = Dict()
31        with self.assertRaises(AttributeError):
32            value = d.empty
```

测试类中以test开头的方法就是用于测试的方法。对每一类测试都要编写一个test\_xxx()方法。

只要对测试应当返回的结果进行断言，就可进行测试。

若要用断言进行测试，只需要调用TestCase类内置的方法 assertEqual()（针对输出）或 assertRaises()（针对抛出异常），如下：

```
1 self.assertEqual(abs(-1), 1) # 断言函数返回的结果与1相等
2
3 with self.assertRaises(AttributeError):
4     value = d.empty
```

## 运行单元测试：

编写好单元测试类之后，只要在命令行通过参数 `-m unittest` 就可直接运行单元测试。

## setUp, tearDown：

可以在单元测试中编写两个特殊的`setUp()`和`tearDown()`方法。这两个方法会**分别在每调用一个测试方法的前后分别被执行**。

`setUp()`和`tearDown()`方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在`setUp()`方法中连接数据库，在`tearDown()`方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码。

# 文档测试

2019年9月12日 16:57

Python内置的文档测试 (doctest) 模块可以直接提取注释中的代码并执行测试。

```
1 # mydict2.py
2 class Dict(dict):
3     '''
4     Simple dict but also support access as x.y style.
5
6     >>> d1 = Dict()
7     >>> d1['x'] = 100
8     >>> d1.x
9     100
10    >>> d1.y = 200
11    >>> d1['y']
12    200
13    >>> d2 = Dict(a=1, b=2, c='3')
14    >>> d2.c
15    '3'
16    >>> d2['empty']
17    Traceback (most recent call last):
18    ...
19    KeyError: 'empty'
20    >>> d2.empty
21    Traceback (most recent call last):
22    ...
23    AttributeError: 'Dict' object has no attribute 'empty'
24    '''
25    def __init__(self, **kw):
26        super(Dict, self).__init__(**kw)
27
28    def __getattr__(self, key):
29        try:
30            return self[key]
31        except KeyError:
32            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)
33
34    def __setattr__(self, key, value):
35        self[key] = value
36
37 if __name__ == '__main__':
38     import doctest
39     doctest.testmod()
```

# IO编程

2019年9月27日 19:30

IO即输入、输出。

其中，流（stream）的概念非常重要。要做到输入输出，就需要两个流。

在IO之中，有一个主要问题：外设与CPU、内存的速度严重不匹配。为了解决这个问题，有两种解决方法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为**同步IO**；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为**异步IO**。

显然，异步IO效率更高，但实现起来也比较复杂。

本章讨论同步IO模式。

# 文件读写

2019年9月27日 20:40

## 读文件：

Python内置的open()函数用于打开文件。

```
>>> f = open('/Users/michael/test.txt', 'r')    # 'r'表示只读模式
```

对打开的文件，可以使用read()方法读取文件全部内容作为一个str。

关闭文件时，使用close()方法。

由于在读写时只要出现错误，后面的f.close()就没法关闭文件，所以一般用try打开文件。

更简洁的方法是使用with语句：

