Python 学习笔记（下）

这份笔记是我在系统地学习python时记录的，它不能算是一份完整的参考，但里面大都是我觉得比较重要的地方。

函数设计与使用

形参与实参

1. 函数调用时向其传送实参，根据不同的实参类型，将实参的值或引用传递给形参。
2. 绝大多数情况下再函数内部修改形参的值不影响实参。e.g.

```
def addone(a):
 print(a)
 a +=1
 print(a)
a = 1
addone(a)
print(a)
>> 1  >> 2 >>1
```

3. 传递给函数的是可变序列并再函数内使用下标等方式增删元素或修改元素值，则修改后的结果可反映到函数之外的实参。(字符串、元组属不可变序列，列表、字典可变)

参数类型

函数参数类型有：普通参数、默认值参数、关键参数、可变长度参数等

默认值参数

- 定义形参时候设置的默认值，可使用 `funcname.__defaults__` 查看函数所有的默认参数的值。

```
def funcname(..., 形参名=默认值):
    函数内容

funcname.__defaults__
```

- 定义带有默认值参数的函数时，默认值参数必须出现在函数形参列表的最右端，即默认值参数的右边不能再出现非默认值参数。（可以有可变长度参数）e.g. `def f (a, b, c=3):`
- 多次调用函数并且不为默认参数传递值时，默认参数仅在第一次调用时进行解释。对于字典、列表这样的默认参数，可能会产生严重的逻辑错误。e.g

```
def demo(newitem, old_list=[]):
    old_list.append(newitem)
    return old_list
print(demo('5', [1,2,3,4]))
print(demo('6', ['a']))
print(demo('5')
print(demo('6')

# 对比
def demo(newitem, old_list=None):
    ...
```

关键参数

调用函数时的参数传递方式，可以按参数名传递值，实参顺序可以与形参顺序不一致。

可变长度参数

两种形式： `*parameter` ， `**parameter` 。 前者接收任意多个实参并将其放在一个**元组**中，后者接收类似于关于关键参数一样的显式赋值形式的多个实参并将其放入**字典**中。

```
def demo(*p):
    print(type(p), p)


>> demo(1,2,3,'string')
<class 'tuple'> (1, 3, 'string')

def demo2(**p):
    for item in p.items():
        print(item)     # item是元组类
    print(type(p))

>> demo2(x=1, y=2, z='string')
('x', 1)
('y', 2)
('z', 'string')
<class 'dict'>
```

参数传递时的序列解包

为含有多个变量的函数传递参数时，可以使用python的列表、元组、集合、字典以及其他的可迭代对象作为实参。

解包字典时，默认使用的是字典的**键**，要使用**值**则需要 `dict.values()` ，使用**键值对**元组则需要 `dict.items()` 。

```
def demo(a,b,c):
      print(a+b+c)

 >>> seq = [1,2,3]
>>> demo(*seq)
6
dic1 = {1:'a', 2:'b', 3:'c'}
dic2 = {'a':1, 'b':2, 'c':3}
>>> demo(*dic1)
6
>>> demo(*dic2)
abc
```

结束语句

没有 `return` 或不返回任何值的 `return` ，则认为返回 `None` 。

例如 `list.sort()` 原地操作不返回值，而 `sorted()` 内置函数返回排序后的列表。

变量作用域

1. 变量已在函数外定义，在函数内引用可直接用，而修改该变量的值需要在函数内用 `global` 声明这个变量为全局变量。
2. 在函数内部直接使用 `global` 关键字将一个变量声明为全局变量，即使在函数外没有定义该全局变量，在调用这个函数后将自动增加新的全局变量。

