



现代程序设计复习总结

第一章:课程简介

1.面向对象简介

- 1. 将现实世界的事物抽象成对象
- 2. 包含属性和基于这些属性的行为
- 3. 对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性，灵活性和扩展性
- 4. 基本特点：封装，继承，多态

+ 添加一条 ↑

2.编译器和解释器

编译器：先整体编译再执行

编译方式：运行速度快，但任何一个小改动需要整体重新编译。可脱离编译环境运行

解释器：边解释边运行

解释方式：运行速度慢，但部分改动不需要整体重新编译。不可脱离解释器环境运行。

第二章:python 基础

1.关键字

```
import keyword
print(keyword.kwlist)
```

输出结果：

```
['False', 'None', 'True', 'and', 'as', 'assert',
'async', 'await', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise',
'return', 'try', 'while', 'with', 'yield']
```

Python ▾

2.上下文管理器（with....as....）

上下文管理器用于规定某个对象的使用范围。一旦进入或者离开该使用范围，会有特殊操作被调用（比如为对象分配或者释放内存）。它的语法形式是 with...as...

```
#不使用上下文管理器
f = open("new.txt", "w")
print(f.closed)           # whether the file is open
f.write("Hello World!")
f.close()
print(f.closed)
```

输出结果：

```
False
True
```

```
#使用上下文管理器
with open("new.txt", "w") as f:
    print(f.closed)
    f.write("Hello World!")
print(f.closed)
```

输出结果：

```
False
True
```

Python ▾

3.其他关键字的使用

- 1.None 和其他任何数据类型比较永远返回 False
- 2.nonlocal 在函数或其他作用域中使用外层（非全局）变量

#首先看这段代码

```
def wh():
    print('func',x,id(x))
x='main2'
wh()
print('func',x,id(x))
```

#输出结果为

```
func main2 2201459930608
func main2 2201459930608
```

#这说明在函数内部使用了与全局变量同名的变量，如果不对该变量赋值（修改变量），那么该变量就是全局变量

```
def wh():
    x='main1'
    print('func',x,id(x))
```

```
x='main2'
wh()
print('func',x,id(x))
```

#输出结果为

```
func main1 2373558983792
func main2 2373559039472
```

#这说明如果对函数内部变量进行赋值，那么该变量就是局部变量

```
def wh():
    global x
    print('func',x,id(x))
    x='main1'
    print('func',x,id(x))
x='main2'
print('func',x,id(x))
wh()
print('func',x,id(x))
```

#输出结果为

```
func main2 1721140622640
func main2 1721140622640
func main1 1721140567152
func main1 1721140567152
```

#这说明 global 首先将局部变量变为全局变量，并且对变量的修改也是全局的

#当出现函数嵌套时：

```
def func():
    print('func1',x,id(x))
    def ifunc():
        print('ifunc',x,id(x))
    ifunc()
```

```
x='main'
print('main1',x,id(x))
func()
print('main2',x,id(x))
```

#输出结果：

```
main1 main 2870455071088
func1 main 2870455071088
ifunc main 2870455071088
main2 main 2870455071088
```

#说明嵌套函数也默认使用全局变量

#当在函数内部定义 x 时：

```
def func():
    x='mainn'
    print('func1',x,id(x))
    def ifunc():
        print('ifunc',x,id(x))
    ifunc()
```

```
x='main'
print('main1',x,id(x))
func()
print('main2',x,id(x))
```

#输出结果为：

```
main1 main 2253447065968
func1 mainn 2253447027824
ifunc mainn 2253447027824
main2 main 2253447065968
```

#这说明内层函数使用外层函数定义的 x

#那如果内层函数想要修改外层函数定义的 x，又不修改全局变量 x 呢？

#这就需要用到 nonlocal

```
def func():
```

```
x='mainn'
print('func1',x,id(x))
def ifunc():
    nonlocal x
    x='mainnn'
    print('ifunc',x,id(x))
ifunc()
print('func1',x,id(x))

x='main'
print('main1',x,id(x))
func()
print('main2',x,id(x))
#输出结果为：
main1 main 2205505974640
func1 mainn 2205505936496
ifunc mainnn 2205510441840
func1 mainnn 2205510441840
main2 main 2205505974640
```

