(十一) Python魔法方法

0x01 `__init__`

构造方法是我们使用频率最高的魔法方法了，几乎在我们定义类的时候，都会去定义构造方法，它的主要作用就是在初始化一个对象时，定义这个对象的初始值。

0x02 `__new__`

这个方法我们一般很少定义，不过我们在一些开源框架中偶尔会遇到定义这个方法的类。实际上，这才是“真正的构造方法”，它会在对象实例化时第一个被调用，然后再调用 `__init__`，它们的区别主要如下：

- `__new__` 的第一个参数是cls，而 `__init__` 的第一个参数是self
- `__new__` 返回值是一个实例，而 `__init__` 没有任何返回值，只做初始化操作
- `__new__` 由于是返回一个实例对象，所以它可以给所有实例进行统一的初始化操作

0x03 `__del__`

`__new__` 和 `__init__` 是对象的构造器， `__del__` 是对象的销毁器，它并非实现了语句 del x (因此该语句不等同于 `x.__del__()`)。而是定义了当对象被垃圾回收时的行为。 当对象需要在销毁时做一些处理的时候这个方法很有用，比如 socket 对象、文件对象。但是需要注意的是，当Python解释器退出但对象仍然存活的时候， `__del__` 并不会 执行。 所以为了保险起见，当我们在对文件、socket进行操作时，要想安全地关闭和销毁这些对象，最好是在try异常块后的finally中进行关闭和释放操作。

0x04 `__getattr__`

这个方法只有在访问某个不存在的属性时才会被调用，看上面的例子，由于gender属性在赋值时，忽略了此字段的赋值操作，所以此属性是没有被成功赋值给对象的。当访问这个属性时， `__getattr__` 被调用，返回unknown

0x05 引用

(十二)Python深/浅拷贝

0x01 赋值

```
a = [1,3,4]
b=a
b.append(2)
print a
>>>>[1, 23, 4, 1233]
print id(a),id(b)
>>>> (4519752408, 4519752408) #c = a 表示 c 和 a 指向相同的地址空间，并没有创建新的对象。也就是旧瓶装新酒，值虽然变了，但是地址空间没有变。
```

0x02 浅拷贝、深拷贝

“浅拷贝”即是指创建一个新的对象，其内容是原对象中元素的引用。这句话理解就是我买了一个新房子(创建一个新地址空间),但是我用的生活用品却还是拿旧家的生活用品(引用旧的元素)

深拷贝”是指创建一个新的对象，然后递归的拷贝原对象所包含的子对象。深拷贝出来的对象与原对象没有任何关联。这种拷贝就是房子是新的，生活用品是新的。虽然看上去值是一样的，但是没有半毛钱关系。

```
import copy
a =[1,2,3,[2,3,4]]
b = copy.copy(a)
c =copy.deepcopy(a)
print id(a),id(b),id(c)
>>>4519891024 4519290424 4519754280

for i,j in zip(a,b):
    print id(i),id(j)
>>>>
140574536521080 140574536521080
140574536521056 140574536521056
140574536521032 140574536521032
4519478408 4519478408
```

```
a 和 b 指向内存中不同的 list 对象，但它们的元素却指向相同的 int 对象。这就是浅拷贝。
for i,j in zip(a,c):
    print id(i),id(j)
>>>
140574536521080 140574536521080
140574536521056 140574536521056
140574536521032 140574536521032
4519478408 4519755288
a 和 b 指向内存中不同的 list 对象，且它们的元素指向不同的 int 对象。这就是深拷贝。

```

问:
为什么使用了深拷贝，前面几条a和b中元素的id还是一样呢？但是最后一条输出的内存地址却不一样?
答:
这是因为对于不可变对象(比如字符串，元组)，当需要一个新的对象时，python可能会返回已经存在的某个类型和值都一致的对象的引用。而且这种机制并不会影响 a 和 b 的相互独立性，因为当两个元素指向同一个不可变对象时，对其中一个赋值不会影响另外一个。而最后一个输出不一致就是由于列表是可变对象。

0x03 引用

赋值、浅拷贝、深拷贝的区别?

(十三)Python垃圾回收

0x01 简介

python的垃圾回收以引用计数为主，标记-清除和分代收集为辅。即当Python的某个对象的引用计数(也就是引用这个对象的次数)降为0时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾了。比如某个新建对象，它被分配给某个引用，对象的引用计数变为1。如果引用被删除，对象的引用计数为0，那么该对象就可以被垃圾回收。例如:

```
a = [1,2,3,4,5]
print(hex(id(a)))
>>> 0x10c8b51b8 分配给他内存了，但是如果这个a一直不用就是占着坑没事做了。
del a

```

然而，减肥是个昂贵而费力的事情。垃圾回收时，Python不能进行其它的任务。频繁的垃圾回收将大大降低Python的工作效率。如果内存中的对象不多，就没有必要总启动垃圾回收。所以，Python只会在特定条件下，自动启动垃圾回收。当Python运行时，会记录其中分配对象(object allocation)和取消分配对象(object deallocation)的次数。当两者的差值高于某个阈值时，垃圾回收才会启动。因此垃圾被回收不是“立即”的，由解释器在适当的时机，将垃圾对象占用的内存空间回收。

0x02 查看引用次数

我们可以使用sys包中的getrefcount()，来查看某个对象的引用计数。需要注意的是，当使用某个引用作为参数，传递给getrefcount()时，参数实际上创建了一个临时的引用。因此，getrefcount()所得到的结果，会比期望的多1。

```
from sys import getrefcount
a = [1, 2, 3]
print(getrefcount(a))
b = a
print(getrefcount(b))
由于上述原因，两个getrefcount将返回2和3，而不是期望的1和2。
```

那对象的引用计数在什么情况下增加呢?