`lambda` 表达式

1. 可以用来声明匿名函数，即没有名字的临时的函数。
2. `lambda` 仅包含一个表达式不包含其他复杂语句。
3. 可以在表达式中调用其他函数，并支持默认值参数和关键参数。
4. 表达式的计算结果就是函数的返回值。

```
f = lambda x, y, z: x+ y+ z
print(f(1,2,3))
g = lambda x, y=2, z=3: x+y+z   # 默认值参数
print(g(1))
print(g(2, z=4, y=5))   # 调用时使用关键参数

L = [(lambda x: x**2), (lambda x: x**3), (lambda x: x**4)]  # Lambda表达式列表，还可有字典形式key: (Lambda)
print(L[0](2), L[1](2), L[2](2))

L = [1,2,3,4,5]
print(map((lambda x: x+10), L))     # 无名Lambda

import random
data = list(range(20))
random.shuffle(data)
data.sort(key=lambda x:x)
data.sort(key=lambda x: len(str(x)))
data.sort(key=lambda x:len(str(x)), reverse=True)
```

使用 `lambda` 表达式时，要注意变量作用域的问题。外部定义的变量不是局部变量。

```
for x in range(10):
    r.append(lambda: x**2)      X

for x in range(10):
    r.append(lambda n=x: n**2)
```

高级话题map, reduce等

`map(func, iterable)` : 将单参函数依次作用到一个序列或迭代器对象的每个元素上，返回一个map对象作为结果，其每一个元素为原序列中元素经过该函数处理后的结果，不对序列或迭代器对象做任何修改。

> *map(func, *iterables) —> map object*
> *Make an iterator that computes the function using arguments from*
> *each of the iterables. Stops when the shortest iterable is exhausted.*

```
#例：map将空格分割的数字字符串变成数字列表
s = '123 425 234'
l_int = list(map(int, s.split()))
[123, 425, 234]
```

`reduce(func, iterable)` : 将双参函数以**累积**的方式从左到右依次作用到一个序列或迭代器对象的所有元素上。e.g. `reduce(lambda x,y: x+y, seq)`
注意：python3中引入需要 `from functools import reduce`

`filter(func, seq)` : 将单参函数作用到一个序列上，返回序列中使得函数返回True的元素组成的列表、元组或字符串。

`yield` : 可用包含它的函数创建生成器。迭代器 惰性求值 （ 尚未掌握 ）

```
def f ():       #生成斐波那契数列
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a+b

a=f()
for i in range(10):
    print(a.__next(), end=' ')

for i in f():
    if i >100:
        break
    print(i, end = ' ')
```

`dis()` : 查看函数的字节码指令 `import dis dis.dis(func)`

嵌套定义与可调用对象

```
def linear(a,b);
    def result(x):
        return a *x +b
    return result
class linear:
    def __init__(self, a, b):       # 新建对象时候调用
        self.a, self.b = a, b
    def __call__(self, x):          # 调用已有对象时用
        return self.a *x +self.b

t = linear(0.3, 2)      # 定义一个可调用对象
t(5)                    # 使用
```

装饰器： 以函数为参数 （ 未掌握 ）

面向对象程序设计

成员方法， 数据成员/成员属性

定义与使用

`isinstance()` 测试一个对象是否为某个类的实例。
`pass` 空语句

self

1. `self` 所有实例方法都必须至少有一个self参数(未必命名为self)，且为第一个形参。
2. 在类的**实例方法**中访问**实例属性**时必须以self为前缀。 `self.attr`
3. 在外部通过**对象名调用对象方法**时不需要传递此参数。 `classAins.func(...)`
4. 通过类名调用则需要显示为self参数传值。 `classA.func(self值, ...)`

类成员与实例成员

类成员：在类中所有方法之外定义的数据成员
实例成员：一般在构造函数 `__init__()` 中定义，且定义时以self作为前缀

```
class A:
    price = 1000            # 类成员
    def __init__(self, c):
        self.color = c       # 实例成员
```

类的方法中可以调用类本身的其他方法，也可以动态地为类和对象增加成员。

```
a = A('red')
a.color
a.price = 100          # 修改
a.newattr = 'a'        # 添加实例属性
A.newattr = ' A'      # 添加类属性

def func1(self, s):
    pass
import types
a.func1 = types.MethodType(func1,  a)   # 动态为对象增加成员方法
```