Python ▾

4.多语句

- 1.可以用； 在同一行显示多条语句
- 2.语句很长时可使用\来实现多行语句
- 3.在 [], {}, 或 () 中的多行语句不需要使用\

```
#示例代码
list1=['and'
,'but']
list2=['and',
'but']
print(list1,
list2)

#输出结果：
['and', 'but'] ['and', 'but']
#这说明对于 [], {}, 或 () 中的多行语句，可以在','的左右两侧直接换行，不用\

a=200+\
    300
b=200\
    +300
print(a,b)
#输出结果为
500 500
#这说明对于不在[], {}, 或 () 中的多行语句，可以在运算符的左右两侧使用\换行
```

Python ▾

5.标识符

在程序中自定义的类名、函数名、变量等符号和名称，叫做标识符

- 1.标识符由字母、数字、下划线(_)组成
- 2.所有标识符可以包括英文、数字以及下划线，但不能以数字开头
3. 以下划线开头的标识符有特殊意义
 - 以单下划线开头代表不能直接访问的类属性
 - 以双下划线开头代表类的私有成员
 - 以双下划线开头和结尾代表 Python 里特殊方法专用的标识 （比较关键）

6.变量

- 1.变量不需要声明
- 2.每个变量在使用前都必须赋值，赋值以后该变量才会被创建
- 3.变量没有类型（赋什么值，就是什么类型）
4. del 语句删除对象引用，删除后不能再引用，除非再次赋值
- 5.获取变量所指对象的内存地址：id(var)

```
a=2
print(id(a))
del a
print(id(2))
c=2
print(id(c))
#输出结果：
140705427726768
140705427726768
140705427726768
#之所以会三个一样，因为
```

7.数据类型

1.变量所指的内存中对象的类型

Number(int, bool, float, complex): 数字

String(str): 字符串

List(list): 列表

Tuple(tuple): 元组

Set(set): 集合

Dictionary(dict): 字典

2.类型的划分

不可变数据: Number、String、Tuple

可变数据: List、Dictionary、Set

3.类型的查询

type() 函数可以用来查询变量所指的对象类型

type()不会认为子类是一种父类类型

4.类型的判断

isinstance(a, int)可以用来判断是否是某类型

isinstance()会认为子类是一种父类类型

5.数据类型不允许改变,这就意味着如果改变数字数据类型的值，将重新分配内存空间:

```
a=20
b=20
if (id(a)==id(b)):
    print(1)
#结果会输出 1
a=20
b=30
if (id(a)==id(b)):
    print(1)
#不输出 1
a=[1]
b=[1]
if (id(a)==id(b)):
    print(1)
#不输出 1
#这是为提高内存利用效率,对于简单对象如 int 对象或字符串对象等，会采取重用对象内存的办法，不同的解释器可能有不同的实现
#这也解释了上一段代码为什么有同样的输出
```

Python ▾

6.数字类型运算

（一）True 和 False 关键字的值是 1 和 0，它们可以和数字相加

```
if True:
    print(1)
if False:
    print(2)
if 1:
    print(3)
if -1:
    print(4)
if 0:
    print(5)
if 2:
    print(6)
#输出结果为：1，3，4，6
#这说明除了 if 0 之外，其他的值都会使判断成立
```

Python ▾

（二）除法与整除

/: 除法，得到浮点数

//: 除法，得到整数 (不一定，分子分母为浮点时得到浮点)

```
print(3/2)
print(10/3)
print(5//2)
print(3.1//2.5)
#输出结果为
1.5
3.3333333333333335
2
1.0
```

Python ▾

(三) 比较状态可以传递

```
a,b=2,3
print(a<b==3)
#结果为 True
#当且仅当两个比较运算符都成立是为 True
```

Python

7.字符串类型

- 单引号和双引号使用完全相同
- 使用三引号(''或''')可以指定一个多行字符串。
- 转义符 '\', 使用r可以让反斜杠不发生转义

```
#三引号指定多行字符串
a='''woghwoewoihg
owhgw'''
print(a)
#输出结果为
woghwoewoihg
owhgw
#r使反斜杠不发生转义
print('\n')
print(r'\n')
#输出结果为

\n
#自动拼接字符串
a='this ' 'is ' 'string'
print(a)
#输出结果为 : this is string
```