- 对象被创建时
- 或被作为参数传递给函数（新的本地引用）
- 或成为容器对象的一个元素

那对象的引用计数在什么情况下减少呢?

- 一个本地引用离开了其作用范围。
- 对象的别名被显式的销毁。例如使用 del 语句
- 对象被从一个窗口（容器）对象中移除
- 窗口（容器）对象本身被销毁

Python的一个容器对象(container)，比如表、词典等，可以包含多个对象。实际上，容器对象中包含的并不是元素对象本身，是指向各个元素对象的引用。

0x03 引用计数回收的缺点

引用计数最大缺陷就是循环引用的问题。循环引用就是两个对象相互引用，但是没有其他变量引用他们；例如:

```
list1 = []                  # list1的引用计数为1
list2 = []                  # list2的引用计数为1
list1.append(list2)          # list2的引用计数加1，变为2
list2.append(list1)          # list1的引用计数加1，变为2
del list1                   # list1的引用计数减1，变为1
del list2                   # list2的引用计数减2，变为1
```

list1和list2相互引用后，引用计数都加1。用del删除了用于引用的变量名后，按理说现在不存在其他对象对它们的引用，list1和list2指向的内存应该被回收。但由于list1与list2的引用计数也仍然为1，所占用的内存永远无法被回收，就会导致内存泄漏。为了解决循环引用问题，Python引入了标记-清除和分代回收两种GC机制。

0x04 引用

gc模块–Python内存释放

(十四) Python正则表达式

0x01简介

正则表达式就是对字符串检索匹配和处理的一种方法，依次拿出表达式和文本中的字符比较，如果每一个字符都能匹配，则匹配成功；一旦有匹配不成功的字符则匹配失败。

0x02 语法

1.元字符字符集 `[]`

```
作用:用来匹配一个方括号里面的字符,
[1-9a\-g]或者[-1-9ag] 匹配1-9、a、-、g的一个字符
[^dot^a]     匹配除了d、o、t、a ^的一个字符。

\d   匹配任何十进制数；它相当于类 [0-9]。
\D   匹配任何非数字字符；它相当于类 [^0-9]。
\s   匹配任何空白字符；它相当于类   [ \t\n\r\f\v]。
\S   匹配任何非空白字符；它相当于类 [^ \t\n\r\f\v]。
```

```
\w   匹配任何字母数字字符；它相当于类 [a-zA-Z0-9_]。
\W   匹配任何非字母数字字符；它相当于类 [^a-zA-Z0-9_]。
这样特殊字符都可以包含在一个字符类中。如，[\s,.]字符类将匹配任何空白字符或","或"."。
```

2.数量词(即对前面的字符或者分组(...)匹配的个数进行指定)
？（重复零或者1次）

- (重复任意次，即包括0次)
- （重复任意次，但至少要有1次）
  {4}
  {4，9}
  重复多个字符分组

```
ca*t      可以匹配ct cat caaat 等
ca?t     可以匹配ct和cat
ca+t     可以匹配cat  caat
ca{4}t    caaaat
```

0x03 re库API

re库API中，一般都有flags参数，通过该参数指定正则表达式选项。传递时一般使用简写，比如开启DOTALL和MULTILINE使用 `re.I|re.M`

```
A  ASCII        使\w\W\b\B\d\D匹配ASCII字符
I  IGNORECASE   忽略大小写
L  LOCALE       使\w\W\b\B匹配本地字符集
M  MULTILINE    多行模式，"^" 匹配每行开头，"$"匹配每行结尾
S  DOTALL       "." 匹配所有字符，包括"\n"
X  VERBOSE      详细模式，忽略空白可以加入注释
U  UNICODE      使\w\W\b\B\d\D匹配unicode字符集
```

0x04 re模块

- re库对于很多函数，例如match，都提供了两种调用方式；方法一是直接使用re对象，通过re库调用，将正则表达式作为参数; 方法二先用complie将正则表达式字符串形式编译为Pattern实例，再使用Pattern实例处理文本并获得匹配结果。(match、find、findall)；

区别:方法二在正则表达式会被多次使用时会减少重复编译花费的时间。但实际上即使不使用compile，re库也会缓存一些近期使用过的正则表达式编译结果，所以这里不用太担心两者的效率差别。

```
# use compile
p = re.compile(r'ab*')
r = p.match('abcc')
print r.group(0)
ab
# don't use compile
r = re.match(r'ab*', 'abcc')
print r.group(0)
ab