> *函数与方法有区别，方法一般指与特定实例绑定的函数，通过对象调用方法时，对象本身被作为一个参数传递过去。普通函数不具此特点。*

私有成员与公有成员

私有： `__privateattr` 外部无法直接访问，需调用公有成员方法或特殊方式访问。 特殊方式 `对象名._类名__privateattr`
公有：可在类的内外部进行访问。
保护： `_protectedattr` 不能用 `from module import \*` 导入 只有类对象和子类对象可以访问
系统定义的特殊成员： `__specialattr__`

方法

1. 四类：公有、私有、静态、类方法，前二者属于对象。

2. 私有方法以 `__` 开始，只能在属于对象的方法中以self.xxx()调用。（或特殊方法）
   私有、公有方法用类名调用时需要显示为该方法传递一个对象名，用于明确访问哪个对象的数据成员。

3. 静态方法和类方法可以通过类名和对象名调用，不能访问属于对象的成员，只能访问属于类的成员。
   一般将 `cls` 作为类方法的第一个参数名称。(两者分别使用 `@classmethod (https://github.com/classmethod)` , `@staticmethod (https://github.com/staticmethod)` 来修饰)

属性

使用 `@property (https://github.com/property)` 或 `property()` 函数来声明一个属性。在python2.x中属性没有真正意义的实现，亦为提供保护机制。（为对象增加新的数据成员时，将隐藏同名的已有属性）

```
class Test:
    def __init__ (self, value):
        self.__value = value

    @property
        def value(self):
            return self.__value

a = Test(3)
a.value     # 访问属性  而非私有成员__value
a.value = 5     # 动态添加新成员，隐藏了定义的属性    (python2未保护)
```

在python3中属性支持全面的保护机制。（只读，可修改，可删除）

```
class Test:
    def __init__ (self, value):
        self.__value = value

    @property
        def value(self):
            return self.__value
只读模式 不允许修改值，以及删除对象属性。
t = Test(3)
t.value

t.value = 5  # 只读模式 不允许修改值

t.v =5
del t.v     # 动态增删新成员

del t.value # 删除对象属性  失败
```

可读可修改不可删除

```
class Test:
    def __init__ (self, value):
        self.__value = value
    def __get(self):
        return self.__value
    def __set(self, v):
        self.__value = v

    value = property(__get, __set)

    def show(self):
        print(self.__value)
t = Test(3)
t.value
t.value = 5 # 允许修改对象值
t.show()    # 可知属性值被修改了
```

可读可修改可删除

```
在类中加入
def __del(self):
    del self.__value

value = property(__get,__set,__del)
可实现删除以及动态增加同名
t = Test(3)
t.value =5
del t.value
t.value     # 私有的__value被删除
t.value = 1 # 为对象动态增加属性和对应的私有数据成员
```

特殊方法与运算符重载

构造函数 `__init__()` ：为数据成员设置初值或进行其他必要的初始化工作，创建对象时会被自动调用和执行。可通过为构造函数定义默认值参数来实现类似于其他语言中构造函数重载的目的。没有编写则提供默认。

析构函数 `__del__()` ：释放对象占用的资源，删除和收回对象空间时自动调用和执行。没有编写则python提供一个默认的析构函数。

python类特殊方法

方法	功能
`__setitem__()`	按照索引赋值
`__getitem__()`	按照索引获取值
`__len__()`	计算长度
`__call__()`	函数调用
`__contains__()`	测试是否包含某个元素
`__str__()`	转为字符串
`eq` , `ne` , `lt` , `le` , `gt` , `ge`	==, !=, <, <=, >, >=
....	....

继承

父类(基类)，子类(派生类)。派生类可以继承父类的公有成员，但不能继承其私有成员。在派生类中调用基类的方法可以使用 `super()` 或通过 `基类名.方法名()` 的方式实现。

```
class Person(object):        # 以Person作为基类派生Teacher  Person需以Object为基类
    def __init__(self, name='', age=20, sex='male'):
        self.setName(name)
        ....
    def setName(self, name):
        if not isinstance(name, str):
            print('name must be str')
            return
        self.__name = name

    def show(self):
        print(self.__name, self.__age, self.__sex)

 class Teacher(Person):
    def __init__(self, name='', age=20, sex='male', department='CS'):
        super(Teacher, self).__init__(name, age, sex)   # 调用基类的构造方法
        #or use Person.__init__(self, name, age, sex)   # 调用类的成员方法来构造
        self.setDepartment(department)
```