Python

- 字符串索引：字符串有两种索引方式，从左往右以 0 开始，从右往左以 -1 开始。

```
#字符串不能改变
s='wwwwwww'
s[0]='a'
s+='b'
#第二，第三行均会报错 :TypeError: 'str' object does not support item assignment
#若想修改：
s='wwwwww'
s='a'+s[1:]
s=s+'b'
print(s)
#输出结果为：awwwwwb
#注意：由于字符串是不可变类型，s与s+'b'不具有相同的地址
```

Python

•字符串常用函数

```
s='wolai'
print(s.find('o'))
print(s.find('z'))
#输出结果为 1，-1
#这说明当 find的对象不再s中时，返回-1，根据上面对 if 语句的测试，if -1 会执行代码块，所以不要用 if s.find('z') 进行分支

#strip()方法用于移除字符串头尾指定的字符（默认为空格或换行符）或字符序列
s='wolaiww'
print(s.strip('w'))
#输出结果为 olai

#注意这里其实并未改变 s，而是副本
s='wolaiww'
s.strip('w')
print(s)
#输出结果为： wolaiww

#split()通过指定分隔符对字符串进行切片，如果第二个参数 num 有指定值，则分割为 num+1 个子字符串。
str = "this is string example....wow!!!"
print (str.split( ))
print (str.split('i',1))
print (str.split('w'))
#输出结果为：
['this', 'is', 'string', 'example....wow!!!']
['th', 's is string example....wow!!!']
['this is string example....', 'o', '!!!']

#Python zfill() 方法返回指定长度的字符串，原字符串右对齐，前面填充0
str = "this is string example....wow!!!"
print str.zfill(40)
```

```
#输出结果为：
00000000this is string example...wow!!!
```

```
#判断使用 zfill 的两个字符串是否有同一地址
a=''
b=a.zfill(10)
print(id(a))
print(id(b))
#结果：两个 id 不同
```