```

- 匹配对象
  所谓匹配对象，就是match,search,findall等搜索操作返回的结果对象。
  search和match的作用是相近的，都是在字符串中匹配一个模式串，唯一的不同是match从首字符开始搜索，而search不要求。search在正则前加^可以达到和match相同的作用。

```
r = re.search(r'([a-z]+) (?P<digit>[\d]+)', 'name id 123')
print r
r = re.match(r'([a-z]+) (?P<digit>[\d]+)', 'name id 123')
print r
r = re.search(r'^([a-z]+) (?P<digit>[\d]+)', 'name id 123')
print r

# output
<_sre.SRE_Match object at 0x016954A0>
None
None
```

findall && finditer
search的加强版，findall返回所有结果是个列表，finditer返回一个迭代器。

```
p = re.compile(r'\b[\d]+\b')
r = p.findall('id 123, age 20.')
print r
r = p.finditer('id 123, age 20')
print [s.group(0) for s in r]

#out put
```

```
['123', '20']
['123', '20']
```

0x05 工具

RegexBuddy
Kodos

0x06 引用

正则表达式操作指南
30分钟入门正则表达式

语法	说明	表达式实例	完整匹配的字符串
	字符		
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade
	预定义字符集（可以写在字符集[...]中）		
\d	数字：[0-9]	a\dc	a1c
\D	非数字：[^\d]	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符：[^\s]	a\Sc	abc
\w	单词字符：[A-Za-z0-9_]	a\wc	abc
\W	非单词字符：[^\w]	a\Wc	a c
	数量词（用在字符或(...)之后）		
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	

边界匹配（不消耗待匹配字符串中的字符）			
^	匹配字符串开头。 在多行模式中匹配每一行的开头。	^abc	abc
$	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	abc$	abc
\A	仅匹配字符串开头。	\Aabc	abc
\Z	仅匹配字符串末尾。	abc\Z	abc
\b	匹配\w和\W之间。	a\b!bc	a!bc
\B	[^\b]	a\Bbc	abc
逻辑、分组			
|	|代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果|没有被包括在()中，则它的范围是整个正则表达式。	abc|def	abc def
(...)	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式中的|仅在该组中有效。	(abc){2} a(123|456)c	abcabc a456c
(?P<name>...)	分组，除了原有的编号外再指定一个额外的别名。	(?P<id>abc){2}	abcabc
\<number>	引用编号为<number>的分组匹配到的字符串。	(\d)abc\1	1abc1 5abc5
(?P=name)	引用别名为<name>的分组匹配到的字符串。	(?P<id>\d)abc(?P=id)	1abc1 5abc5
特殊构造（不作为分组）			
(?:...)	(...)的不分组版本，用于使用'|'或后接数量词。	(?:abc){2}	abcabc
(?iLmsux)	iLmsux的每个字符代表一个匹配模式，只能用在正则表达式的开头，可选多个。匹配模式将在下文中介绍。	(?i)abc	AbC
(?#...)	#后的内容将作为注释被忽略。	abc(?#comment)123	abc123
(?=...)	之后的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	a(?=\d)	后面是数字的a
(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	a(?!\d)	后面不是数字的a
(?<=...)	之前的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	(?<=\d)a	前面是数字的a
(?<!...)	之前的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	(?<!\d)a	前面不是数字的a
(?(id/name) yes-pattern |no-pattern)	如果编号为id/别名为name的组匹配到字符，则需要匹配yes-pattern，否则需要匹配no-pattern。 |no-patern可以省略。	(\d)abc(?(1)\d|abc)	1abc2 abcabc

(十五) Python匿名函数

0x01 简介

lambda表达式是单行最小函数，它不需要定义函数名因此被称为匿名函数。如果你不想在程序中对一个函数使用两次，你也许会想用lambda表达式，它们和普通的函数完全一样。

注意点:

1.对于只调用一次的函数，写成lambda更简单，因为无需使用def定义一个新的函数，

lambda调用结束就会被销毁。
2.lambda已经隐含了return，它是一个表达式，而不是语句
3.lambda只能写单行表达式。

0x02 语法

lambda 参数列表：表达式
可以理解参数列表就是正常函数的参数，表达式就是对参数操作后的返回值。
比如

```
dota = lambda x,y:x+y   #创建一个函数dota

print dota(3,4)
等价于
def dota(x,y):
    return x+y

print dota(3,4)
等价于
def dota(x,y):return x+y

print dota(3,4)

注意:
lambda只能写单行表达式如果遇到需要if else即:
b = lambda x: x+1 if x>10 else x-1

如果没有参数即:
a= lambda:1

print a()
```

(十六) Python装饰器

0x01 简介

装饰器就是我们希望增强一个函数，但是又不希望修改那个函数代码，也就是说在代码运行期间动态增加功能的方式叫做装饰器。装饰器其实也就是一个函数，一个用来包装函数的函数，返回一个修改之后的函数对象。经常被用于有切面需求的场景，较为经典的有插

入日志、性能测试、事务处理等。通常使用装饰器可以抽离出大量与函数功能本身无关的代码并继续重用。

0x02 案例

```
正常情况如果复用输出logging的话不想在函数里面多写，就重新定义一个函数专门用来输出日志的，然后调用。
def use_logging(func):
    logging.warn("%s is running" % func.__name__)
    func()

def bar():
    print 'i am bar'
def foo():
    print 'i am foo'

f =use_logging(bar)
f2 = use_logging(foo)
```

函数use_logging就是装饰器，它把真正的业务方法func包裹在函数里面，看起来像bar被use_logging装饰了。

```
def use_logging(func):

    def wrapper(*args, **kwargs):
        logging.warn("%s is running" % func.__name__)
        return func(*args)
    return wrapper

@use_logging
def foo():
    print("i am foo")

@use_logging
def bar():
    print("i am bar")

bar()
foo()
```