python支持多继承，父类中有相同的方法名，而在子类中使用时没有指定父类名，则解释器将从左向右按顺序搜索。（尚未理解）

派生类没有定义构造方法时，创建派生类对象会调用基类的构造方法。基类构造方法无法调用派生类中重定义的私有方法可以调用在派生类中重写的公有方法。

文件操作

文件主要有文本文件和二进制文件之分。

文件对象

`open()` 函数可以以指定模式打开指定文件并创建文件对象。
`file1 = open(文件名[, 打开方式[, 缓冲区]])`

- 打开模式

```
    Character Meaning
    --------- ---------------------------------------------------------
    'r'       open for reading (default)
    'w'       open for writing, truncating the file first
    'x'       create a new file and open it for writing
    'a'       open for writing, appending to the end of the file if it ex
ists
    'b'       binary mode （可与其它模式组合使用）
    't'       text mode (default)
    '+'       open a disk file for updating (reading and writing)
    'U'       universal newline mode (deprecated)
```

- 文件对象属性

属性	说明
closed	判断文件是否关闭是则返回True
mode	返回文件的打开模式
name	返回文件的名称

- 文件对象常用方法

属性	说明
flush()	缓冲区写入文件，不关闭文件
close()	缓冲区写入文件，同时关闭文件，释放文件对象
read([size])	从文件中读取size个字符(py3.x)的内容作为结果返回， 缺省一次读取所有的内容
readline()	从文本文件中读取一行内容 作为结果返回 **(含换行符)**
readlines()	把文本文件中的每行文本作为一个字符串存入列表中，返回该列表 **(含换行符)**
seek(offset[,whence])	把文件指针移动到新的位置（指定**字节**的位置）， offset表示相对于whence的位置。whence为0表示从头开始， 为1表示从当前位置开始，2表示从文件尾开始计算
tell()	返回指针的当前位置
turncate([size])	未指定size：删除从当前指针到文件末尾的内容，指定size： 保留文件前size个字符，其余的删除
write(s)	把字符串s的内容写入文件
writelines(s)	把字符串列表写入文本文件，不添加换行符

文本文件操作案例

- 文件的写入操作
  上下文管理器 `with` 可以**自动管理资源**，任何原因跳出with块都会保证文件被正确关闭，并且可以在代码块结束后**自动还原**进入该代码块时的**现场**。(后半尚未理解）
  下方的写入内容的编码为GBK(pycharm终端内写入)

```
s = `文本文件的读取方法\n文本文件的写入方法\n`
 #方法1
f = open('1.txt', 'a+')  # 从文件尾部读写
f.write(s)
f.close()

 #方法2
with open('1.txt', 'a+') as f:
    f.write(s)
```

- 文件的读取(2.txt存放’ SDIBT 中国福建福州’）
  `read()` 读取文件中指定数量的字符而不是字节，中文和英文字母一视同仁。

```
fp = open('2.txt', 'r')
fp.read(5)
UnicodeDecodeError: 'gbk' codec can't decode byte 0xad in position 8: ill
egal multibyte sequence
(把文件改成gbk编码后解决)
'SDIBT'
>>> fp.read(2)
' 中'        #空格中
>>> fp.read(1)
'国'
>>> fp.read(2)
'福建'
>>> fp.read(2)
'福州'
```

- 读取并显示文本文件所有行

```
f=open('1.txt', 'r')
#方法1
while True:
    line = f.readline()
    if line == '':          # 读取的行为''时候表示已经到文件尾部
        break
    print(line, end='')     # readline读取包括换行符
文本文件的读取方法
文本文件的写入方法


f.seek(0)   #回到文件头部
#方法2
li = f.readlines()
for line in li:
    print(line, end= '')
f.close()
```