Python ▾

8.列表类型

```
list1=['a','b','c']
list2=['d','e','f']
print(list1*2)
print(list1+list2)
#输出结果为：
['a', 'b', 'c', 'a', 'b', 'c']
['a', 'b', 'c', 'd', 'e', 'f']
```

```
list3=['a','b','c','d','e','f','g','h','i','j','k','l']
print(list3[1:9:2])
#输出结果为：
['b', 'd', 'f', 'h']
#说明第一个数字为起点，第二个为终点，包括起点但不包括终点
```

```
nl=[0,1,2,3,4,5,6,7,8,9,10]
print(nl[0:2])
print(nl[:2])
print(nl[1:2])
#输出结果为：
[0, 2, 4, 6, 8, 10]
[0, 2, 4, 6, 8, 10]
[1, 3, 5, 7, 9]
#这说明起点缺省时为 0，终点缺省时为 len (list)
```

```
ll=['A','B','C','D','E','F','G']
print(ll[-1::-1])
print(ll.reverse())
print(ll)
#输出结果为：
['G', 'F', 'E', 'D', 'C', 'B', 'A']
None
['G', 'F', 'E', 'D', 'C', 'B', 'A']
#说明；reverse () 的返回值是 None，但会对列表进行逆序操作
```

```
ll=['A','B','C','D','E','F','G']
del ll[0]
print(ll)
#输出结果为：
['B', 'C', 'D', 'E', 'F', 'G']
```

```
ll=['A','B','C','D','E','F','G']
ll.remove('A')
#输出结果为：
['B', 'C', 'D', 'E', 'F', 'G']
#说明：多个 A 时只 remove 最前面的一个
```

```
ll=['A','B','C','D','E','F','G','A']
ll.clear()
print(ll)
#输出结果为：
[]
```

```
ll=['A','B','C','D','E','F','G','A']
del ll[:]
print(ll)
#输出结果与上相同，注意，不能使用 del ll，会报错
```

```
list1=['A','B','C']
list2=['E','F','G']
list1.append('D')
print(list1)
list1.extend(list2)
print(list1)
#输出结果为：
['A', 'B', 'C', 'D']
['A', 'B', 'C', 'D', 'E', 'F', 'G']
#说明：append 是 append 一个元素，extend 是 extend 另一个列表；另：两个函数返回值都是 None
```

```
#复制：会有深浅复制的问题：
list1=['A','B','C']
list2=list1
list1.append('D')
print(list2)
#输出结果为：
```

```
[ 'A', 'B', 'C', 'D']
#为了避免这个情况，引入 copy：
list1=[ 'A', 'B', 'C']
list2=list1.copy()
list1.append('D')
print(list2)
#输出结果为：
[ 'A', 'B', 'C']

#一些推导式的例子：
a=[x**2 for x in range(6)]
#a=[0, 1, 4, 9, 16, 25]
a=list(map(lambda x:x**2,range(6)))
#a=[0, 1, 4, 9, 16, 25]
pis=[str(round(math.pi, i)) for i in range(1,6)]
#pis=['3.1', '3.14', '3.142', '3.1416', '3.14159']
a=[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
#a=[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
#上式最后一个和 zip 的区别：
#zip 是按位置对应结合成元组，而上式是分别遍历合成元组
x=[1,2,3]
y=[3,1,4]
z=zip(x,y)
for i in range(3):
    a=next(z)
    print(a)
#结果为：
(1, 3)
(2, 1)
(3, 4)

#内存分配：
list1=[1,2,3]
list2=[1,2,3]
print(id(list1))
print(id(list2))
#输出的两个 id 不同

list1=[1,2,3]
print(id(list1))
list1.append(4)
print(id(list1))
#输出的两个 id 相同

list1=[1,2,3]
print(id(list1))
list2=list1
print(id(list2))
#输出的两个 id 相同

list1=[1,2,3]
print(id(list1))
list2=list1[:]
print(id(list2))
#输出的两个 id 不同

#二维列表（矩阵）
matrix=[[1,4,7],[2,5,8],[3,6,9]]
print(matrix)
matrix2=[[row[i] for row in matrix] for i in range(3)] #实现矩阵交换行列，本质上是在按列遍历矩阵
print(matrix2)
#输出结果为：
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

#按列遍历矩阵：
for i in range(3):
    for j in matrix:
        print(j[i],end=' ')
#输出结果为：
1 2 3 4 5 6 7 8 9
```

Python ▾

9.元组类型：与列表类似，不同之处在于元素不可修改

```
#虽然 tuple 的元素不可改变，但它可以包含可变的对象
#尝试以下代码：
tup1=( [1,2,3],0)
tup1[0]=[1,2,3,4]
#第二行报错：TypeError: 'tuple' object does not support item assignment
#与字符串类似，元组类型不可改变，但若改为如下代码：
tup1=( [1,2,3],0)
tup1[0].append(4)
print(tup1)
#输出结果为：
```

```
([1, 2, 3, 4], 0)

#构造 0 个或 1 个元素的元组
tup1 = ()
tup2 = (20,) # 一个元素，需要在元素后添加逗号，否则含义不明确
#注：元组中的元素值不允许删除，但 del 能删除整个元组
```

Python

10.集合（set）：一般用于进行成员关系测试和删除重复元素

```
#集合的创建：
#可以使用大括号 { } 或者 set() 函数创建集合
s1={1,2,3,1}
value=[1,2,3,4,3]
s2=set(value)
print(s1)
print(s2)
#输出结果为：
{1, 2, 3}
{1, 2, 3, 4}
#这说明 set 会自动去重
#注：创建空集合必须用 set() 而非 { }, { } 用来创建空字典

#集合的运算
s1={1,2,3,5}
s2={1,2,3,4}
print(s1-s2) #集合的差
print(s2-s1)
print(s1|s2) #集合的并
print(s1&s2) #集合的交
#输出结果为：
{5}
{4}
{1, 2, 3, 4, 5}
{1, 2, 3}

#元组的元素批量添加：
s1={1,2,3}
s2={2,3,4,5}
s1.update(s2)
print(s1)
#输出结果为：
{1, 2, 3, 4, 5}

#集合不能包括可变类型：
list1=[1,2,3]
list2=[4,5,6]
s1={list1,list2}
print(s1)
#第三行报错：TypeError: unhashable type: 'list'

#包含 list 的元组能否加入集合：
tup1=([1,2,3],0)
tup2=(1,2)
s1={tup1,tup2}
print(s1)
#结果是不能，第三行报错：TypeError: unhashable type: 'list'
```

