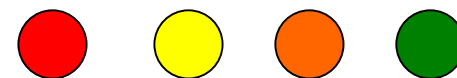


Python与数据科学导论 C02

—— 函数、函数闭包

信息科学技术学院

胡俊峰



本次课提纲

- 列表生成式（续）， 对象的copy机制
- 函数
- 高阶函数、Lambda表达式
- 函数的闭包与装饰器
- 文件操作
- Jupyter notebook的宏命令操作



嵌套的列表生成式

```
1 # 二维列表
2 matrix = [[1, 2, 7], [4, 9], [6, 5, 4, 3]]
3
4 flatten_matrix = [val
5                     for sublist in matrix      # 数据源列表
6                     for val in sublist if val < 6 # 遍历每个数据源
7                     ]
8
9 print(flatten_matrix)
10
```

[1, 2, 4, 5, 4, 3]



List comprehension:

```
[expression for target1 in iterable1 [if condition1]
    for target1 in iterable2 [if condition2]
    for target1 in iterable3 [if condition3]
    for target1 in iterable4 [if condition4]
    .....
    for targetN in iterableN [if conditionN]]
```

```
ls = [ 'a', 'b', 'c' ]
newLs = [e.upper() for e in ls]
```

newLs

['A', 'B', 'C']

```
ls = [ 2, 4, 6 ]
li = [1, 2, 3]
newLs = [e**i for e in ls if e != 2
        for i in li if i != 2
        ]
```

newLs

[4, 64, 6, 216]

zip()函数

- 输入多个可迭代对象，依次取出对应元素（完成计算-合并）
- 直到其中一个迭代器到达尾部

```
a = [10, 20, 30, 40]
b = ['ab', 'cd', 100]
c = (i for i in range(5))
li = [(i, j, k) for i, j, k in zip(a, b, c)]
li
```

```
[(10, 'ab', 0), (20, 'cd', 1), (30, 100, 2)]
```



```
1  # Program to multiply two matrices using list comprehension
2
3
4  X = [[12, 7, 3], [4 , 5, 6], [7 , 8, 9]] # 3x3 matrix
5  Y = [[5, 8, 1, 2], [6, 7, 3, 0], [4, 5, 9, 1]] # 3x4 matrix
6  result = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]] # result is 3x4
7
8  result = [[sum(a*b for a, b in zip(X_row, Y_col))
9             for Y_col in zip(*Y)] # 解包再zip == 转置
10           for X_row in X]
11
12  for r in result:
13      print(r)
```

```
[114, 160, 60, 27]
[74, 97, 73, 14]
[119, 157, 112, 23]
```

拆分-合并字符串列表

str.split(sep=None, maxsplit=-1)

- Return a **list** of the words in the string, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).

```
] : s = 'ab,cde, fgh, ijk'

print(s.split(',')) # 切分开逗号分割的串
print(s.split(',', maxsplit= 2))

['ab', 'cde', 'fgh', 'ijk']
['ab', 'cde', 'fgh,ijk']
```

str.join(iterable)

- Join a list of words into a string.

```
] : delimiter = ':'

mylist = ['Brazil', 'Russia', 'India', 'China']

print(delimiter.join(mylist))
```

Brazil:Russia:India:China

zip() 函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表。

```
1 str = [[i, j] for i, j in zip('abc', 'bcd')]  
2 print (str)
```

```
[['a', 'b'], ['b', 'c'], ['c', 'd']]
```

join() 方法用于将序列中的元素以指定的字符连接生成一个新的字符串。**split**用来分割字符串 **input()**函数用来读入一个字符串

```
1 a, b, c, *_ = map(float, input().split(' ')) #可以读取用空格分开的前三个浮点数  
2 print(a, b, c)
```

```
2.3 4 1.101  
2.3 4.0 1.101
```

```
1 print( ''.join(['0', '1'][i==j] for i, j in zip(input(), input()))) # 布尔量做下标
```

```
hello world!  
hell o w ld  
11110000011
```

布尔量做下标

对象拷贝机制

加赋值引用并不会直接导致对象被复制

- 浅拷贝复制引用关系
- 深拷贝复制引用的关系及所引用的对象



- 对象复制 VS
对象引用的复制

```
1 li1 = [7, 8, 9, 10]
2
3 li2 = li1 # 传引用
4
5 li1[1] = 16
6
7 li2
```

li2 内容被同时改变

[7, 16, 9, 10]

```
1 b = [{'g': 1.5}] * 4 # 复制引用
2 print(b)
3
4 b[0]['g'] = '32'
5 print(b)
```

[{'g': 1.5}, {'g': 1.5}, {'g': 1.5}, {'g': 1.5}]
[{'g': '32'}, {'g': '32'}, {'g': '32'}, {'g': '32'}]

```
1 b = [{'g': 1}] + [{'g': 1}] + [{'g': 1}] + [{'g': 1}] # 对象列表
2 b[0]['g'] = 2
3 b
```

加法 创建对象列表, 每个元素内容是独立的

[{'g': 2}, {'g': 1}, {'g': 1}, {'g': 1}]



浅拷贝 不拷贝子对象 原始数据改变 子对象会改变

```
: import copy  
  
ls = [1, 2, 3, ['a', 'b']]  
  
c=copy.copy(ls) ← 浅拷贝，复制容器内的引用  
  
c
```

```
: [1, 2, 3, ['a', 'b']]
```

```
: ls[3].append('cccc')  
  
ls.append(6)  
  
ls
```

← 直接append列表 vs 改变列表元素的内容

```
: [1, 2, 3, ['a', 'b', 'cccc'], 6]
```

```
: c
```

```
: [1, 2, 3, ['a', 'b', 'cccc']]
```

深拷贝，复制容器中的对象引用，以及引用对象的内容的内容...