- 文件指针的移动
  完成读写操作后，都会自动移动文件指针。可以配合 `tell()` 和 `seek()` 使用来确定读写位置。tell和seek对应的是字节数，指针指的是下一个要进行读写的位置，其中字母 占1字节，汉字占2字节。（似乎是cp936编码如此）

```
>>> f.tell()
0
>>> f.readline()
'文本文件的读取方法\n'
>>> f.readline()
'文本文件的写入方法\n'
>>> f.tell()
40
f.seek(4)
f.read(1)   # 3.x报错 因为第四个字节是本字的后半部分位置

f.seek(3)
f,read(1)   # 本
```

- 补充

```
print(line, file=fp, end='')        #print亦可用于写入到文件

for line in fp:                     #用迭代方式读取文件行
    pass
```

二进制文件操作案例

数据库、图像、可执行文件等属二进制文件。
序列化：将数据转成对象的二进制形式。
反序列化：序列恢复为对象

直接读写bytes

```
#str->bytes
b = bytes(strobj, encoding='xxx')
b = b'ascii value'
#bytes->str
s = bytesobj.decode('gbk', 'ignore')

#写入
f = open(filename, 'wb+')
f.write(b'ascii can write directly')
bytes0 = bytes('中文\n', encoding='gbk')
f.write(bytes0)
# 读取
f.seek(0)
bytes1 = f.read()
str1 = bytes1.decode('gbk', ignore)
```

pickle 模块

方法	描述
dump(object, file)	
dumps(object)->string	
load(file)->object	一次load加载一个元素
load(string)->object	

```
import pickle

def pdump():
    global f
    n = 4
    i = 1300
    a = 99.2
    dic = {'1': 'a', '2': 'b'}
    try:
    pickle.dump(n, f)
    pickle.dump(i, f)
    pickle.dump(a, f)
    pickle.dump(dic, f)
    except Exception as e:
        print(e)
    finally:
        f.flush()

def pload():
    global f
    f.seek(0)
    n = pickle.load(f)
    i = 1
    while i < n:
    x = pickle.load(f)
    print(x)
    i = i + 1

if __name__ == '__main__':
    global f
    f = open('pickle.dat', 'wb+')
    pdump()
    pload()
    f.close()
```

struct 模块

> *DESCRIPTION*
> *Functions to convert between Python values and C structs.*
> *Python bytes objects are used to hold the data representing the C struct*
> *and also as format strings (explained below) to describe the layout of data*
> *in the C struct.*

文件级操作

文件内容读写：使用上述文件对象
处理文件路径： `os.path` 模块
命令行读取文件内容： `fileinput` 模块
创建临时文件： `tempfile` 模块
表示和处理文件系统路径： `pathlib`

`os` 与 `os.path` 模块

> *os提供操作系统功能和访问文件系统的简便方法*
> *os.path提供大量用于路径判断、切分、连接以及文件夹遍历的方法*

- os模块常用文件操作方法

方法	说明
access(path, mode)	按照mode指定的权限访问文件
open(path, flags, mode, *,dir_fd)	按照mode指定的权限打开文件，默认权限为读写执行
chmod()	改变文件的访问权限
remove(path)	删除指定的文件
rename(src, dst)	重命名文件或目录，可以实现改名或移动
stat(path)	返回文件的所有属性
fstat(path)	返回打开的文件的所有属性
listdir(path)	返回path目录下的文件和目录列表
startfile(filepath[, operation])	使用关联的应用程序打开指定文件

- os.path模块常用文件操作方法

方法	说明
abspath(path)	返回绝对路径
dirname(p)	返回目录的路径
exists(path)	判断文件是否存在
getatime(filename)	返回文件的最后访问时间
getctime(filename)	返回文件的最后创建时间
getmtime(filename)	返回文件的最后修改时间
getsize(filename)	返回文件的大小
isabs(path)	判断是否为绝对路径
isdir(path)	判断path是否为目录

方法	说明
isfile(path)	判断path是否为文件
join(path, * paths)	连接两个或多个path
split(path)	对路径进行分割，以列表形式返回
splitext(path)	从路径中分割文件的扩展名
splitdrive(path)	从路径中分割驱动器的名称

```
#示例：列出当前目录下扩展名为py的文件
import os
print([fname for fname in os.listdir(os.getcwd()) if os.path.isfile(fname) and fname.endswith('.py')])    #列表推导式

#path
p = os.path.join('D:\\', 'CODE')
>>> p
'D:\\CODE'
>>> p = os.path.join(p, 'Django')
>>> p
'D:\\CODE\\Django'
>>> exits(p)
True

>>> p1 = os.path.join('D:', 'CODE')
>>> p1
'D:CODE'
>>> os.path.exists(p1)
True
```