Python

11.字典：一种映射类型，其元素是键值对，无序的键(key)：值(value) 集合

注意：1.键(key)必须使用不可变类型

2.同一个字典中键(key)必须是唯一的

```
#构建字典的几种方式：
d1=dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
d2=dict(sape=4139, guido=4127, jack=4098)
d3=dict(list(enumerate(['one', 'two', 'three'],start=1))))
print(d1,d2,d3)
#输出结果为：
{'sape': 4139, 'guido': 4127, 'jack': 4098}
{'sape': 4139, 'guido': 4127, 'jack': 4098}
{1: 'one', 2: 'two', 3: 'three'}
#说明：enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标
#start=1说明下标从 1 开始
l1=list(enumerate(['one', 'two', 'three'],start=1))
print(l1)
#输出结果为：
[(1, 'one'), (2, 'two'), (3, 'three')]

#如何让字典按键排序输出：
for key in sorted(dic.keys()):
```



```
print(key,dic[key])

#如何构建有序字典，引入collections库：
import collections
unordered_d=dict([(3, 'sape'),(1, 'guido'),(2, 'jack')])
dic=collections.OrderedDict()
dic=collections.OrderedDict(sorted(unsorted_d.items(), key=lambda dc:dc[1], reverse = True))
print(dic)
#输出结果为：
OrderedDict([(3, 'sape'), (2, 'jack'), (1, 'guido')])
#reverse=True代表倒序
#常规dict并不跟踪插入顺序，迭代处理会根据键在散列表中存储的顺序来生成值。在OrderDict中则相反，它会记住元素插入的顺序，
#也就是说：
import collections
print 'Regular dictionary:'
d = {}
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'
for k, v in d.items():
    print k, v
print '\nOrderDict:'
d = collections.OrderedDict()
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'
for k, v in d.items():
    print k, v
#输出结果为：
Regular dictionary:
a A
c C
b B
OrderDict:
a A
b B
c C
```

Python ▾

12. is 和==的区别

is 用于判断两个变量引用对象是否为同一个
== 用于判断引用变量的值是否相等

```
#示例
list1=[1,2,3,4]
list2=[1,2,3,4]
if list1 is list2:
    print(1)
if list1==list2:
    print(2)
#输出结果为：
2

#实际上 list1 is list2 表示 id(list1) ==id(list2) ，由于列表是可变类型，两个值相同的列表在创建时不会指向同一个地址
#但是我们可以利用深浅复制来让程序输出1
list1=[1,2,3,4]
list2=list1
if list1 is list2:
    print(1)
if list1==list2:
    print(2)
#输出结果为：
1
2
```

Python ▾

13 强制类型转换

```
#示例
#repr()与str()类似，区别在于：
#str() 的输出追求可读性，输出格式要便于理解，适合用于输出内容到用户终端
#repr() 的输出追求明确性，除了对象内容，还需要展示出对象的数据类型信息，适合开发和调试阶段使用
from datetime import datetime
now = datetime.now()
print(str(now))
print(repr(now))
#输出结果为：
2022-01-07 19:10:19.941179
datetime.datetime(2022, 1, 7, 19, 10, 19, 941179)

#eval(str) 用来计算在字符串中的有效Python表达式,并返回一个对象，简单说就是去掉字符串的引号
a='3+4'
b=eval(a)
print(b)
```

```
#输出结果为：
7

#一些其他的类型转换
#frozenset(s) 可将 s 转换为不可变集合
#chr(x) 将一个整数转换为一个字符
#ord(x) 将一个字符转换为它的整数值
#hex(x) 将一个整数转换为一个十六进制字符串
```