```
1 with open('/path/to/file', 'r') as f:
2     print(f.read())    #甚至无需写close()
```

对于比较大的文件，不便一次读入时，可以使用read(size), readline(), readlines()等方法。

## file-like Object：

拥有read()方法的对象，在Python中统称file-like Object。

如file，字节流，网络流，自定义流.....

## 字符编码，二进制文件：

默认的读取文件是以UTF-8编码读取，如果要读取其他编码，需要给open()函数传入encoding参数。如果要打开二进制文件，则用'rb'模式打开文件。

## 写文件：

'w', 'wb'模式用于写打开的文件。但这实际上是覆盖原有内容的写入，如果要追加内容，应该使用'a' (append) 模式。

需要注意的是，**如果没有close，可能会写不完全。**



# StringIO和BytesIO

2019年9月27日 22:16

## StringIO：

StringIO相当于一个内存中的str型缓冲区。

创建： `f = StringIO( )` (括号内可以放入初始化数据)

`write( )`方法写入，`getvalue( )`方法或像读文件一样读取。

## BytesIO：

与StringIO类似，但操作的是二进制数据。

# 操作文件和目录

2019年9月27日 22:59

利用Python的os模块（以及其下的os.path模块），可以对文件和目录进行操作。

由于操作系统分为POSIX（包括Unix，Linux，OS X）和NT两种，某些函数是和操作系统相关的。

os.environ保存了操作系统中定义的环境变量。

要操作文件和目录，使用os.path模块。

# 序列化

2019年9月27日 23:14

变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization，marshalling，flattening等等，都是一个意思。序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python中用于序列化的模块是pickle模块。

`pickle.dumps()` 将任意对象序列化成一个个bytes以写入文件。

相反的，`pickle.loads()` 将bytes反序列化成对象。

## JSON：

序列化为JSON是在不同编程语言间传递对象的好方法。

JSON表示的对象就是标准的JavaScript语言对象，有如下对应关系：

JSON类型	Python类型
<code>{}</code>	<code>dict</code>
<code>[]</code>	<code>list</code>
<code>"string"</code>	<code>str</code>
<code>1234.56</code>	<code>int</code> 或 <code>float</code>
<code>true/false</code>	<code>True/False</code>
<code>null</code>	<code>None</code>

要将Python对象转为JSON很简单，只需要使用json模块的`dumps()`函数即可。

同样的，用`json.loads()`可以将JSON转为Python对象。

但是，实际上很多时候我们需要自己编写一个转换函数传入`json.dumps()`的default参数，才能将对象序列化为JSON。

# 进程和线程

2019年9月28日 0:08

多任务的实现有3种方式：

- 多进程模式；
- 多线程模式；
- 多进程+多线程模式。（由于复杂，一般用不到）

线程是最小的执行单元，而进程由至少一个线程组成。如何调度进程和线程，完全由操作系统决定，程序自己不能决定什么时候执行，执行多长时间。

多进程和多线程的程序涉及到同步、数据共享的问题，编写起来更复杂。

# 多进程

2019年9月28日 16:22

## 多进程:

操作系统中讲过，Unix/Linux中的fork()系统调用用于创建子进程。

fork调用一次，却在父进程、子进程中分别返回，父进程中的返回值是子进程的ID，子进程则返回0。因此，可以通过判断fork()的返回值判断当前程序是父进程还是子进程。

**进程可以通过getppid()得到其父进程的ID，通过getpid()得到自身的ID。**

Python在os模块中封装了常用的系统调用，也包括fork()，getpid()，getppid()等，因此可以方便地创建子进程。

然而，Windows并没有fork调用，以上的操作在Windows中行不通。

但是我们还有替代方案！

**multiprocessing模块**是一个跨平台的多进程支持模块，可以在Windows中使用。

这个模块用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()方法启动这个Process实例即可，这样创建进程比fork()还要简单。举例：