深拷贝 包含对象里面的自对象的拷贝，所以原始对象的改变不会造成深拷贝里任何子元素的改变

```
1 import copy
2
3 list = [1, 2, 3, ["a", "b"]]
4
5 d = copy.deepcopy(list)  #深拷贝，所引用的对象都重新生成
6
7 list.append(4)
8 list[3][0] = 'c'
9
10 print(d)
```

[1, 2, 3, ['a', 'b']] ← 深拷贝，所引用的容器对象内容也被生成副本

```
1 ls[3].append('cccc')
2
3 ls.append(6)
4 ls[3][0] = 'c'
5
6 print(ls)
7 print(c)
```

← 原对象内容被修改，不会被传递

← 所引用对象内容被修改

[1, 2, 3, ['c', 'b', 'cccc'], 6]

[1, 2, 3, ['c', 'b', 'cccc']]



名字绑定 及 引用计数

- 名字是对一个对象的称呼，python将赋值语句认为是一个命名操作（或者称为名字绑定）。
- python中的所有对象都有引用计数
- 对象的引用计数在下列情况下会增加：
 - 赋值操作； 在一个容器（列表，序列，字典等等）中包含该对象
- 对象的引用计数在下列情况下会减少：
 - 离开了当前的名字空间（该名字空间中的本地名字都会被销毁）
 - 对象的一个名字被绑定到另外一个对象
 - 对象从包含它的容器中移除
 - **名字**被显式地用del销毁（如：del i）
- 引用计数为0时会启动对象回收机制（递归引用会导致内存泄露）



```
from sys import getrefcount as grc # 引用计数
```

```
num1 = 2678
```

```
num2 = num1 + 1
```

```
print(grc(num1)) # 打印num1的引用计数
```

```
num3 = num1
```

```
print(grc(num1))
```

```
ref_dict = dict(globals()) # 获得全局引用表
```

```
print([ref for ref in ref_dict if ref_dict[ref] is num1]) # 查看全局表中引用num1的变
```

```
print(grc(num1)) # 再打印num3的引用计数
```

```
del num1
```

```
print(grc(num3))
```

通过Py_IncRef(PyObject *o),
Py_DecRef(PyObject *o). 这对操
作函数来动态调整每个对象实例
的reference_count属性值。

3

4

['num3', 'num1']

6

5

Python的函数

- 函数的一般性质：定义、传参、调用
- 函数主导的生成器与迭代器



Python的函数定义： def 函数名(参数列表):
程序块
return

```
1  def fib(max):  
2  
3      n, a, b = 0, 0, 1  
4  
5      while n < max:  
6  
7          print(b, end = ' —> ')  
8  
9          a, b = b, a + b  
10  
11         n = n + 1  
12  
13     return 'done'  
14  
15 fib(5)
```

1 —> 1 —> 2 —> 3 —> 5 —>

'done'



参数传对象引用:

```
def demo(num1, num2, num_list): # 参数 传引用

    print("函数内部")

    # 赋值语句
    num1, num2 = num2, num1    # 分别指向新对象
    print("num2的id= ", id(num2))
    num_list[2] = 10          # 修改外部对象
    num_list = [1, 2, 3, 4]   # 赋值语句导致指向新的对象

    print(num1, num2)
    print(num_list)

a = 99
b = 101
lst = [4, 5, 6]
print("a的id = ", id(a))
demo(a, b, lst)
print('函数外部')
print(a, b)
print(lst)
```

```
a的id = 94518506617216
函数内部
num2的id= 94518506617216
101 99
[1, 2, 3, 4]
函数外部
99 101
[4, 5, 10]
```



返回值也是引用:

```
def demo(num1, num2, num_list): # 参数 传引用

    print("函数内部")

    # 赋值语句
    num1, num2 = num2, num1 # 分别指向新对象

    num_list[2] = 10 # 修改外部对象

    num_list = [1, 2, 3, 4] # 赋值语句导致指向新的对象

    print(num1, num2)
    print(num_list)

    return num_list, num1 # 返回多个值 (都是引用)

a = 99
b = 101
lst = [4, 5, 6]
print("b的id = ", id(b))

lst, a = demo(a, b, lst) # lst指向新的列表 a指向新的对象

print('函数外部')
print(a, b)
print("a的新id = ", id(a))
print(lst)

b的id = 94518506617280
函数内部
101 99
[1, 2, 3, 4]
函数外部
101 101
a的新id = 94518506617280
[1, 2, 3, 4]
```

