文章目录

显示有限的接口到外部

当发布python第三方package时，并不希望代码中所有的函数或者class可以被外部import，在__init__.py中添加__all__属性，
该list中填写可以import的类或者函数名，可以起到限制的import的作用，防止外部import其他函数或者类

1	#!/usr/bin/env python

```
2
        # -*-
coding:
utf-8 -*-
3
        from
base
        import
APIBase
4
        from
client
```

```
5
import Client
6
from decorator import interface, export, stream
7
from server
8
import Server
```

```
9
                from storage import Storage
10
                from util import (LogFormatter, disable_logging_to_stderr,
11
12
```

```
enable_logging_to_kids, info)

__all__ = [
    'APIBase',
    'Client',
    'LogFormatter',
    'Server',
```

'Storage',

'disable_logging_to_stderr',

'enable_logging_to_kids',

'export',

'info',

'interface',

	'stream']

with的魔力

with语句需要支持上下文管理协议的对象，上下文管理协议包含__enter__和__exit__两个方法. with语句建立运行时上下文需要通过这两个方法执行进入和退出操作.

其中上下文表达式是跟在with之后的表达式，该表示大返回一个上下文管理对象

1	# 常见with使用场景
2	with open("test.txt",

3	"r")
4	as my_file: # 注意, 是 __enter__()方法的返回值赋值给了 my_file,
	for line in my_file:

```
        print
            line
```

详细原理可以查看这篇文章，浅谈 Python 的 with 语句

知道具体原理，我们可以自定义支持上下文管理协议的类，类中实现__enter__和__exit__方法

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

```
3
classMyWith(object):
4
    def__init__(self):
5
        print "__init__method"
6
```

```
7   def__enter__(self):
8           print "__enter__ method"
9
            return self
```

```
10          # 返回对象给as后的变量
11  def __exit__(self, exc_type, exc_value, exc_traceback):
12          print "__exit__ method"
13
```

```
        if exc_traceback is None:
14
            print "Exited without Exception"
15
16
```

```
17                  return True
18              else:
19                  print "Exited with Exception"
20
```

```
21      return False
22
        def test_with():
23          with MyWith()
```

```
        as
my_with:
24
        print
25
"running
my_with"
26
        print
        "----
--分割线--
---"
27
```

```
        with MyWith()
28      as my_with:
29          print "running before Exception"
30          raise Exception
```

```
31
        print
32  "running after Exception"
33
    if __name__ == '__main__':
```

	test_with()

执行结果如下:

1	__init__
	method
2	
	__enter__
	method
3	

4	running my_with	
5	__exit__ method	
6	Exited	

7	without Exception
8	----------分割线----------
9	__init__ method
10	

	__enter__	
11	method	
12	running before	
	Exception	
13		
	__exit__	

method

14

Exited

15

with

Exception

16

Traceback (most recent call last):

17

File

18 "bin/python", line 34,

in

19

exec(compile(__file__f.read(), __file__,

"exec"))

	File "test_with.py", line 33, in test_with()	

File "test_with.py", line 28, in test_with raise Exception

	Exception

证明了会先执行__enter__方法，然后调用with内的逻辑，最后执行__exit__做退出处理，并且，即使出现异常也能正常退出

filter的用法

相对filter而言，map和reduce使用的会更频繁一些，filter正如其名字，按照某种规则过滤掉一些元素

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

```
3
lst = [1, 2, 3, 4, 5, 6]
4
5
```

```
6
        # 所有奇数都会返回True, 偶数会返回False被过滤掉
7
        print filter(lambda x: x %
8
        2 != 0, lst)
9
        #输出结果
```

	[1, 3, 5]

一行作判断

当条件满足时，返回的为等号后面的变量，否则返回else后语句

1	lst = [1,

2	2, 3]
3	new_lst = lst[0]
4	if lst is not None
5	else

6	None print new_lst # 打印结果 1	

装饰器之单例

使用装饰器实现简单的单例模式

```
1        # 单例装饰器
2
3
```

```
def singleton(cls):
    instances = dict()
    # 初始为空
```

```
4
def_singleton(*args, **kwargs):
5
    if cls
6
    not
    in instances:
7
    #如果不存在,则创建并放入字典
```

```
8
        instances[cls] = cls(*args, **kwargs)
9
    return instances[cls]
10
return _singleton
```

```
11
                @singleton
12
                classTest(object):
13
                    pass
14
```

```
if __name__ == '__main__':
    t1 = Test()
    t2 = Test()
```

```
18
        # 两者具有相同的地址
        print t1, t2
```

