其他常用标准库

笔记本：	极客时间Python		
创建时间：	2018/5/26 22:40	更新时间：	2018/6/14 17:37
作者：	尹会生	位置：	22° 21'4 N 113° 35'50 E
URL：	about:blank		

Python标准库的官方文档在如下位置
https://docs.python.org/3/library/index.html

日常应用比较广泛的模块是：

1. 文字处理的 re
2. 日期类型的time、datetime
3. 数字和数学类型的math、random
4. 文件和目录访问的pathlib、os.path
5. 数据压缩和归档的tarfile
6. 通用操作系统的os、logging、argparse
7. 多线程的 threading、queue
8. Internet数据处理的 base64 、json、urllib
9. 结构化标记处理工具的 html、xml
10. 开发工具的unitest
11. 调试工具的 timeit
12. 软件包发布的venv
13. 运行服务的__main__

5.数据压缩和归档的tarfile
5.1 在windows系统中压缩包是zip rar格式， 但是在linux 系统中经常用到的是.tar.gz。 为什么是双扩展名，因为gz是gzip压缩工具，只能对文件压缩， 因此需要tar格式对目录进行打包。

我们需要一个os模块新的功能，叫做walk（），用于遍历目录下所有的文件和目录：

```
import os
for root, dir, files in os.walk("."):
    print(root, dir, files)
```

5.2 将目录/tmp/tarball创建一个压缩包叫做/tmp/当前日期.tar.gz

```
import tarfile
import os
import time
backupdate = time.strftime('% Y % m % d’ )
backupstring = '/tmp/' + backupdate + '.tar.gz'
# 创建压缩包名
```

```
tar = tarfile.open(backupstring, "w:gz")
# 创建压缩包
for root, dir, files in os.walk("/tmp/tarball"):
    for file in files:
        fullpath = os.path.join(root, file)
        tar.add(fullpath)
    tar.close()
```

5.3 对.tar.gz 格式进行解压缩

```
import tarfile

def extract(tar_path, target_path):
    try:
        tar = tarfile.open(tar_path, "r:gz")
        filenames = tar.getnames()
        for filename in filenames:
            tar.extract(filename, target_path)
        tar.close()
    except Exception as e:
        print('extract error %s' %e)

extract('/tmp/tartest.tar.gz' , '/tmp/x')
```

6.通用操作系统的os、logging、argparse

6.1 os 模块里包含了非常多的方法，主要是和文件相关的一系列工具， 如我们使用过的os.path , os里还包括了如 os.read() os.close() os.access() 等一系列方法， 他们分别对应 我们之前讲解的文件操作的底层方法，

因为里面大部分操作文件的知识点我们都介绍过，我们就不在这里重复为大家讲解了， 建议大家通过官方文档进行了解。

6.2 logging模块是Python用来输出日志的模块，我们初学python时使用print( ) 来输出需要调试的信息，但是生产环境程序要将关键信息记录到特定文件中，这些信息用于定位问题，查找bug或审计等需求。这类信息文件叫做日志。我们通过代码为大家演示一下python如何产生日志：

```
import logging
logging.basicConfig(
filename='log1.log',
format='%(asctime)s -%(name)s-%(levelname)s-%(module)s:%(message)s',
datefmt='%Y-%m-%d %H:%M:%S %p',
level=logging.DEBUG)
```

```
while True:
    option = input("input a digit:")
    if option.isdigit():
        print("right", option)
        logging.info('option correct')
    else:
        logging.error("Must input a digit!")
```

为了便于区分日志的重要程度，将日志做了级别的划分， Python日志有5个级别

```
logging.debug('有bug')
logging.info('有新的信息')
logging.warning('警告信息')
logging.error('错误信息')
logging.critical('紧急错误信息')
```

这些级别可以用英文关键字表示，也可以使用数字的方式表示级别：

Level	Numeric value
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

```
logging.log(10,'log')
```

总结一下使用日志的好处：

1 相对用随意的print() 和 记录临时文件，日志的输出信息能够使用相同格式进行持久化保存
2 分级别记录错误信息，方便程序调试

6.3 argparse
argparse 是命令行参数处理工具，比如一个Linux下的命令 ls，执行的时候可以带不同的参数， 也可以使用 --help参数获取ls命令的使用帮助
ls afile
ls -l
ls --help