`shutil` 模块

`import shutil`
`copyfile(path1, path2)` ：复制文件
`make_archive()` , `unpack_archive()` ：压缩与解压缩
`shutil.rmtree()` ：删除文件夹

目录操作

- os模块常用目录操作方法与成员

方法	说明
mkdir(path[, mode=0777])	创建目录

方法	说明
makedirs(path1/path2..., mode=511)	创建多级目录
rmdir(path)	删除目录
rmvedirs(path)	删除多级目录
listdir(path)	返回指定目录下的文件和目录信息
getcwd()	返回当前工作目录
get_exec_path()	返回可执行文件的搜索路径
chdir(path)	把path设为当前的工作目录
walk(top, topdown=True, onerror=None)	遍历目录树，该方法返回一个元组，包含3个元素： 所有路径名，所有目录列表，文件列表
sep	当前操作系统所使用的路径分隔符
extsep	当前操作系统所使用的文件扩展名分隔符

```
#walk 示例
import os
for root, dirs, files in os.walk(".", topdown=False):
    for name in files:
        print(os.path.join(root, name))
    for name in dirs:
        print(os.path.join(root, name))
```

异常处理

异常处理结构

`try` 被监控的语句块 可能会引发异常
`except` 捕捉相应的异常
`else` 没有发生异常的时候执行
`finally` 无论是否发生异常都会执行

异常类与自定义异常

```
class ShortInputException(Exception):
    """自定义异常类"""
    def __init__(self, length, atleast):
        Exception.__init__(self):
        self.length = length
        self.atleast = atleast

try:
    s = input('请输入：')
    if len(s)<3:
        raise ShortInputException(len(s), 3)
except EOFError:
    print('EOF结束')
except ShortInputException as x:
    print('ShortInputException:输入长度{}, 应该至少为{}'.format(x.length, x.atleast)
else:
    print("nothing happened')
```

断言与上下文管理

`with` 是从Python2.5引入的一个新的语法，它是一种上下文管理协议，目的在于从流程图中把try,except 和finally 关键字和资源分配释放相关代码统统去掉，简化try....except....finlally的处理流程。with通过**enter**方法初始化，然后在**exit**中做善后以及处理异常。所以使用with处理的对象必须有**enter**()和**exit**()这两个方法。其中**enter**()方法在语句体（with语句包裹起来的代码块）执行之前进入运行，**exit**()方法在语句体执行完毕退出后运行。with 语句适用于对资源进行访问的场合，确保不管使用过程中是否发生异常都会执行必要的“清理”操作，释放资源，比如文件使用后自动关闭、线程中锁的自动获取和释放等。

BIF补充

BIF: built-in functions, 内置函数

- `dir(__builtins__)` 查看内置函数列表
- `bin()` 将数转换为二进制数
- `int(binstr, 2)` 二进制字符串转换为int
- `pow(x, n)` 计算x的n次幂
- `divmod()` Return the tuple (x//y, x%y). Invariant: div*y + mod == x.
- `zip(a,b)`:
  - a = [x,y,z] b = [c,d,e] d = dict(zip(a,b))
  - list(zip(a,b)) >> [(x,c),(y,d),(z,e)]

文章信息

标题：Python 学习笔记（下）
作者：快刀切草莓君
分类：编程语言
发布时间：2020年2月11日
最近编辑：2020年4月11日
浏览量：100

快刀切草莓君

Weibo (https://weibo.com/3031783235)　Github (https://github.com/Zaaachary)　关于

友情链接

Touko (https://wasteland.touko.moe/)　老齐 (https://itdiffer.com/)　Mr_Wang (http://blog.wh241.cn/)

SlyLi (http://blog.slyli.cn/)

互联网ICP备案：闽ICP备18004703号-1 (http://beian.miit.gov.cn/)