这样我们就可以省去bar = use_logging(bar)这一句了，直接调用bar()即可得到想要的结果。如果我们有其他的类似函数，我们可以继续调用装饰器来修饰函数，而不用重复修改

函数或者增加新的封装。这样，我们就提高了程序的可重复利用性，并增加了程序的可读性。 装饰器在Python使用如此方便都要归因于Python的函数能像普通的对象一样能作为参数传递给其他函数，可以被赋值给其他变量，可以作为返回值，可以被定义在另外一个函数内。

案例二:

```
@a
@b
@c
def f ():
等价于
f = a(b(c(f)))
```

0x03 引用

Python之装饰器知必会
理解Python装饰器

(十七) Python生成/解析器

0x01 简介

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器（Generator）。生成器最大的魅力在于，可以无限生成下去的可能性，比如写个切片斐波拉契数列的函数,通过next（）限制了死循环。

```
def foo():

    yield 1

    yield 2

    yield 3

print (list(foo()))
```

```

[1,2,3]

#如果一个函数中存在yield，则这个函数就是生成器函数。

```

```
L = (x * x for x in range(5))

sum(L)

30

#首先必须要有小括号，这也就是跟列表生成式的区别。使用结束生成器如果还想调用要再次生成下。

#其和列表生成式区别主要是一个输出列表，一个输出对象。

#当表达式的结果数量较少的时候， 使用列表生成式还好， 一旦数量级过大， 那么列表生成式就会占用很大的内存，
#而生成器并不是立即把结果写入内存，而是保存的一种计算方式，通过不断的获取，可以获取到相应的位置的值，

#所以占用的内存仅仅是对计算对象的保存。

```

3.6 ### 列表解析式

```
L = [x * x for x in range(5)]

[0,1,4,9,16,25]

sum(L)

30


#for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

[x * x for x in range(1, 11) if x % 2 == 0]

[4, 16, 36, 64, 100]

```

```
[m + n for m in 'ABC' for n in 'XYZ']
```

```
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']

#双层循环

```

```
#最后把一个list中所有的字符串变成小写:

L = ['Hello', 'World', 'IBM', 'Apple']

[s.lower() for s in L]

#列表生成式也可以使用两个变量来生成list:

d = {'x': 'A', 'y': 'B', 'z': 'C' }

#获取一个路径下所有文件和文件名

import os

[d for i in os.listdir('.')]

# 把列表中所有的字符串取出，并变成小写

L = ['ABC', 'BCD', 18, 'CDE',88]

[s.lower() for s in L if isinstance(s,str)]

#首先必须要有中括号，其次先运行后面，然后x*x，然后把值赋给L，进行下一次循环。

#写列表生成式时，把要生成的元素x * x放到前面，后面跟for循环，就可以把list创建出来

```

3.7 引用

迭代器

列表解析

(十八) Python高阶函数

0x01 简介

变量可以指向函数，所以函数名也是变量，是变量就能作为参数传入另一函数中，这种参数中带有函数参数的函数就称为高阶函数。比如ads()函数是求绝对值的函数，如下所示的add()则是一个高阶函数

```
>>> def add(a,b,f):
        return f(a)+f(b)

>>> add(3,6,abs)
9
>>>
```

0x02 Map/reduce

map()函数接收两个参数，一个是函数，一个是序列，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list返回，例如：计算每一元素的立方次幂

```
>>> def f(x):
        return x**3

>>> L = [3,4,5,6,7,8,1,2]
>>> map (f,L)
[27, 64, 125, 216, 343, 512, 1, 8]
>>>
```

立方次幂

我们也可以使用map()函数，计算任意复杂度的函数，把定义好的函数与序列传入map()就可完成，方便简洁。

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。当函数是做加法运算时与sum()函数功能相同。但是reduce最主要的作用则依赖于所传入的函数。

```
>>> def fn(x,y):
        return (x+1)*10 + y

>>> L = [2,4,3,1,5]
>>> reduce(fn,L)
35425
>>>
```

函数fn()自己随意写的

map() 、reduce()结合使用:

```
>>> def fun(x,y):
        return (x-1)*10 - y

>>> def charChangeInt(l):
        return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5':
5, '6': 6, '7': 7, '8': 8, '9': 9}[l]

>>> reduce(fun,map(charChangeInt,'37291'))
11599
>>>
```

0x03 filter

filter()函数与map()类似，但filetr用于过滤序列，接收一个函数与一个序列，他把传入的函数一次作用于序列中的每一元素，且是通过返回值的True还是False决定保留还是丢弃该元素。

```
>>> def f(x):
        return x % 3 == 1

>>> L = [2,4,6,7,8,9,30,10]
>>> filter(f,L)
[4, 7, 10]
>>>
```

过滤留下符合条件得元素

0x04 引用

Python 高阶/map/reduce/filter/sorted函数

(十九) Python多线程

0x01 简介

- 多线程：多线程即并发处理事务，子任务相互独立可以被分成多个执行逻辑。比如我破解一个网站密码，使用passwd=1和使用passwd=2是他们都是我即将要破解的密码，每个个要破解的密码就是一个子任务，他们相互独立，线程上可以同时进行破解加快尝试。因此它们的本质是异步，需要有多个并发任务，每个事物的运行顺序是不确定的，随机的，不可预测的，这样的编程任务是可以被分成多个执行流，每个流都有一个要完成的目标。
- 进程: 进程是程序的一次执行，每个进程都有自己的地址空间，内存，数据栈等。