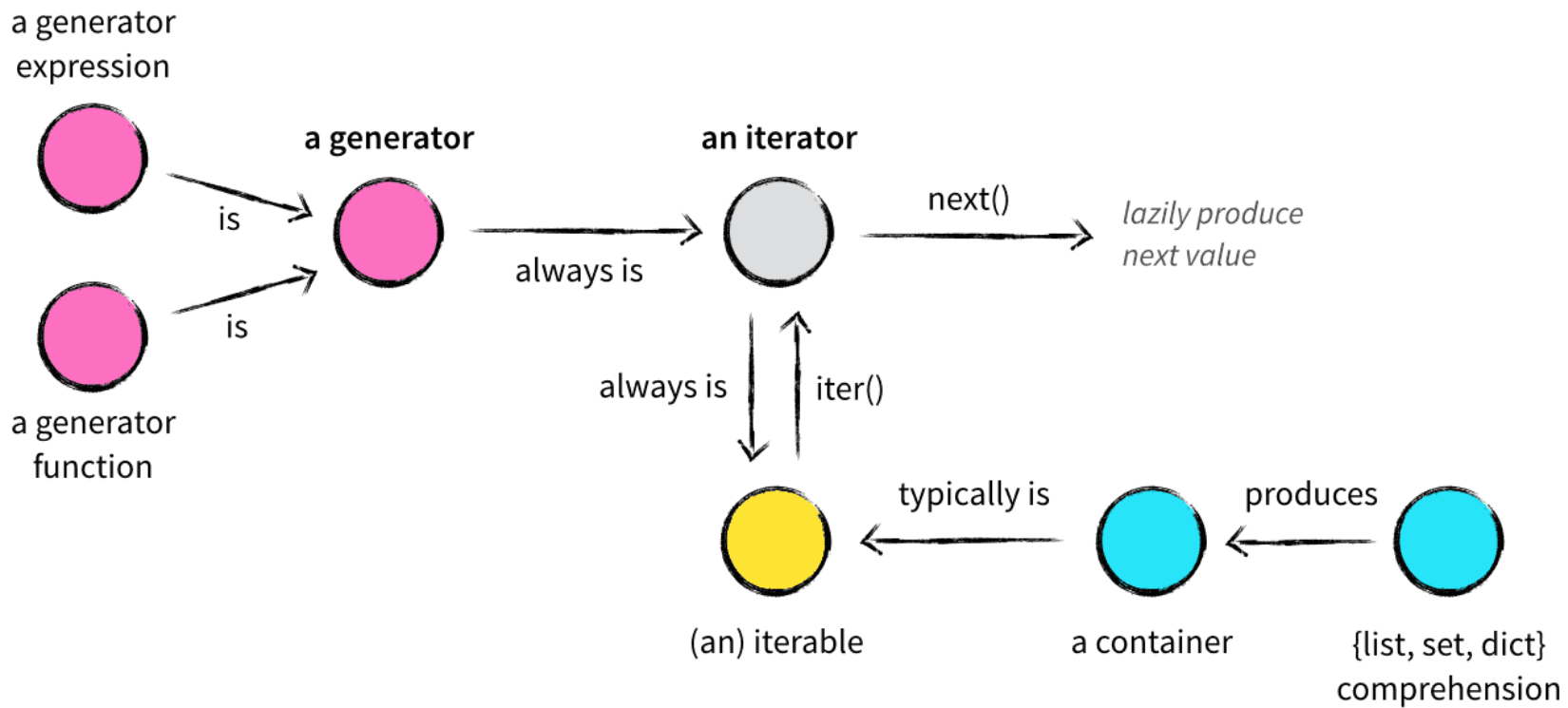


列表实现快速排序算法

```
1  def quicksort(arr):
2      if len(arr) <= 1:      #递归返回条件
3          return arr
4      pivot = arr[len(arr) // 2]
5      left = [x for x in arr if x < pivot]      #生成小于pivot的左列表
6      middle = [x for x in arr if x == pivot]      #middle列表, 支持有重复值
7      right = [x for x in arr if x > pivot]      #生成大于pivot的右列表
8      return quicksort(left) + middle + quicksort(right)      #返回一个新的列表
9
10 print (quicksort([3, 6, 8, 10, 1, 2, 1]))
```



生成器 (generator)



```
: li = [x * x for x in range(4)]  
print(li)  
type(li)
```

[0, 1, 4, 9]

: list

```
: g = (x * x for x in range(4)) # generator expression  
  
print(g)  
type(g)
```

<generator object <genexpr> at 0x000001A23D6D3510>

: generator

```
: for n in g:  
    print(n)
```

0
1
4
9

```
In [9]: next(g)
```

StopIteration

Cell In[9], line 1

----> 1 next(g)

StopIteration:

Traceback (most recent call last)

用函数实现一个 generator —— yield关键字

- 带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象。每次next调用，执行到yield位置函数返回next元素



- 函数实现一个 generator

```
1  def fib(max = 5): ← 设置缺省参数值
2
3      n, a, b = 0, 0, 1
4
5      while n < max:
6
7          yield b    #此时返回下一个序列元素
8
9          a, b = b, a + b
10
11         n = n + 1
12
13 f = fib()
14 f
```

<generator object fib at 0x0000019DEB305138>



```
: 1  for i in f:
   2
   3      print(i, out = ' ', '')
```

- 也可以手动调用全局的 `next()` 方法（因为 `fab(5)` 是一个 generator 对象），这样我们就可以更清楚地看到 `fab` 的执行流程：

```
1  f = fib(5)
2
3  next(f)
```

1

```
1  next(f)
```

1

```
1  next(f)
```

2

- 函数实现一个 generator

```
1  def fib(max = 5):  
2  
3      n, a, b = 0, 0, 1 ← 内部变量，所有实例都有一个副本  
4  
5      while n < max:  
6  
7          yield b    #此时返回下一个序列元素  
8  
9          a, b = b, a + b  
10  
11         n = n + 1  
12  
13 f = fib()  
14 f
```

<generator object fib at 0x0000019DEB305138>



生成器函数： 派生多个生成器实例

```
f1 = fib(5)  # 生成器1  
  
f2 = fib(9)  
  
print(next(f1), next(f1), next(f1), next(f2))
```

1 1 2 1

```
print([i for i in f1])  
print([i for i in f2])  # f1, f2 相互独立
```

[3, 5]

[1, 2, 3, 5, 8, 13, 21, 34]



小结一下：

- 迭代器逻辑上可以看作是一个协议（protocol）
- 迭代器、生成器使用中常作为一个流式数据源



Lambda表达式（匿名函数）

```
1 lambda x, y: x + y          # lambda函数（算子）
```

```
<function __main__.<lambda>(x, y)>
```

```
1 _(2, 3)                     # 调用函数
```

5

```
1 (lambda x, y: x + y)(2, 3) # lambda表达式
```

5



接受函数的函数 —— 高阶函数

- `map()`函数接收两个参数，一个是函数，一个是Iterable的对象，`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。

```
1  def f(x):  
2  
3      return x * x  
4  
5  r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])  
6  
7  list(r)
```

———— [1, 4, 9, 16, 25, 36, 49, 64, 81]

reduce()函数

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

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

13579



filter()函数

- 过滤序列，filter()接收一个函数和一个序列。把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

```
1 def is_odd(n):  
2  
3     return n % 2 == 1  
4  
5 list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
```

[1, 5, 9, 15]

```
1 fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]  
2  
3 odd_numbers = list(filter(lambda x: x % 2, fibonacci))  
4  
5 odd_numbers
```

[1, 1, 3, 5, 13, 21, 55]



递归函数+lambda表达式实现快排

```
def qsort(a):  
    if len(a) <= 1:  
        return a  
    else:  
        return (qsort(list(filter(lambda x: x <= a[0], a[1:]))) # a[0]是哨兵  
                + [a[0]]  
                + qsort(list(filter(lambda x: x > a[0], a[1:]))) # 多加了一层
```

```
print (qsort([3, 6, 8, 10, 1, 2, 1]))
```

```
[1, 1, 2, 3, 6, 8, 10]
```



Sorted()函数:

接受一个可迭代对象, 返回一个排好序的list

```
In [10]: ls = [5, 2, 3, 1, 4]
```

```
new_ls = sorted(ls)  
ls
```

```
Out[10]: [5, 2, 3, 1, 4]
```

```
In [11]: sorted?
```

Signature: `sorted(iterable, /, *, key=None, reverse=False)`

Docstring:

Return a new list containing all items from the iterable in ascending order.