对命令行参数的处理，Python可以通过argparse 模块实现：

prog.py

```
# prog.py
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

执行时是这样的：

$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h]

optional arguments: -h, --help show this help message and exit
$ python3 prog.py -v
usage: prog.py [-h]prog.py: error: unrecognized arguments: --verbose
$ python3 prog.py foo
usage: prog.py [-h]prog.py: error: unrecognized arguments: foo

第一个没有任何输出和出错
第二个测试为打印帮助信息，argparse会自动生成帮助文档
第三个测试为未定义的-v参数，会出错
第四个测试为未定义的参数foo，出错

为程序定义一个必选的参数echo

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

$**python3 prog.py**
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo

$python3 prog.py -h
usage: prog.py [-h] echo

positional arguments:
 echo

optional arguments:
 -h, --help show this help message and exit

$python3 prog.py abc
abc

还可以为echo参数增加帮助

```
parser.add_argument("echo", help="echo the string you use here")
```

$ python3 prog.py -h
usage: prog.py [-h] echo

positional arguments:
echo echo the string you use here

接下来再做一个练习，比如输入一个参数，求平方

```
import argparse
parser = argparse.ArgumentParser()
# parser.add_argument("square", help="display a square of a given number")
# 为避免类型错误，增加接收类型
parser.add_argument("square", help="display a square of a given number",
                    type=int)

args = parser.parse_args()
print(args.square**2)
```

$**python3** prog.py 4
Traceback (most recent call last):
File "prog.py", line 5, in <module>
print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'

$**python3** prog.py 4
16

如果实际应用发现命令行参数不是必须输入的怎么写呢？ 接下来添加一个可选参数

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

使用了--verbosity 或者 -v 形式，参数就变为了可选的参数，即 ： 不输入参数也可以

练习一下，增加“-v”再试下

```
parser.add_argument("-v", "--verbosity", help="increase output verbosity")
args = parser.parse_args()
```

注意：不输入参数，输入 python prog.py -v 1 和 python prog.py --verbosity 1 都没有问题，但是 -v 后面不加参数会报错

$ python prog.py -v
usage: prog.py [-h] [-v VERBOSITY]
prog.py: error: argument -v/--verbosity: expected one argument

如何像 -h 一样直接使用，不用带后面的参数呢，增加 action='store_true'

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

如果你觉得这些功能还不满足你日常使用的话argparse还能支持可选参数，如：
choices=[0, 1, 2] 让参数只能在 0 1 2 中选择一个
default=0 参数默认值是0

等等 这些就是argparse 模块经常用到的功能了，如果希望继续丰富自己的参数，可以参考官方文档

7.多线程的 threading、queue
8.Internet数据处理的 base64 、json、urllib
9.结构化标记处理工具的 html、xml

7 多线程我们已经介绍完了， 8 9 会在后面的爬虫为大家详细介绍

10.开发工具的unitest

什么是unitest单元测试？ 单元测试是测试的一种类型，表示对最小可测试的单位进行测试。 要理解单元测试需要从软件设计说起。软件设计要解决的目标有两个，一个是实现需求，另一个就是可重复的稳定运行。
当代码量从hello world 到 上万行之后，如果需要重构代码时，就不能光靠智商-聪明才智 来维护代码，保证不出bug。这时候测试的价值就体现出来了，所以在新编写代码时单元测试体现不出来威力。

我们来看个简单的例子：
Person.py

```
class Person:
    name = []

    def set_name(self, user_name):
        self.name.append(user_name)
        return len(self.name) - 1

    def get_name(self, user_id):
        if user_id >= len(self.name):
            return 'There is no such user'
        else:
            return self.name[user_id]


if __name__ == '__main__':
    person = Person()
    print('User Abbas has been added with id ', person.set_name('Abbas'))
    print('User associated with id 0 is ', person.get_name(0))
```

__name__ 是个特殊的变量，Python会为他自动赋值为当前的模块名称， 如果被直接运行的话，它的名字就是 __main__ ；
这句话的意思就是：当模块直接被运行时，下面的代码块将被运行，如果这个文件被当做模块导入，就不会运行。

$python3 Person.py

User Abbas has been added with id 0
User associated with id 0 is Abbas

Process finished with exit code 0

单元测试是什么结构?