Python

14 随机函数

```
#choice
import random
list1=[1,2,3,4]
print(random.choice(list1)) #从 list1里随机挑一个

#randrange 从指定范围内，按指定基数递增的集合中获取一个随机数，基数默认值为 1
print(random.randrange(1,10,2))#表示从 1到 10 中，步长为 2，随机挑选，也就是从（1，3，5，7，9）中随机挑选

#random() 随机生成下一个实数，在[0,1)范围内
print(random.random()) #不能设置范围，只能是 0-1

#seed([x]) 改变随机数生成器的种子
random.seed(1)
print(random.random())
#以上代码运行两次，结果一样，说明种子固定，随机数固定

#shuffle(list) 将序列的所有元素随机排序
list1=[1,2,3,4]
random.shuffle(list1)
print(list1)

#uniform(x, y) 随机生成下一个实数，在[x,y]范围内
```

Python

+ 添加一条 ↑

第三章:控制流和函数

一.控制流

1.条件控制：简单的 if –elif–else

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

Python

注：1.如果只有一条语句，可以写在一行
例：if test<10: print(test)
2.在嵌套 if 语句中，可以 if...elif...else 结构放在另外一个 if...elif...else 结构中

2.While 循环：

```
n=100
sum=0
counter=1
while counter<=n:
    sum=sum+counter
    counter+=1
```

Python

注：1.没有 do...while 循环
2.无限循环：while True（使用 CTRL+C 可退出当前的无限循环）（这好像是对苹果说的，我的电脑退出不了）
3.while ... else 在条件语句为 False 时执行 else 的语句块

```
#while else 示例：
a=1
while a<10:
    a+=2
    print(a)
else:
    print(a)
#输出结果为：
3
```

```
5
7
9
11
11
```

Python ▾

3.for 循环（也可以与 else 合用）

```
for i in range(3):
    print(i)
else:
    print(10)
#输出结果为：
0
1
2
10
```

Python ▾

4.break 和 continue

break 语句可以跳出 for 和 while 的循环体，且对应循环的 else 块不执行
continue 语句跳过当前循环块中的剩余语句，然后继续进行下一轮循环

二.函数 def func()

1.函数第一行语句应用文档字符串进行函数说明

第一行关于函数用途的简介，这一行应该以大写字母开头，以句号结尾
如果文档字符串有多行，第二行应该空白，与之后的详细描述明确分隔
详细描述应有一或多段以描述对象的调用约定、边界效应等

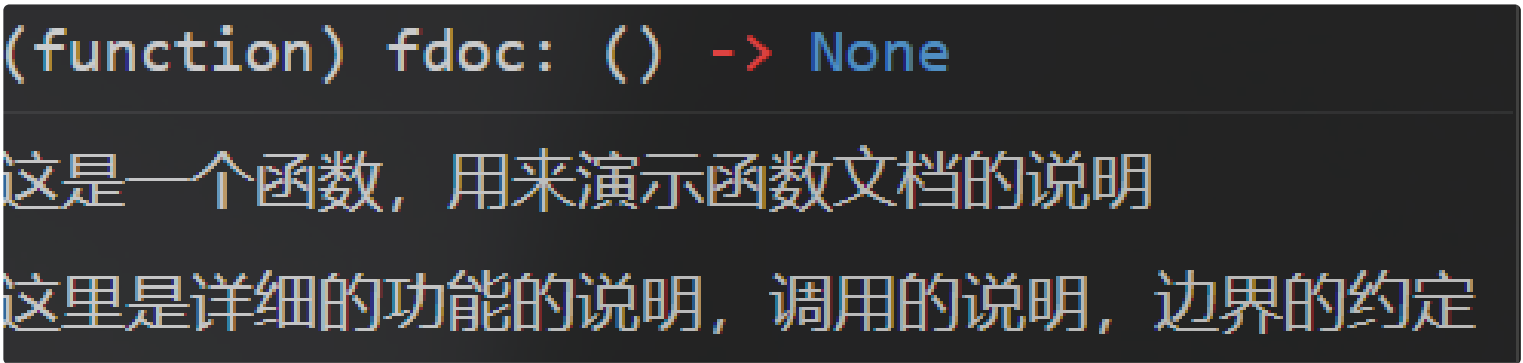
```
def fdoc():
    '''这是一个函数，用来演示函数文档的说明

    这里是详细的功能的说明，调用的说明，边界的约定'''
    pass
print(fdoc.__doc__)
#输出结果为：
这是一个函数，用来演示函数文档的说明

    这里是详细的功能的说明，调用的说明，边界的约定
```