A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

Type: builtin_function_or_method

Sorted()用例:

```
1 ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']  
2 print(sorted(ids))  
3 print(sorted(ids, reverse=True))  
4 print(sorted(ids, key=lambda x: int(x[2:])))  
5 ids
```

['id1', 'id100', 'id2', 'id22', 'id3', 'id30']

['id30', 'id3', 'id22', 'id2', 'id100', 'id1']

['id1', 'id2', 'id3', 'id22', 'id30', 'id100']

- ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']

对象自带的Sort方法：

```
In [13]: ls.sort?  
ls.|
```



Tab键进行代码提示和补全

Signature: `ls.sort(*, key=None, reverse=False)`

Docstring: Stable sort *IN PLACE*.

Type: builtin_function_or_method



Python函数的一些高阶技术

- 函数可变参数列表
- 函数闭包
- 偏函数（部分函数 partial function）
- 函数装饰器



python函数的可变参数列表

****args and **kwargs***

- 主要用于函数定义。支持将不定数量的参数传递给一个函数。
- *args 是用来发送一个非键值对的可变数量的参数列表给当前函数。
**argv是用来接受一个键值对的可变数量的参数列表给当前函数



- 非键值对的可变数量的参数列表

```
1  def test_var_args(f_arg, *argv):    #起始参数（引用），后继序列
2
3      print("first normal arg:", f_arg)
4
5      for arg in argv:
6
7          print("another arg through *argv:", arg)
8
9
10 test_var_args('2018', 'python', 'eggs', 'test')
```

first normal arg: 2018

another arg through *argv: python

another arg through *argv: eggs

another arg through *argv: test



```
1 def test_args(arg1, arg2, *argv, **argd):    #起始参数（引用），后继序列
2
3     print("参数2:", arg2)
4
5     for arg in argv:
6         print("参数列表", arg)
7
8     for k, v in argd.items():
9         print(k, ':', v)
10
11 test_args(2021, 'python', 'data science', 'deep learning',
12           classroom1 = '理教107', time1 = 'Mon 5-6', classroom2 = '理教108', time2 = 'Thur 5-6' )
```

参数2: python

参数列表 data science

参数列表 deep learning

classroom1 : 理教107

time1 : Mon 5-6

classroom2 : 理教108

time2 : Thur 5-6

40 • 混合参数列表

```
1 def cheeseshop(kind, *arguments, **keywords):
2
3     print("-- Do you have any", kind, "?")
4     print("-- I'm sorry, we're all out of", kind)
5
6     for arg in arguments:
7         print("-- Do you have any", arg, "?")
8         print("-- I'm sorry, we're all out of", arg)
9
10    print("--" * 40)          # 打印分割线
11
12    for kw in keywords:
13        print(kw, ":", keywords[kw])
14
15    cheeseshop("tomato", "cabbage", "cucumber",
16               shopkeeper = "Boss",
17               client="Johnson")
```

```
-- Do you have any tomato ?
-- I'm sorry, we're all out of tomato
-- Do you have any cabbage ?
-- I'm sorry, we're all out of cabbage
-- Do you have any cucumber ?
-- I'm sorry, we're all out of cucumber
```

```
shopkeeper : Boss
client : Johnson
```



```
In [70]: 1 def myfun(aa, bb):  
2         print(aa + bb)  
3  
4 dic ={'a':1, 'b': 2}
```

```
In [71]: 1 myfun(*dic)
```

ab

```
In [73]: 1 dic ={'aa':1, 'bb': 2} ←  
2 myfun(**dic)
```

3

```
In [52]: 1 def fun1(a, *b, c, **d):  
2         print(a)  
3         print(b)  
4         print(c)  
5         print(d)  
6  
7 fun1(1, 2, 3, x= 4, c = 5, n = 6)
```

1
(2, 3)
5
{'x': 4, 'n': 6}



嵌套函数的定义：

```
: def fo(par=0):  
    i = par + 1 ← 内部参数  
    def fi(x = 0):      # 外部不可见  
        return x + i  
    return fi()  
print(fo(10))
```





```
1 def print_msg(msg):    # This is the outer enclosing function
2
3     hi = 'Hi,'
4
5     def printer():    # This is the nested function
6         print(hi + msg)    # 引用了外部函数的参数
7
8     return printer    # 生成并返回一个函数实例
9
10 # We execute the function
11 f1 = print_msg("morning!")
12 f2 = print_msg("nice day!")
```

```
1 f1()
2 f2()
3 print(id(f1), id(f2))
```

Hi,morning!

Hi,nice day!

2275266316608 2275266317904



变量的作用域： local-nonlocal-global

nonlocal & global

- python引用变量的顺序为： 当前作用域局部变量->外层作用域变量->当前模块中的全局变量->python内置变量
- global关键字可以用在任何地方，包括最上层函数中和嵌套函数中，**即使之前未定义该变量**，global修饰后也可以直接使用
- nonlocal关键字只能用于嵌套函数中，并且外层函数中定义了相应的局部变量



```

1  def scope_test():
2      def do_local():
3          spam = "local spam No.1"  # 1号 ← 局部变量
4      def do_nonlocal():
5          nonlocal spam
6          spam = "nonlocal spam No.2 " # 2号
7      def do_global():
8          global spam
9          spam = "global spam No.3" #3号
10         spam = "test spam"          # 2号设置
11         do_local()
12         print("After local assignment:", spam)  # 打印 2号 spam
13         do_nonlocal()
14         print("After nonlocal assignment:", spam)
15         do_global()
16         print("After global assignment:", spam)
17     scope_test()
18     print("In global scope:", spam) # 注意缩进 ← 引用全局变量

```

After local assignment: test spam
 After nonlocal assignment: nonlocal spam No.2
 After global assignment: nonlocal spam No.2
 In global scope: global spam No.3