- 线程:所有线程运行在同一个进程中，共享相同的运行环境，线程有开始 顺序执行 结束三部分。
- GIL: 首先就像CPU那样同时可以运行多个进程，内存可以存放多个程序，但是任意时刻只有一个程序在cpu运行；同理python解释器(python代码有python虚拟机也就是解释器来控制)在运行多个线程，任意时刻，只有一个线程在解释器中运行，而控制同一时刻只有一个线程在运行的锁机制就是GIL全局解释锁。
- 整个流程：python多线程受到GIL限制(任意时刻只有一个线程在运行)，因此并不是真正意义上多线程，只是飞快的进行线程之间切换，你所看到飞速的线程在跑其实就是飞速的进行切换线程。退出就是当一个线程结束了他就退出了，也可以用sys.exit()退出一个进程，但是你不可以直接杀掉一个线程。
- 多线程返回的字符串用print有时会错行，可以用sys模块中sys.stdout.write()方法可规范输出。

0x02 创建多线程

- 方法一: 创建线程要执行的函数，把这个函数传递进Thread对象里，让它来执行。

```
import threading,time

def func(key):
    print "Hello,world! - %s - %s\n" %(key, time.ctime())

def main():
    threads = []
    for i in xrange(5):                                                    #总线程数
        th = threading.Thread(target=func, name='dota',args=(i,)) #创建名为dota的新线程，args后面是个列表，也可以是[],如果是括号一定要有逗号。
        threds.append(th)
        th.start()
    for i in threads:  # 主线程中等待所有子线程退出
        print i        # <Thread(Thread-1, stopped 123145572798464)>
        i.join()
    print 'END'
```

- 方法二: 直接从Thread继承，创建一个新的子类，把线程执行的代码放到这个新的类中执行;创建线程对象后, 通过调用start()函数运行线程, 然后会自动调用run()方法。

```
class Task(threading.Thread)  #继承Thread类
    def __init__(self,queue):
```

```
        threading.Thread.__init__(self)#用来初始化父类的代码,python中self.method实质上是method(self), 从定义方法的时候就可以看出来。所以这句话就是在初始化这个对象的基类部分
        self._queue = queue
    def run(self):
        while not self._queue.empy():
            thing = self._queue.get()
            do(thing)

def main():
    threads=[]
    queue=Queue.Queue()
    for i in range(10):
        thread.append(Task(queue))
    for i in threads;
        i.start()
    for i in threads:
        i.join()

```

- 实例目录遍历

```
# -*- coding:utf-8 -*-
# !/usr/bin/env python

import requests
import sys
import Queue
import threading

class DirScan(threading.Thread):

    def __init__(self,queue):
        threading.Thread.__init__(self)
        self._queue=queue
    def run(self):
        while not self._queue.empty():
            url = self._queue.get()
            print url
            response = requests.head(url)
            if response.status_code == 200:
                sys.stdout.write(url+'\n')

def main():
    print "start scanning directories"
    threads=[]
```

```
    queue=Queue.Queue()
    with open('1.txt','r') as fr:
        for i in fr:
            queue.put("http://www.coco413.com/"+i.strip())
    print queue.qsize()
    for i in range(64):
        threads.append(DirScan(queue))
    for i in threads:
        i.start()
    for i in threads:
        i.join()

if __name__=="__main__":
    main()
```

0x03 线程锁

如果多个线程共同对某个数据(比如同时打开一个文本操作)修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。使用Thread对象的Lock和Rlock可以实现简单的线程同步，这两个对象都有acquire方法和release方法，对于那些需要
每次只允许一个线程操作的数据，可以将其操作放到acquire和release方法之间。如下：
多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表并打印。那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print"获得锁定了，那么就让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，释放锁以后，再让线程"set"继续。经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。因此总的一句话锁就是其中一个线程对xx进行操作时候，其他线程都不能对这个线程进行干预，直到我这个线程结束了之后才可以。

在threading模块中，定义两种类型的琐：threading.Lock和threading.RLock
这两种琐的主要区别是：RLock允许在同一线程中被多次acquire。而Lock却不允许这种情况。注意：如果使用RLock，那么acquire和release必须成对出现，即调用了n次acquire，必须调用n次的release才能真正释放所占用的琐。
如果只是简单的加锁解锁可以直接使用threading.Lock()生成锁对象, 然后使用acquire()和release()方法

```
```

```
threadLock = threading.Lock() # 实例化一个线程锁
    def process(self,i):
        try:
        threadLock.acquire()   #加锁
        print i                                #同时刻只有获取锁的那个线程可以执行这句话
        finally:
                threadLock.release()   #释放锁
    def scan(self):
        all_threads = []
        for i in range(10):
            t = threading.Thread(target=self.process,args=(i,))
            all_threads.append(t)
            t.start()
```