Python ▾

显示效果为：



2.参数传递

(1)不可变类型：如数、字符串和元组

传递的只是a的值，没有影响a对象本身，在 fun(a)内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身

```
def fa(a):
    a=100
    print(hex(id(a)))
    print(hex(id(100)))
a=10
print(hex(id(a)))
print(hex(id(10)))
fa(a)
print(hex(id(a)))
print(hex(id(10)))
#输出结果为：
0x7ff88909a2b0
0x7ff88909a2b0
0x7ff88909adf0
0x7ff88909adf0
0x7ff88909a2b0
0x7ff88909a2b0
#说明：首先程序给 a 赋值 10，由于数值是不可变类型，a 和 10 具有同样的地址，所以前两行输出结果一样
#在调用 fa（a）时，fa 函数中对 a 进行了修改，实际上是对 a 的副本进行了修改，由于指向了 100，所以 a 的副本和 100 具有相同的地址
#最后，函数内部的修改不影响全局变量，所以外部的 a 还是和 10 同样的地址
```

Python ▾

(2)可变数据类型：如列表，字典，集合

如 fun(L)修改后 fun 外部的L 也会受影响

```
def fc(a):
    a.append(100)
    print(hex(id(a)))
l=[1,2,3]
print(l)
print(hex(id(l)))
fc(l)
print(l)
#输出结果为：
[1, 2, 3]
0x229b898db88
0x229b898db88
[1, 2, 3, 100]
#说明：由于列表是可变数据类型，函数内部的操作影响全局变量，且可变数据类型的修改不会改变地址~
```

Python

3.参数类型

(1)必须参数，好像也叫位置参数

必须以正确的顺序传入函数，调用时数量必须和声明时一样（最正常的情况）

(2)关键字参数

使用关键字参数来确定传入的参数值，关键字的参数应跟随在位置参数后
允许函数调用时参数的顺序与声明时不一致，解释器能够根据参数名匹配参数值

```
def fun(name,key):
    print(name)
    print(key)
fun(key='lambda x:x[1]',name='test')
#输出结果为：
test
lambda x:x[1]
#好像意思就是传参的时候把参数在函数里的名字给了，它就会自己对应上
```

Python

(3)默认参数

调用函数时，如果没有传递参数，则会使用默认值

```
def fun(name='zjc',key):
    print(name)
    print(key)
fun(key='lambda x:x[1]')
#这样会报错，带有缺省值的变量必须在位置变量后面
def fun(key,name='zjc'):
    print(name)
    print(key)
fun(key='lambda x:x[1]')
#输出结果为：
zjc
lambda x:x[1]
```

Python

默认值只被赋值一次，这使得当默认值是可变对象时会有所不同
比如列表、字典或者大多数类的实例，也即默认值在后续调用中会累积

```
def f(a, L=[]):
    print(hex(id(L)))
    L.append(a)
    return L
print(f(1))
print(f(2))
print(f(3))
#输出结果为：
0x275e361db88
[1]
0x275e361db88
[1, 2]
0x275e361db88
[1, 2, 3]
#相同的三次调用，输出结果不同，因为默认值在累积
#如何避免：
def f(a, L=None):
    if L is None:
        L = [] #每次调用重新进行初始化
    print(hex(id(L)))
    L.append(a)
    return L
```

```
print(f(1))
print(f(2))
print(f(3))
#输出结果为：
0x2314441db88
[1]
0x2314441db88
[2]
0x2314441db88
[3]
#为什么这样就可以了我也不知道，但他就是可以了
```