staticmethod装饰器

类中两种常用的装饰，首先区分一下他们

- **普通成员函数，其中第一个隐式参数为**对象
- classmethod装饰器，类方法(给人感觉非常类似于OC中的类方法)，其中第一个隐式参数为类
- staticmethod装饰器，没有任何隐式参数. python中的静态方法类似与C++中的静态方法

```
1
#!/usr/bi
```

```
n/env python
2
    # -*- coding: utf-8 -*-
3
class A(object):
4
    # 普通成员函数
```

```
5
    def foo(
            self,
        x)
6
        :
7
        print
            "executin
        g foo(%s,
        %s)" % (
8           self,
        x)
```

```
9
    @classmethod
        # 使用classmethod进行装饰
10
    def class_foo(cls, x):
11
        :
```

```
12
        print "executing class_foo(%s, %s)" % (cls, x)
13
    @staticmethod
14
        # 使用staticmethod进行装饰
15
```

```
    defstatic_foo(x):
16
    print "executing static_foo(%s)" % x
17
    deftest_three_method():
18
```

```
        :
19
obj =
20
        A()
21
        # 直接调用噗通的成员方法
22
```

obj.foo(

"para")

23 # 此处obj对象作为成员函数的隐式参数, 就是self

24

obj.class_foo(

"para")

25 # 此处类作为隐式参数被传入, 就是cls

```
26
A.class_foo(
"para")
27
#更直接的类方法调用
28
obj.static_foo(
"para")
29
# 静态方法并没有任何隐式参数,但是要通过对象或
```

者类进行调用

30

A.static_foo(

"para")

31

if __name_

32

_ ==

'__main__'

```
        :
33
test_three_method()
34
        # 函数输出
35
executing foo(<__main_
```

	_.	
	A object at 0x100ba4e10>, para)	
	executing class_foo(< class '__main__.A'>, para)	

	executing class_foo(< class '__main__.A'>, para)	
		executing static_foo (para)
		executing static_foo (para)

property装饰器

- 定义私有类属性

将property与装饰器结合实现属性私有化(更简单安全的实现get和set方法)

```
1
#python
内建函数
2
property(
fget=
None,
fset=
None,
fdel=
None,
doc=
```

	None)

fget是获取属性的值的函数,fset是设置属性值的函数,fdel是删除属性的函数,doc是一个字符串(like a comment).从实现来看，这些参数都是可选的

property有三个方法getter(), setter()和delete() 来指定fget, fset和fdel。 这表示以下这行

1	classStudent(object):
2	@property #相当于property.getter(score) 或者property(score)

```
3
    defscore(self):
4
        return self._score
5
    @score.setter #相当于score = property.setter(score)
6
```

```
7
                defscore(self, value):
8
                    if not isinstance(value, int):
9
                        raise ValueError(
10
```

```
                'score must be an integer!')
11
        if value <
            0
12
        or value >
            100:
13
        raise ValueErro
```

```
r(
'score must between 0 ~ 100!')
self._score = value
```

iter魔法

- 通过yield和__iter__的结合, 我们可以把一个对象变成可迭代的
- 通过__str__的重写, 可以直接通过想要的形式打印对象

```
1 #!/usr/bin/env python
```

2	# -*- coding: utf-8 -*-	
3	classTestIter(object):	
4	def__init__(self):	

```
5
                self.lst = [
                    1,
6
                    2,
                    3,
                    4,
7
                    5]


8
            defread(self):
```

```
9
        for ele in xrange(len(self.lst)):
10
            yield ele
11
    def__iter__(self):
12
```

```
        return self.read()
13
14  def __str__(self):
15      return ','.join(map(str, self.lst))
```

```
16
    __repr__ = __str__
17
    def test_iter():
18
        obj = TestIter()
19
```

```
        for num
20
        in obj:
21
            print num
22
    print obj
```

```
23
        if __name__ == '__main__':
24
            test_iter()
25
26
```

27	
28	

神奇partial

partial使用上很像C++中仿函数(函数对象).

在stackoverflow给出了类似与partial的运行方式

1	defpartial(func, *part_args):

2	defwrapper(*extra_args):	
3	args = list(part_args)	
4	args.extend(extra_args)	

```
5
6   return func(*args)
7   return wrapper
```

利用用闭包的特性绑定预先绑定一些函数参数，返回一个可调用的变量，直到真正的调用执行

```
1   #!/usr/bin/env python
```

```
2
        # -*- coding: utf-8 -*-
3
        from functools import partial
4
        def sum(a, b):
```

```
5
                return a
                + b
6
7       deftest_p
        artial():
8       fun =
        partial(su
        m,
```

```
        2)

9       # 事先绑定一个参数, fun成为一个只需要一个参数的可调用变量

10

        print fun(

        3)

11      # 实现执行的即是 sum(2, 3)
```

```
12  if __name__ == '__main__':
13      test_partial()
14      # 执行结果
15
```