0x04 方法

- Threading模块对象:
  Thread 一个表示线程控制的类，这个类常被继承。
  Timer 定时器，线程在一定时间后执行
  activeCount()或者active_count() 返回当前线程对象Thread的个数
  enumerate() 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
  currentThread()或current_thread() 返回当前线程对象
  Lock() 返回一个新的锁对象
- Thread类的成员变量和函数如下
  start() 启动一个线程
  run() 线程执行体（定义县城的功能的函数，一般会被子类重写）
  name 线程名
  ident 线程ID
  daemon 是否守护线程
  isAlive()、is_alive() 线程是否存活(布尔标志，表示这个线程是否还在运行中)
  getName()、setName() Name的get&set方法（返回线程名字和设置线程名字）
  isDaemon() 返回线程daemon标志
  setDaemon() daemon的get&set方法(把县城的daemon标志设为daemonic【一定要在调用start()函数之前调用】)
  join([timeout]) 等待线程结束(程序挂起，直到线程结束；如果给了timeout，则最大堵塞timeout秒)；如果你的主线程除了等线程结束外，还有其他的事情要做(比如处理或等待其他的客户请求)，那就不用调用join()，只有在你要等待线程结束的时候才用调

用join(),
总的一句话就是等待线程结束了才会往下跑。

- 问题1:什么时候用join?

```
    def process(self,i):
        print i

    def scan(self):
        all_threads = []
        for i in range(10):
            t = threading.Thread(target=self.process,args=(i,))
            all_threads.append(t)
            t.start()
        print 'END'
```

比如这个正常情况下 END是最后输出，但是线程运行起来他不管了END输出随机夹在1，2，3，4等之间了，如果想要最后输出END怎么办呢，所以要在t.start()之后用到join()

```
    def scan(self):
        all_threads = []
        for i in range(10):
            t = threading.Thread(target=self.process,args=(i,))
            all_threads.append(t)
            t.start()
        for i in all_threads:
            i.join()
        print 'END'
```

- 问题2:什么时候需要用线程守护?
如果你设置一个线程为守护线程,就表示你在说这个线程是不重要的,在进程退出的时候,不用等待这个线程退出。如果你的主线程在退出的时候，不用等待那些子线程完成，那就设置这些线程的daemon属性。即，在线程开始（thread.start()）之前，调用setDeamon（）函数，设定线程的daemon标志。（thread.setDaemon(True)）就表示这个线程“不重要”。

```
import time,threading
def dota():
    print "start fun"
    time.sleep(2)
    print "end fun"
print "main thread"
t1 = threading.Thread(target=dota,args=())
```

```
#t1.setDaemon(True)
t1.start()
time.sleep(1)
print "main thread end"
#比如开启了守护线程，程序在主线程结束后，直接退出了。 导致子线程没有运行完。
#输出
main thread
start fun
main thread end
```

- 问题3:如何修改显示名字?

```
import time,threading
def dota():
    print "start fun"
    time.sleep(2)
    print "end fun"

for i in range(10):
    t1 = threading.Thread(target=dota,args=(),name="di{}ci".format(i))
    #t1.setName('di{}ci'.format(i))
    t1.start()
    #print t1.getName()
    print t1.name
    time.sleep(1)
    print "main thread end"
```

- 问题4:如何实现每隔一段时间就多线程调用一个函数?

```
import time,threading
def dota(i):
    print i
    time.sleep(2)

for i in range(10):
    t1 = threading.Timer(5,dota,[i])
    t1.start()
```

0x05 Queue

简介

在python中，多个线程之间的数据是共享的，多个线程进行数据交换的时候，不能够保证数据的安全性和一致性，所以当多个线程需要进行数据交换的时候，队列就出现了，队列

可以完美解决线程间的数据交换，保证线程间数据的安全性和一致性。

Queue模块实现了多生产者多消费者队列, 尤其适合多线程编程。Queue类中实现了所有需要的锁原语;Queue模块实现了三种类型队列:

- 普通的FIFO队列（Queue）即先进先出,第一加入队列的任务, 被第一个取出，专业点也就是生产者将数据依次存入队列，消费者依次从队列中取出数据。【排队】
- LIFO队列（LifoQueue）即后进先出，最后加入队列的任务, 被第一个取出。[洗盘子]
- 优先级队列（PriorityQueue）即优先级队列，保持队列数据有序, 最小值被先取出。

类

`Queue.Queue(maxsize = 0)`
构造一个FIFO队列,maxsize设置队列大小的上界, 如果插入数据时, 达到上界会发生阻塞, 直到队列可以放入数据. 当maxsize小于或者等于0, 表示不限制队列的大小(默认)
`Queue.LifoQueue(maxsize = 0)`
构造一LIFO队列,maxsize设置队列大小的上界, 如果插入数据时, 达到上界会发生阻塞, 直到队列可以放入数据. 当maxsize小于或者等于0, 表示不限制队列的大小(默认)
`Queue.PriorityQueue(maxsize = 0)`
#构造一个优先级队列,,maxsize设置队列大小的上界, 如果插入数据时, 达到上界会发生阻塞, 直到队列可以放入数据. 当maxsize小于或者等于0, 表示不限制队列的大小(默认). 优先级队列中, 最小值被最先取出

异常

```
`Queue.Empty`
#当调用非阻塞的get()获取空队列的元素时，引发异常
`Queue.Full`
#当调用非阻塞的put()向满队列中添加元素时，引发异常，其中queue.full 与 maxsize 大小对应
```