函数闭包： 内部定义了一个函数，然后把该函数作为返回值

```
1 import pickle
2 def print_msg(msg):    # This is the outer enclosing function
3
4     hi = 'Hi,'
5
6     def printer(x):    # This is the nested function
7         ➡ nonlocal hi    # UnboundLocalError: local variable 'hi' referenced before assignment
8         hi += msg
9         print(hi + x)    # 引用了外部函数的参数
10
11     return printer    # 生成并返回一个函数实例
12
13 # We execute the function
14 f1 = print_msg('morning! ')
15 f2 = print_msg("nice day! ")
```

```
1 f1('sir')
2 f2('madam')
```

Hi,morning! sir
Hi,nice day! madam



在 `__closure__` 属性中保存环境

In [5]:

```
1 f1('sir')  
2 f2('madam')
```

```
Hi,morning! morning! sir  
Hi,nice day! nice day! madam
```

In [6]:

```
1 print(f1.__closure__[0].cell_contents)  
2 print(f1.__closure__[1].cell_contents)  
3 print(f2.__closure__[0].cell_contents)
```

```
Hi,morning! morning!  
morning!  
Hi,nice day! nice day!
```



函数闭包还可以实现类似
函数模板的功能

```
: def inc(x):  
    return x + 1  
  
def dec(x):  
    return x - 1  
  
def make_operate_of(func): # 要求输入一个函数  
    def operate(x): # 要求输入一个参数  
        return func(x)  
    return operate  
  
addone = make_operate_of(inc) # 输出一个指定的计算模式  
minusone = make_operate_of(dec)  
  
print(addone(2))  
print(minusone(2))
```


函数装饰器：一般用于抽取共性操作作为功能切片，对一类函数进行包装

```
def decorator(func):  
    def dechouse():  
        print("木地板，", "吊顶", end = "") # end替换换行符  
        func() ←  
    return dechouse  
  
def house():  
    print("房子")  
  
def classroom():  
    print("教室")  
  
ordinary() # 原本的样子  
  
newhouse = decorator(house)  
newhouse()  
newclassroom = decorator(classroom)  
newclassroom()
```

房子
木地板， 吊顶房子
木地板， 吊顶教室



@ 号加装饰器函数名放在函数定义的前面，
表明定义了一个装饰后的函数实例

```
@decorator
def officeroom():
    print("办公室")

officeroom()
```

木地板， 吊顶办公室

```
def smart_divide(func): #对除法操作进行安全检查
    def inner(a,b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("除数不能为0")
            return

        return func(a,b)
    return inner

@smart_divide # 更加安全的除法
def divide(a,b):
    return a/b

divide(4,3)
divide(4,0)
```

I am going to divide 4 and 3

I am going to divide 4 and 0

除数不能为0

通过函数闭包对装饰器进行定制

```
def specify(req):  
    def decorator(func):  #装饰器函数  
        def dechouse():  
            print( req + "木地板, ", "吊顶", end = "") # end替换换行符  
            func()  
        return dechouse  
    return decorator  # 函数闭包, 返回定制的装饰器函数  
  
@specify(' 高档')  # 通过参数实现定制化的装饰器  
def house():  
    print("房子")  
  
@specify(' 普通')  # 通过参数实现定制化的装饰器  
def officeroom():  
    print("办公室")  
  
house()  
officeroom()
```

高档木地板, 吊顶房子
普通木地板, 吊顶办公室

```
def star(func):
    def inner(*args):
        print("*" * 30)      # 重复30次
        func(*args)
        print("*" * 30)
    return inner
```

```
def percent(func):
    def inner(*args):
        print("%" * 30)
        func(*args)
        print("%" * 30)
    return inner
```

```
@star
@percent    # 较下层对应的是更内层
def printer(msg):
    print(msg)
printer("今天沙尘太大啦！！")
```

```
*****
%%%%%%%%%%
今天沙尘太大啦！！
%%%%%%%%%%
*****
```



```

# https://www.geeksforgeeks.org/memoization-using-decorators-in-python/
def memoize_factorial(f): # 输入被装饰的操作函数f
    memory = {}          # 所有被装饰的函数实例共享一个公共缓存（记忆化存储）

    def inner(num):
        if num not in memory: # 查询缓存
            memory[num] = f(num) # 递归调用操作函数f，返回结果进行缓存写入
            print('%d not in ' % (num), end='')
        return memory[num] # 如果缓存有内容，则直接返回结果，对递归函数进行优化

    return inner

@memoize_factorial
def facto(num):
    if num == 1:
        return 1
    else:
        return num * facto(num-1)

print(facto(4))
print(facto(5))
print(facto(4))