```
1 from multiprocessing import Process
2 import os
3
4 # 子进程要执行的代码
5 def run_proc(name):
6     print('Run child process %s (%s)...' % (name, os.getpid()))
7
8 if __name__ == '__main__':
9     print('Parent process %s.' % os.getpid())
10    p = Process(target=run_proc, args=('test',)) #p是Process类实例
11    print('Child process will start.')
12    p.start() #启动进程p
13    p.join() #join方法用于等待子进程结束再继续执行
14    print('Child process end.')
```

## 进程池:

当需要启动大量子进程时，就可以用进程池的方式批量创建子进程。

进程池在os模块中是Pool类，构造函数参数就是要创建的子进程数量。

如果总进程数超过了CPU核数，就会有进程要等前面的进程执行完成才能开始执行。

## 子进程:

subprocess模块用于单独控制子进程的输入输出。

其中重要的有subprocess.call()和subprocess.communicate()方法。

(这还是不仔细展开了。)

## 进程间通信:

进程间有多种通信方式。如管道, 消息队列, 信号量, 信号, 共享内存, 套接字。

multiprocessing模块中自然也有提供对应支持, 如Queue (消息队列) 类, Pipes (管道) 类等。

在Unix/Linux下, multiprocessing模块封装了fork()调用, 使我们不需要关注fork()的细节。

由于Windows没有fork调用, 因此, multiprocessing需要“模拟”出fork的效果, 父进程所有Python对象都必须通过pickle序列化再传到子进程去, 所以, 如果multiprocessing在Windows下调用失败了, 要先考虑是不是pickle失败了。

# 多线程

2019年9月28日 17:49

每个进程都有至少一个线程，默认启动的线程是主线程。

Python的线程是真正的POSIX Thread。

多线程功能使用**threading**模块。

类似于多进程，线程实际上是一个从参数传入执行函数的Thread类实例，对该类实例执行start()方法即执行。

## Lock：

多线程的危险之处在于多个线程其实共享着同一个进程的内存空间，当多个线程同时读写同一变量时会导致混乱。

因此引入线程锁机制，当资源被某个线程占用时，就同时获得了该资源的锁，其他线程由于没有该资源的锁，在线程运行期间只能等待。

**创建锁通过threading.Lock()实现。**

**由Lock()的首字母大写形式，可以看出锁其实也是一个类，上锁时需要创建一个Lock类实例。**

**获得锁和释放锁的方法分别是acquire()和release()。**

同时，要注意避免发生死锁问题。

## 多核CPU：

Python的解释器CPython在执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

**也就是说，CPython根本没有对多线程进行多核优化，实际上还是单核运算。**

**目前在Python中，只能通过多进程实现多核任务。**

# ThreadLocal

2019年9月28日 22:22

线程有共享的部分，也有自己的个性。

其中包括线程ID，寄存器组的值，线程的堆栈，错误返回码，信号屏蔽码，线程优先级等等。这些东西是线程自己的局部变量。

ThreadLocal是一种特殊的全局变量，对每个线程而言是局部变量，每个使用这种变量的线程都拥有独立的变量副本，线程之间不会互相影响。

Python中使用threading模块中的 **threading.local()** 方法创建ThreadLocal对象。

这个对象刚创建时是没有额外方法、属性的，相当于一个object类实例。

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。



# 进程 vs. 线程

2019年10月2日 15:11

多任务一般采用Master-Worker模式，即一个Master（主进程/线程）负责分配任务，多个Worker负责执行任务。

多进程模式的稳定性较高，一个Worker出现故障不会影响其他进程，而Master只负责分配任务一般不会出现故障。

多线程虽然会比多进程快一些（因为共享内存？），但不够稳定，一个线程崩溃就可能导致整个进程崩溃。

多进程+多线程又太过复杂！

## 计算密集型 vs. IO密集型：

计算密集型：

大量计算，消耗CPU资源，同时进行的数量应当等于CPU的核心数。

代码运行效率至关重要，不适合用Python写，最好使用C语言。

IO密集型：

网络、磁盘IO占多，CPU占用较少，大部分时间都是在等待IO操作完成。

任务越多，CPU的效率越高（当然，也有上限）。

**常见的大多数任务都是IO密集型任务。**

适合用开发效率高，代码量少的脚本语言，如Python。

## 异步IO：

如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

对应到Python语言，单线程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

# 分布式进程

2019年10月2日 16:50

在多线程与多进程之间应该优先选择多进程。多进程不仅更加稳定，还可以分布到多台机器上。

multiprocessing模块的子模块managers子模块对分布式进程提供了支持。

分布式进程通过一个可由网络访问的Queue来传递任务和接收结果。

# 正则表达式

2019年10月2日 17:19

**正则表达式是用来匹配字符串的强力武器。**

## 基本匹配：

\d: 匹配一个数字 (digital)

\w: 匹配一个数字或字母 (word)

\s: 匹配一个空白字符

.: 匹配一个任意字符

R\*: 匹配任意个R (包括0个)

R+: 匹配至少一个R

R?: 匹配0或1个R

R{n}: n个R

R{n,m}: n-m个R

## 进阶：

[...]: 定义字符集, 如 [0-9a-zA-Z\_] 可以匹配一个数字、字母或者下划线

[^...]: 补充字符集, 即匹配字符集之外的字符

A|B: 匹配A或B

^: 行的开头

\$: 行的结尾

## re模块：

re模块用于提供正则表达式支持。

由于Python字符串中 '\\' 用于转义, 正则表达式中 '\\' 也用于转义, 则要匹配一个 '\\' 需要在正则表达式中写 '\\\\', 这很麻烦, 故建议使用原生字符串 (r前缀表示raw string, 其中字符不转义, 仅作为普通字符) 书写正则表达式。

**正则表达式匹配使用 re.match() 方法。第一个参数是正则表达式, 第二个参数是需要匹配的字符串。如果匹配成功, 返回一个Match对象, 否则返回None。**

由于None会被if语句认为False, 可以直接用if re.match(...)判断是否匹配。

## 切分字符串：

re模块的re.split()方法比字符串的split方法更加高级, 可以使用正则表达式匹配分割符。

**split(pattern, string, maxsplit=0, flags=0)**

## 分组：

用小括号 ( ) 括起来的表示一个分组 (Group) 。

如果在正则表达式中定义了分组，对返回的Match对象使用group( )方法就可以提取其中的子串。group(0)表示原始字符串，group(1)、group(2).....表示第1、2、.....个子串。

## 贪心匹配：

正则匹配默认是贪心匹配，即尽可能多地匹配字符。

如果不要贪心匹配，则使用R\*?，R+?，R??即可。

## 编译：

当我们在Python中使用正则表达式时，re模块内部会干两件事情：

1. **编译正则表达式**，如果正则表达式的字符串本身不合法，会报错；
2. 用编译后的正则表达式去匹配字符串。

对于要重复使用的正则表达式，出于效率的考虑，我们可以**预编译**该正则表达式。

```
1 >>> import re
2 # 编译:
3 >>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')
4 # 使用:
5 >>> re_telephone.match('010-12345').groups()
6 ('010', '12345')
7 >>> re_telephone.match('010-8086').groups()
8 ('010', '8086')
```

编译生成的是Regular Expression对象，调用对应的方法时不用给出正则字符串。