对象

```
Queue.qsize() 返回队列的大小
Queue.empty()  #如果队列为空，返回True,反之False
Queue.full()  #如果队列满了，返回True,反之False
Queue.put(item) #写入队列，timeout等待时间
Queue.put_nowait(item)  #相当queue.put(item, False)
Queue.get([block[, timeout]]) #获取队列，timeout等待时间
Queue.get_nowait()    #相当queue.get(False)
Queue.task_done()  #在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
Queue.join()  #实际上意味着等到队列为空，再执行别的操作
```

名称	说明
qsize()	用于返回队列当前的大小
empty()	返回队列是否为空
put(item [,block [,timeout]]	将 item 放入 Queue 尾部，item 必须存在，可以参数 block 默认为 True, 表示当队列满时，会等待队列给出可用位置，为 False 时为非阻塞，此时如果队列已满，会引发 queue.Full 异常。 可选参数 timeout，表示会阻塞设置的时间，如果队列无法给出放入 item 的位置，则引发 queue.Full 异常
get([block [,timeout]]	从队列中删除一条记录并将其返回。默认 block = True, timeout = None ，操作会在队列空时阻塞直到可以从中取出一条记录，如果 timeout 为正，则阻塞时间超时后，抛出 Empty 异常。如果 block=False ，则在队列为空时立即返回并抛出 Empty 异 常。
task_down()	通知上一个队列操作已经完成。消费者线程在 get() 条目后，在完成相应工作后，需要调用 task_done() 以告知队列处理任务已经完成，如果 task_down()调用次数超过队列中元素个数，则抛出 ValueError 异常。
join()	阻塞一直到队列中所有 items 都被 get() 并处理。
full()	当队列满的时候，返回 True，否则返回 False
put_nowait(item	等效于 put(item, block=False)
get_nowait()	等效于 get(item, block=False)

实例

```
import Queue
import threading
class DoRun(threading.Thread):
    def __init__(self,queue):
        threading.Thread.__init__(self)
        self._queue = queue
    def run(self):
        while not self._queue.empty():
            key = self._queue.get()
            print key
def main():
    threads = []
    threads_count = 10
    queue = Queue.Queue()
    for i in range(1,255):
        queue.put(i)
    for i in range(threads_count):
        threads.append(DoRun(queue))
    for i in threads:
        i.start()
    for i in threads:
        i.join()
```

```
if __name__ == '__main__':
    main()
```

0x06 引用

虫师多线程

(二十)Python开发工具

Pycharm

0x01 快捷键

```
command + b/command+鼠标 跳转到声明处
command + delete 删除当前行
command + ?  注释当前行
command + R  全局搜索
Shift  + 光标  选取内容
command + J  智能补全
Tab/Shift  + Tab  缩进
command + option +L 代码块对齐
control + Shift + R  执行脚本
command + option + R    Debug启动脚本
```

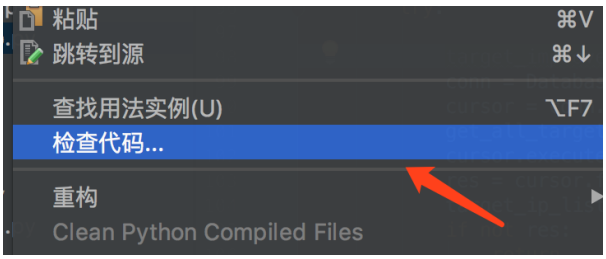
0x02 常用工具

- 中文汉化
- 添加格式化autopep8
- 对比文件
- 运行调试
- 添加模板

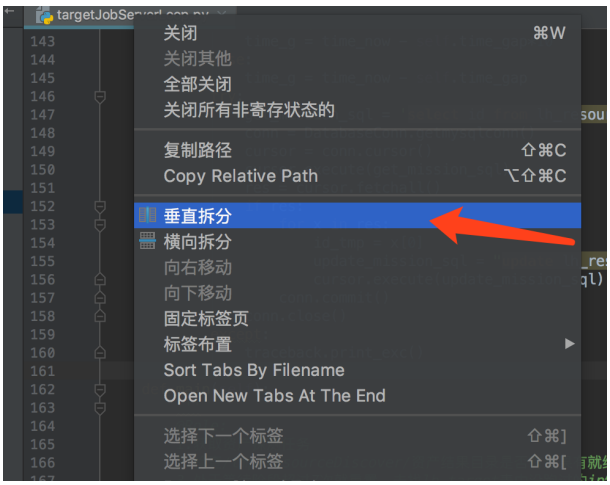
```
# -*- coding:utf-8 -*-
# !/usr/bin/env python
"""
@Time    : ${DATE} ${TIME}
@Author  :
@Desc    :
```

```
"""
import sys
reload(sys)
sys.setdefaultencoding("utf-8")

class MainLoop(object):

    def __init__(self):
        pass


    def main(self):
        pass
if __name__ == "__main__":
    main = MainLoop()
    main.main()
```