```

装饰器实现记忆化存储

```

1 not in 2 not in 3 not in 4 not in 24
5 not in 120
24

```

Python的文件操作

- lpyhton文件操作
- 文本文件读写
- 字节文件操作



文件读写

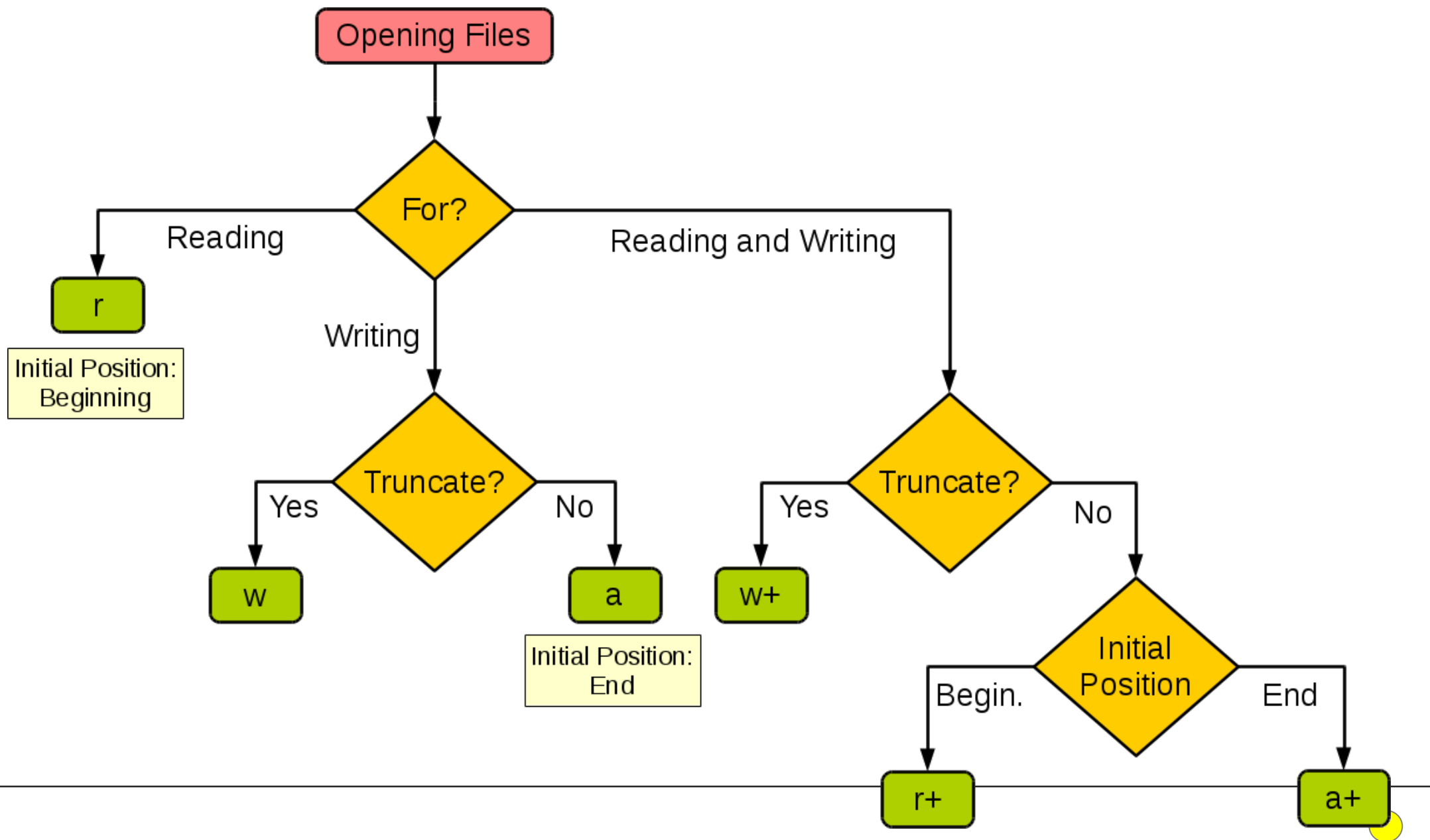
可以参考对比C语言文件操作

python通过 `open()` 函数打开一个文件对象，一般的用法为 `open(filename, mode)`，其完整定义为 `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)`。

`filename` 是打开的文件名，`mode` 的可选值为：

- t 文本模式 (默认)。
- x 写模式，新建一个文件，如果该文件已存在则会报错。
- b 二进制模式。
 - 打开一个文件进行更新(可读可写)。
- r 以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
- rb 以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。一般用于非文本文件如图片等。
- r+ 打开一个文件用于读写。文件指针将会放在文件的开头。
- rb+ 以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。一般用于非文本文件如图片等。
- w 打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
- wb 以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。
- w+ 打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。





• 文件读取

```
1 # readlines() 将会把文件中的所有行读入到一个数组中
2 f = open('test_input.txt')
3 print(f.readlines())
```

```
['testline1\n', 'testline2\n', 'test line 3\n']
```

```
1 # read() 将读入指定字节数的内容
2 f = open('test_input.txt')
3 print(f.read(8))
```

```
testline
```

```
1 # 但是一般情况下，我们
2 f = open('test_input.txt')
3 for line in f:
4     print(line)
```

```
testline1
```

```
testline2
```

```
test line 3
```

```
1 # 这种读入方法同样会保留行尾换行，结合print()自带的换行，
2 # 打印后会出现一个间隔的空行
3 # 所以一般我们读入后，会对line做一下strip()
4 f = open('test_input.txt')
5 for line in f:
6     print(line.strip())
```

```
testline1
```

```
testline2
```

```
test line 3
```

向文件写入

python中，通过文件对象的 `write()` 方法向文件写入一个字符串。

```
1 of = open('test_output.txt', 'w')
2 of.write('output line 1')
3 of.write('output line 2\n')
4 of.write('output line 3\n')
5 of.close()
```



字节文件的直接存取

```
f = open('test_input.txt', 'rb+')
f.write(b'sds0123456789abcdef')
f.seek(5)          # Go to the 6th byte in the file
print(f.read(1))
print(f.tell())
f.seek(-3, 2)      # Go to the 3rd byte from the end 0-1-2
print(f.read(1))
f.close()
```

b表示字节 → b' 2'
6
b' d'

Whence: 0代表从文件开头开始算起,
1代表从当前位置开始算起,
2代表从文件末尾算起

上下文管理器： with

```
1 with open('test_input.txt') as myfile:  
2     for line in myfile:  
3         print(line) ← 退出自动关闭文件  
4 myfile.closed == 1
```

sds0123456789abcdef

hello world!

True



Python的异常处理

- 常规的异常处理流程
- 自定义与触发异常



Python Errors and Built-in Exceptions

- 错误处理导致异常：软件的结构上有错误，导致不能被解释器解释或编译器无法编译。这些些错误必须在程序执行前纠正。
- 程序逻辑或不完整或不合法的输入、值域不合法导致运行流程异常；



语法错误、值域溢出或无法执行导致异常

```
# We can notice here that a colon is missing in the if statement.
```

```
if a < 3
```

```
File "<ipython-input-5-607a69f69f94>", line 1
```

```
    if a < 3
```

```
        ^
```

```
SyntaxError: invalid syntax
```

```
# ZeroDivisionError: division by zero
```

```
1 / 0
```

```
-----  
ZeroDivisionError
```

```
t call last)
```

```
<ipython-input-6-b710d87c980c> in <module>()  
----> 1 1 / 0
```

```
ZeroDivisionError: division by zero
```

```
# FileNotFoundError
```

```
open("imaginary.txt")
```

```
-----  
FileNotFoundError
```

```
Traceback (m
```

```
t call last)
```

```
<ipython-input-7-1f07e636ec19> in <module>()  
----> 1 open("imaginary.txt")
```

```
FileNotFoundError: [Errno 2] No such file or directory  
ary.txt'
```

```
Traceback (most recen
```