16	5
17	

神秘eval

eval我理解为一种内嵌的python解释器(这种解释可能会有偏差)，会解释字符串为对应的代码并执行，并且将执行结果返回

看一下下面这个例子

1	#!/usr/bin/env python

```
2
        # -*-
coding:
utf-8 -*-
3
deftest_fir
st():
4
return
        3
5
```

```
6 deftest_second(num):
7 return num
8 action = {
        # 可以看做是一个 sandbox
```

```
9
                "para":
                    5,
10
11              "test_first" : test_first,
12              "test_second":
```

```
test_second

13

}

14

deftest_eavl():

15

condition =
```

```
16  "para == 5 and test_second(test_first) > 5"
17
    res = eval(condition, action)
18      # 解释condition并根据action对应的动作执行
19      print res
```

```
20      if __name__ == '_
21
```

exec

- exec在Python中会忽略返回值, 总是返回None, eval会返回执行代码或语句的返回值
- exec和eval在执行代码时，除了返回值其他行为都相同
- 在传入字符串时, 会使用compile(source, '', mode) 编译字节码. mode的取值为exec和eval

```
1
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
deftest_first():
    print
```

```
    "hello"

def test_second():

def test_first():
    print
```

```
    "second"

9

def test_third():

10

        print

    "third"

11
```

```
action = {
12
"test_second": test_second,
13
"test_third": test_third
14
15
}
```

```
16
    deftest_exec():
17
        exec "test_second" in
18  action
```

```
19  if __name__ == '__main__':
20      test_exec()
21      # 无法看到执行结果
22
```

23	

getattr

getattr(object, name[, default])Return the value of the named attribute of object. name must be a string. If the string is the name of one of the object’s attributes, the result is the value of that attribute. For example, getattr(x, ‘foobar’) is equivalent to x.foobar. If the named attribute does not exist, default is returned if provided, otherwise AttributeError is raised.
通过string类型的name，返回对象的name属性(方法)对应的值，如果属性不存在，则返回默认值，相当于object.name

1	# 使用范例

```
2
classTestGetAttr(object):
3
    test = "test attribute"
4
    defsay(self):
5
```

```
        print
6       "test method"
7
deftest_getattr():
8
my_test = TestGetAttr()
9
```

```
                try:
10
                print getattr(my_test, "test")
11
            except AttributeError:
12
```

```
        print "AttributeError!"
13
14
        try:
15
            getattr(my_test, "say")()
16
```

except AttributeError:

17

没有该属性,且没有指定返回值的情况下

18

print

"Method Error!"

19

```
20  if __name__ == '__main__':
21      test_getattr()
22      # 输出结果
23
```

24	test attribute
25	test method

命令行处理

1	defprocess_command_line(argv):

2	"""	
3	Return a 2-tuple: (settings object, args list).	
4	`argv` is a list of arguments, or `None` for ``sys.argv[1:]``.	
5		

6	"""	
7	if argv is None:	
8		

```
9       argv = sys.argv[1:]
10      # initialize the parser object:
11      parser = optparse.OptionParser(
12
```

```
formatter=optparse.TitledHelpFormatter(width=
        78),

add_help_option=
None)

        #
define
```

```
    options here:
16
17  parser.add_option(
        # customized description; put --help last
18
        '-h',
        '--help',
19      action=
```

```
                'help',
20
                help=
                'Show
                this help
                message
                and exit.')
21
22              settings,
                args =
                parser.pa
                rse_args(a
                rgv)
```

```
23
        # check number of arguments, verify values, etc.:
24
        if args:
25
            parser.error(
26
                'program takes no command-line
```

```
                argument
                s; '
27
                '"%s"
                ignored.'
                % (args,))
28
                        #
                further
                process
                settings
                & args if
                necessary
29
```

```
    return settings, args

def main(argv=None):

    settings, args = process_command_line(argv)
```

```
        # application code here, like:
34
        # run(settings, args)
35
        return
36      0
        # success
```

```
37
        if __name__ == '__main__':
38
            status = main()
            sys.exit(status)
```

读写csv文件

```
1  # 从csv中读取文件, 基本和传统文件读取类似
2
3  import csv
4
5  with open(
```

```
        'data.csv',
4
            'rb')
            as f:
5
    reader =
    csv.reade
    r(f)
6
        for
    row
7       in
    reader:
```

```
8         print row
9         # 向csv文件写入
10        import csv
```

```
11
        with open('data.csv', 'wb') as f:
12
13
    writer = csv.writer(f)
14
```

writer.writerow([

15 'name',

'address',

'age'])