```
import unittest

class Testing(unittest.TestCase):
    def test_string(self):
        a = 'some'
        b = 'some'
        self.assertEqual(a, b)

    def test_boolean(self):
```

```
        a = True
        b = True
        self.assertEqual(a, b)

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

Basic_Test.testing

```
import unittest

class Testing(unittest.TestCase):
    def test_string(self):
        a = 'some'
        b = 'some'
        self.assertEqual(a, b)

    def test_boolean(self):
        a = True
        b = True
        self.assertEqual(a, b)

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

使用命令行执行
$ python3 -m unittest **-v** unitest_v1.py
test_boolean (unitest_v1.Testing) ... ok
test_string (unitest_v1.Testing) ... ok

Ran 2 tests in 0.001s

OK

注意： 1 要继承 unittest.TestCase ， 表示一个单独的测试单元 --- 测试用例
2 测试方法要以 test_ 开头
3 assertEqual 是单元测试引入的方法，还有哪些方法呢?
单元测试的方法：

METHOD	**CHECKS THAT**
assertEqual(a,b)	a==b
assertNotEqual(a,b)	a != b

assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a,b)	a is b
assertIs(a,b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

单元测试的目的就是为了保证 set_name 和 get_name 两个方法返回正常的结果，对Person.py的单元测试用如下代码：

PersonTest.py

```
import unittest
import Person as PersonClass


class Test(unittest.TestCase):
    """
    The basic class that inherits unittest.TestCase
    """
    person = PersonClass.Person() # instantiate the Person Class
    user_id = [] # variable that stores obtained user_id
    user_name = [] # variable that stores person name

    # test case function to check the Person.set_name function
    def test_0_set_name(self):
        print("Start set_name test\n")
        """
```

```
        Any method which starts with ``test_`` will considered as a test case.
        """
        for i in range(4):
        # initialize a name
        name = 'name' + str(i)
        # store the name into the list variable
        self.user_name.append(name)
        # get the user id obtained from the function
        user_id = self.person.set_name(name)
        # check if the obtained user id is null or not
        self.assertIsNotNone(user_id) # null user id will fail the test
        # store the user id to the list
        self.user_id.append(user_id)
        print("user_id length = ", len(self.user_id))
        print(self.user_id)
        print("user_name length = ", len(self.user_name))
        print(self.user_name)
        print("\nFinish set_name test\n")

    # test case function to check the Person.get_name function
    def test_1_get_name(self):
        print("\nStart get_name test\n")
        """
        Any method that starts with ``test_`` will be considered as a test case.
        """
        length = len(self.user_id) # total number of stored user information
        print("user_id length = ", length)
        print("user_name length = ", len(self.user_name))
        for i in range(6):
            # if i not exceed total length then verify the returned name
        if i < length:
            # if the two name not matches it will fail the test case
            self.assertEqual(self.user_name[i], self.person.get_name(self.user_id[i]))
        else:
            print("Testing for get_name no user test")
            # if length exceeds then check the 'no such user' type message
            self.assertEqual('There is no such user', self.person.get_name(i))
            print("\nFinish get_name test\n")


if __name__ == '__main__':
    # begin the unittest.main()
    unittest.main()
```

11.调试工具的 timeit

11.1 timeit 测量执行时间

通常在一段程序的前后都用上time.time(),进行相减就可以得到一段程序的运行时间，不过python提供了更强大的计时库：timeit

#导入timeit.timeit
from timeit import timeit
timeit('x=1')

测试一个函数运行1000次的的执行时间：

```
from timeit import timeit

def func():
s = 0
for i in range(1000):
    s += i
    print(s)

# timeit(函数名_字符串，运行环境_字符串，number=运行次数)
t = timeit('func()', 'from __main__ import func', number=1000)
print(t)
```

11.2 repeat 多次调用timeit

```
from timeit import repeat

def func():
    s = 0
    for i in range(1000):
        s += i

#repeat和timeit用法相似，多了一个repeat参数，表示重复测试的次数(可以不写，默认值为3.)，返回值为一个时间的列表。
t = repeat('func()', 'from __main__ import func', number=100, repeat=5)
print(t)
print(min(t))
```

12.软件包发布的venv

12.1 venv是在开发环境中使用的包，它解决了版本冲突问题

随着项目越来越多， 你使用不同版本的Python的可能性就越大，至少会支持不同Python版本的库，我们不得不面对一种很常见的情况是库不向后兼容。怎么解决依赖冲突呢？venv正是为此而生，它允许你安装多个Python版本， 每个版本对应自己的Python（或库）。它其实并没有安装一个新的Python副本，而是通过很奇妙的方法来保持环境独立

使用方法：

1.创建虚拟环境 ,test 为虚拟环境所在的文件夹的名称

```
$ python3 -m venv test
```

2. 激活虚拟环境 source activate（要进入安装目录下的bin目录）

3. 退出虚拟环境 deactivate

13.__name__ 变量

```
if __name__ == '__main__':
    # 如果是以模块方式运行的话,这部分代码块不运行，单独执行的话此部分会运行
```