内建异常处理流程：

- 异常：是因为程序出现了错误而在正常控制流以外采取的行为，python用异常对象(exception object)来表示异常。遇到错误后，会引发异常。
- 两个阶段：
 - 引起异常发生的错误，
 - 检测（和采取可能的措施）阶段。
- 当前流将被打断，用来处理这个错误并采取相应的操作。这就是第二阶段，异常引发后，调用很多不同的操作可以指示程序如何执行。
- 如果异常对象并未被处理或捕捉，程序就会用所谓的回溯（traceback）终止执行
- 我们可以使用`local().__builtins__`来查看所有内置异常，如右图所示。

```
ans = locals()['__builtins__'].__dict__  
for k, v in ans.items():  
    if "Error" in k:  
        print(k, v)
```

```
TypeError <class 'TypeError'>  
ImportError <class 'ImportError'>  
ModuleNotFoundError <class 'ModuleNotFoundError'>  
OSError <class 'OSError'>  
EnvironmentError <class 'OSError'>  
IOError <class 'OSError'>  
EOFError <class 'EOFError'>  
RuntimeError <class 'RuntimeError'>  
RecursionError <class 'RecursionError'>  
NotImplementedError <class 'NotImplementedError'>  
NameError <class 'NameError'>
```



Python Built-in Exceptions

Exception	Cause of Error
AssertionError	Raised when <code>assert</code> statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the <code>input()</code> functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's <code>close()</code> method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.



Python 异常处理流程

- 当有异常出现时，它会使当前的进程停止，并且将异常传递给调用进程，直到异常被处理为止。
- 如：function A → function B → function C
- function C 中发生异常. 如果C没有处理，就会层层上传到B，再到A

```
def C(x):  
    x / (x-x)  
def B(x):  
    C(x)
```

```
def A(x):  
    B(x)
```

```
A(2)
```

ZeroDivisionError Traceback (most recent call last)

<ipython-input-1-cb9f0c9139a7> in <module>()
7 B(x)

8
----> 9 A(2)

<ipython-input-1-cb9f0c9139a7> in A(x)
5

6 def A(x):

----> 7 B(x)

8

9 A(2)

<ipython-input-1-cb9f0c9139a7> in B(x)
2 x / (x-x)

3 def B(x):

----> 4 C(x)

5

6 def A(x):

<ipython-input-1-cb9f0c9139a7> in C(x)
1 def C(x):

----> 2 x / (x-x)

3 def B(x):

4 C(x)

5

ZeroDivisionError: division by zero

Python中捕获与处理异常： try: ... except

* :

- 在Python中，可以使用try语句处理异常。
- 可能引发异常的关键操作放在try子句中，并且将处理异常的代码编写在except子句中。
- 如果没有异常发生，则跳过Except的内容，并继续正常流程。但是，如果发生任何异常，它将被Except捕获

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occured.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

```
The entry is a
Oops! <class 'ValueError'> occured.
Next entry.
```

```
The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.
```

```
The entry is 2
The reciprocal of 2 is 0.5
```

Except可以指定要捕获的异常类型:

```
try:
    # do something
    pass

except ValueError:
    # handle ValueError exception
    pass

except (TypeError, ZeroDivisionError):
    # handle multiple exceptions
    # TypeError and ZeroDivisionError
    pass

except:
    # handle all other exceptions
    pass
```

```
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
    except ValueError:
        print("Value Error")
    except (ZeroDivisionError):
        print("ZeroDivision Error")
print("The reciprocal of", entry, "is", r)
```

```
The entry is a
Value Error
The entry is 0
ZeroDivision Error
The entry is 2
The reciprocal of 2 is 0.5
```

主动触发异常 Raising Exceptions

- 在Python编程中，当运行时发生相应的错误时会引发异常，但是我们可以使用关键字raise强制引发它。
- 我们还可以选择将值传递给异常，以阐明引发该异常的原因。

```
try:
    a = int(input("Enter a positive integer: "))
    if a <= 0:
        raise ValueError(f"{a} is not a positive number!")
except ValueError as ve:
    print(ve)
```

```
Enter a positive integer: -3
-3 is not a positive number!
```



Try...finally语句

- Python中的try语句可以有一个可选的finally子句。该子句无论如何执行，通常用于释放外部资源。
- 例如，我们可能通过网络或使用文件或使用图形用户界面（GUI）连接到远程数据中心。
- 在所有这些情况下，无论资源是否成功，我们都必须清除该资源。这些操作（关闭文件，GUI或与网络断开连接）在finally子句中执行，以确保执行

```
try:
    f = open("test.txt",encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

```
-----
-----
FileNotFoundError                                Traceback (most recent call last)
<ipython-input-17-5a8f24f64426> in <module>()
```

```
1 try:
----> 2     f = open("test.txt",encoding = 'utf-8')
      3     # perform file operations
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'test.txt'
```

During handling of the above exception, another exception occurred:

```
NameError                                Traceback (most recent call last)
<ipython-input-17-5a8f24f64426> in <module>()
```

```
3     # perform file operations
4 finally:
----> 5     f.close()
```

```
NameError: name 'f' is not defined
```



ipython magic命令

python magic命令

ipython解释器提供了很多以百分号%开头的magic命令，这些命令很像linux系统下的命令行命令（事实上有些是一样的）。

查看所有的magic命令：

```
%lsmagic
```

Available line magics:

```
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cd
%clear %cls %colors %conda %config %connect_info %copy %ddir %debug %dhist
%dirs %doctest_mode %echo %ed %edit %env %gui %hist %history %killbgscripts
%ldir %less %load %load_ext %loadpy %logout %logon %logstart %logstate %log
stop %ls %lsmagic %macro %magic %matplotlib %mkdir %more %notebook %page %
pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint %precis
ion %prun %psearch %psource %pushd %pwd %pycat %pylab %qtconsole %quickref
%recall %rehashx %reload_ext %ren %rep %rerun %reset %reset_selective %rmdir
%run %save %sc %set_env %store %sx %system %tb %time %timeit %unalias %un
load_ext %who %who_ls %whos %xdel %xmode
```

Available cell magics:

```
%%! %%HTML %%SVG %%bash %%capture %%cmd %%debug %%file %%html %%javascript
%%js %%latex %%markdown %%perl %%prun %%pypy %%python %%python2 %%python3 %
%ruby %%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile
```



Available line magics:

```
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cd  
%clear %cls %colors %conda %config %connect_info %copy %ddir %debug %dhist  
%dirs %doctest_mode %echo %ed %edit %env %gui %hist %history %killbgscripts  
%ldir %less %load %load_ext %loadpy %logout %logon %logstart %logstate %log  
stop %ls %lsmagic %macro %magic %matplotlib %mkdir %more %notebook %page %  
pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint %precis  
ion %prun %psearch %psource %pushd %pwd %pycat %pylab %qtconsole %quickref  
%recall %rehashx %reload_ext %ren %rep %rerun %reset %reset_selective %rmdir  
%run %save %sc %set_env %store %sx %system %tb %time %timeit %unalias %un  
load_ext %who %who_ls %whos %xdel %xmode
```

Available cell magics:

```
%%! %%HTML %%SVG %%bash %%capture %%cmd %%debug %%file %%html %%javascript  
%%js %%latex %%markdown %%perl %%prun %%pypy %%python %%python2 %%python3 %  
%ruby %%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile
```

Automagic is ON, % prefix IS NOT needed for line magics.

line magic 以一个百分号开头, 作用与一行;

cell magic 以两个百分号开头, 作用于整个cell。



`line magic` 以一个百分号开头，作用与一行；

`cell magic` 以两个百分号开头，作用于整个cell。

使用 `whos` 查看当前的变量空间：

```
i = 5
a = 5
print(a is i)
j = 'hello world!'
a = 'hello world!'
print(a is j)
```

```
%whos
```

```
True
```

```
False
```

```
Variable      Type      Data/Info
```

```
-----
```

```
a             str      hello world!
```

```
b             list     n=4
```

```
i             int      5
```

```
j             str      hello world!
```

在Python中，整数和短小的字符，Python都会缓存这些对象，以便重复使用，不是频繁的建立和销毁。当创建多个等于整数常量的引用时，实际上是让这些引用会指向同一个对象



使用 `reset` 重置当前变量空间:

```
%reset -f
```

```
print (a)
```

```
-----  
-  
NameError                                Traceback (most recent call last)  
<ipython-input-12-a45fdcf41272> in <module>  
      1 get_ipython().run_line_magic('reset', '-f')  
      2  
----> 3 print (a)  
  
NameError: name 'a' is not defined
```

再查看当前变量空间:

```
%whos
```

Interactive namespace is empty.

Ipthon下常用的一些操作:

%cd 修改目录 例: %cd c:\\data

%ls 显示目录内容

%load 加载代码

%save 保存cell

%%writefile 命令用于将单元格内容写入到指定文件中
，文件格式可为txt、py等

%run 运行脚本

%run -d 交互式调试器

%timeit 测量代码运行时间 # %一行

%%timeit 测量代码运行时间 # %%一个代码块

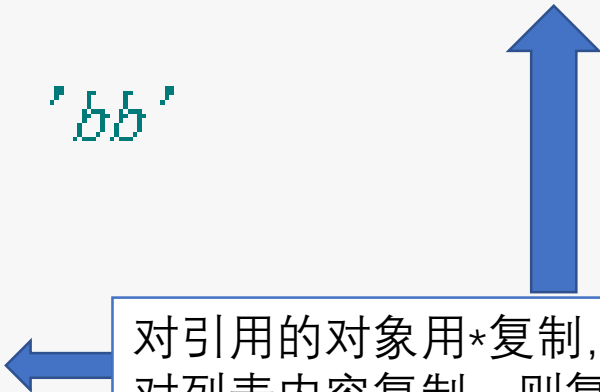


使用 `writetfile` 将cell中的内容写入文件:

```
%%writetfile test_magic.py
print ("%%开头的magic的作用区域延续到整个cell")

a = [3, 'aa', 34.4] * 4 # a = [[3, 'aa', 34.4]] * 4
print(a)
a[1] = 'bb' #a[0][1] = 'bb'
print (a)

b = [{'k1': 1.5}] * 4
b[0]['k1'] = 10
print(b)
```



对引用的对象用*复制，创建对象列表，复制引用，指向同一个对象
对列表内容复制，则复制所有对象

Overwriting test_magic.py

使用 `ls` 查看当前工作文件夹的文件:

使用 `run` 命令来运行这个代码:

```
: %ls
```

```
%run test_magic.py
```

驱动器 C 中的卷是 OS

卷的序列号是 8488-139B

C:\Users\hujf\2020notebooks\2020计概备课\Python_Basics-master\python_test 的目录

```
2020/10/21  06:47    <DIR>          .
2020/10/21  06:47    <DIR>          ..
2020/10/21  07:04                231 test_magic.py
```

1 个文件

231 字节

2 个目录 1,473,178,247,168 可用字节

%%开头的magic的作用区域延续到整个cell

```
[3, 'aa', 34.4, 3, 'aa', 34.4, 3, 'aa', 34.4, 3, 'aa', 34.4]
```

```
[3, 'bb', 34.4, 3, 'aa', 34.4, 3, 'aa', 34.4, 3, 'aa', 34.4]
```

```
[{'k1': 10}, {'k1': 10}, {'k1': 10}, {'k1': 10}]
```



ipython 使用帮助命令

使用 `?` 查看函数的帮助, 或: 光标移动到方法上面, 按 `shift+tab`, 弹出文档, 连续按选择文档详细程度

```
In [63]: input?
```

使用 `??` 查看函数帮助和函数源代码 (如果是用python实现的) :

```
In [65]: # 查看其中sort函数的帮助
input??
```

Signature: `input(prompt='')`

Source:

```
def raw_input(self, prompt=''):
    """Forward raw_input to frontends
```

Raises

`StdinNotImplementedError` if active frontend doesn't support stdin.
"""

```
if not self._allow_stdin:
    raise StdinNotImplementedError(
```