单行写入

data = [

(

'xiaoming
',

'china',

'10'),

(

'Lily',

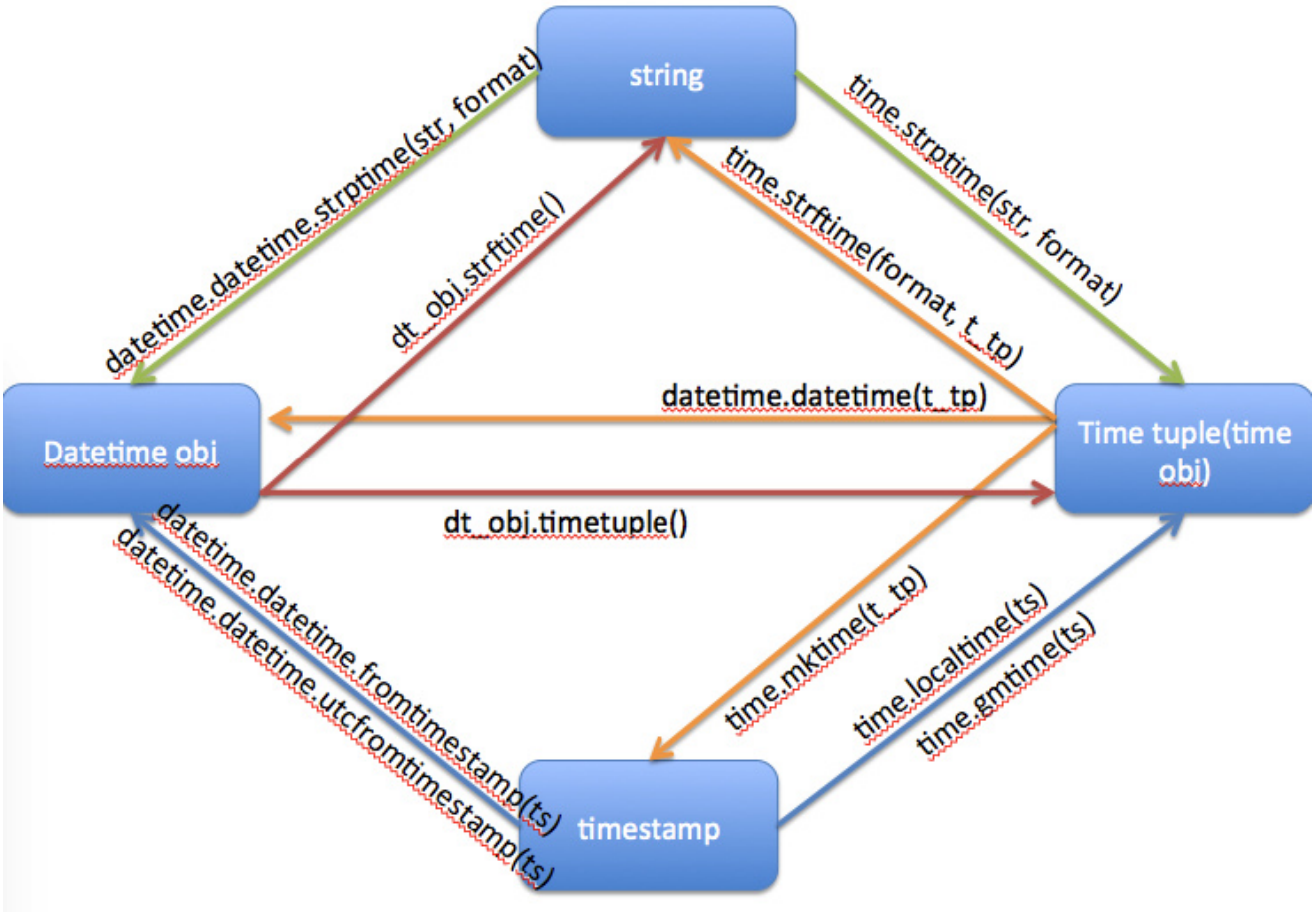
'USA',

'12')]

	writer.writerows(data) # 多行写入

各种时间形式转换

只发一张网上的图，然后差文档就好了，这个是记不住的



字符串格式化

一个非常好用，很多人又不知道的功能

1	>>> name = "andrew"

2	>>>"my name is {name}".format(name=name)
3	'my name is andrew'

参考链接

- What is the difference between @staticmethod and @classmethod in Python?
- Python @property versus getters and setters
- How does the @property decorator work?
- How does the functools partial work in Python?
- What’ s the difference between eval, exec, and compile in Python?
- Be careful with exec and eval in Python
- Python (and Python C API): new versus init
- Python ‘self’ keyword`self`不是关键字, 是一个约定的变量名
- Python进阶必读汇总
- 使python类可以判断真值
- Best Python Resources
- Python安全编码指南