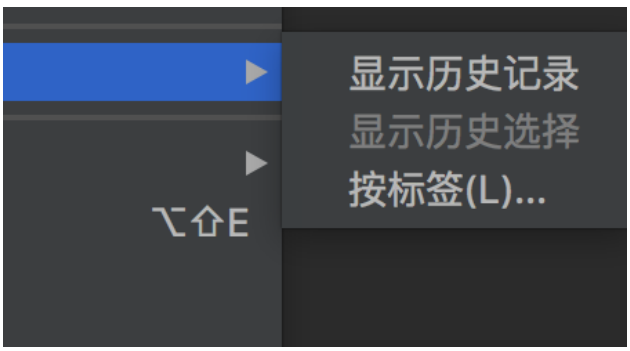
- 代码审查



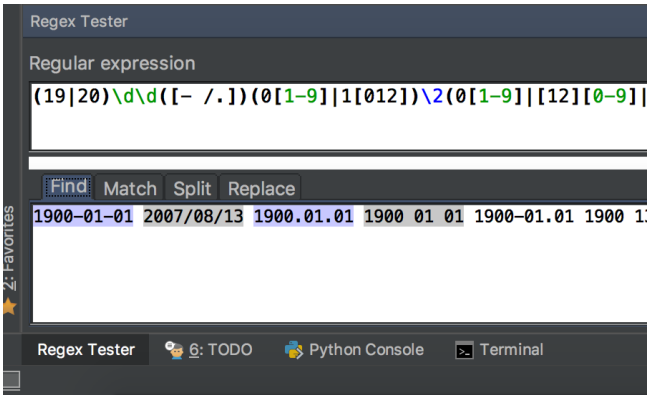
- 垂直分割



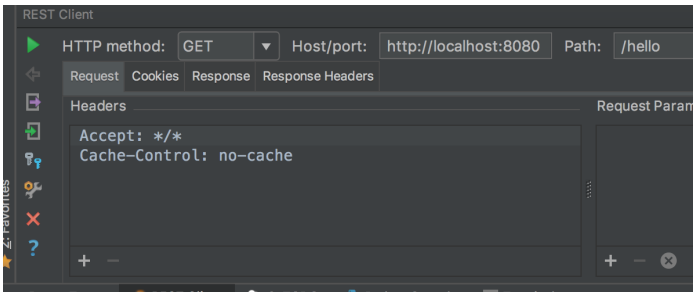
- 历史记录



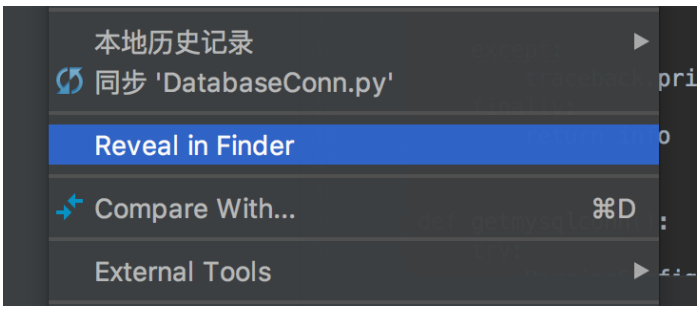
- 正则测试



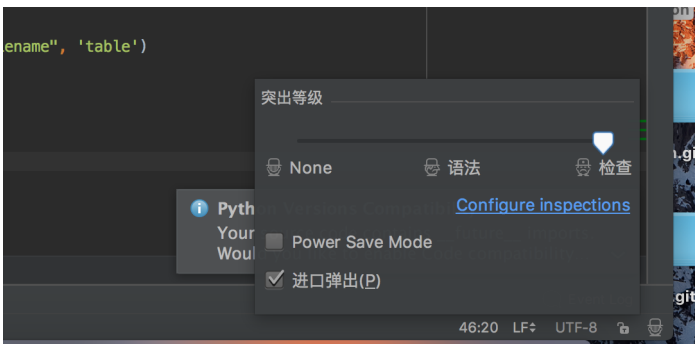
- Rest接口测试



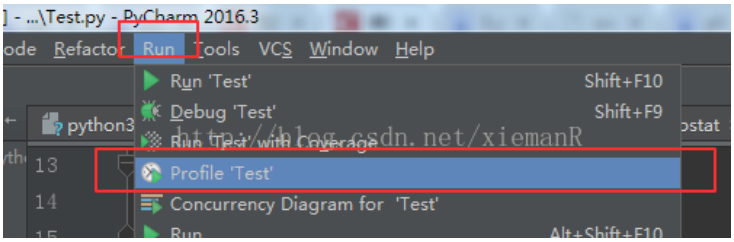
- 打开本地文件



- 取消波浪线



- 性能测试



Sublime Text 3

0x01 安装插件

- BracketHighlighter
- JsFormat(Ctrl+Alt+F)
- Emmet
- SublimeTmpl(ctrl+alt++shift+p)
- Pretty JSON(command+control+J)
- AutoPep8
- Anaconda
- AutoFileName
- Package Control
- SublimeCodeIntel

0x02 常用功能

- 分屏
  `command + option + 2`
- 编译python运行环境

```
{
    "cmd": ["/usr/bin/python", "-u", "$file"],
    "file_regex": "^[ ]*File \"(…*?)\", line ([0-9]*)",
    "selector": "source.python"
}
```

- 中文汉化

Ipython

- %time 测试性能时间

```
 %time a = [x for x in xrange(100)]
CPU times: user 78.8 ms, sys: 2.5 ms, total: 81.3 ms
Wall time: 80 ms
```

- %history 历史记录

- %run 运行