Python

(4)可变参数： 需要一个函数能够处理比声明时更多的参数， 且声明时不需要命名

1.加*的参数会以元组(tuple)的形式导入， 存放所有未命名的参数变量

```
def ptest( arg1, *vartuple ):
    print (arg1)
    print (vartuple)
ptest( 70, 'test', 50 )
#输出结果为：
70
('test', 50)
```

Python

2. 加**的参数会以字典的形式导入

```
def ptest2( arg1, **vardict ):
    print (arg1)
    print (vardict)
ptest2(10,a=2,b=4)
#输出结果为：
10
{'a': 2, 'b': 4}
```

Python

注意： 通常这些可变参数是参数列表中的最后一个

任何出现在不定长参数的后面的参数只能是关键字参数， 不能是位置有关参数

```
def concat(*args, sep):
    print(args)
    print(sep)
concat(1,2,3)
#会报错：TypeError: concat() missing 1 required keyword-only argument: 'sep'
#意思是 args 把传的参都用了，没有参数给 sep 了
#所以可以给 sep 一个缺省值
def concat(*args, sep='\n'):
    print(args)
    print(sep)
concat(1,2,3)
#输出结果为：
(1, 2, 3)
/
#或者用关键词传参：
def concat(*args, sep):
    print(args)
    print(sep)
concat(1,2,sep=3)
#输出结果为：
(1, 2)
3
```

Python

注意： 声明函数时， 参数中星号 可以单独出现

星号是分隔符， 后面的参数必须用关键字传入， *本身不是参数

```
def f(a,b*,c):
    return a+b+c
f(1,2,3)
#报错：TypeError: f() takes 2 positional arguments but 3 were given
def f(a,b*,c):
    return a+b+c
print(f(1,2,c=3))
#输出结果为 6
```

Python

5.参数分拆

要传递的参数已经是一个数据结构如列表等， 但要调用的函数却只接受分成一个一个的参数

```
args = [3, 6]
print(*args) #列表分拆(*)
```

```
d={'name': 'zjc', 'age': 36, 'job': 'prof.'}
def printInfo(name, age, job):
    print("Name:{0}\tAge:{1}\tJob:{2}".format(name, age, job))
    pass
printInfo(**d) #字典分拆(**)
#输出结果为：
3 6
Name:zjc      Age:36   Job:prof.
```

Python

6.匿名函数

不再使用def 语句标准的形式定义函数，使用lambda来创建匿名函数

```
#语法：lambda [arg1 [,arg2,.....argn]]:expression，冒号前是参数，可以有多个，用逗号隔开，冒号右边的为表达式（只能为一行）
sum = lambda x,y : x+y
print(sum(1,2))
#输出结果为：3
```

Python

7.return

不带参数的return 语句返回None，没有return 语句自动返回None

8.函数嵌套：

```
def func():
    a=2
    b=3
    def func_inner():
        return a*b
    c=func_inner()
    return c
print(func())
#输出结果为：6
#嵌套函数就是在外函数的内部现场定义一个函数来调用，内部函数可以使用外部函数的变量
#内函数同时可以用lambda定义：
def make_incrementor(n):
    return lambda x: x + n
f=make_incrementor(1)
print(f)
print(f(0))
print(make_incrementor(1)(0))
#输出结果为：
<function make_incrementor.<locals>.<lambda> at 0x00000017B28128E58>
1
1
#说明：仅传一个参时，返回的是由lambda定义的一个函数对象
```

Python

9.闭包

在一个外函数中定义了一个内函数，内函数里运用了外函数的变量，并且外函数的返回值是对内函数的引用

```
def outer(x):
    b=[x] #python2.x的时候用
    def inner(y):
        nonlocal x #python3.x的时候用
        x+=y
        b[0]+=y
        return x
    return inner

f1=outer(10)
print(f1(1))
#输出结果为11
f1=outer(10)
print(f1(2))
#输出结果为12
```

Python

10.函数注解

- 1.参数注解：定义在参数名称的冒号后，紧随着一个用来表示注解的值表达式
- 2.返回注释（Return annotations）是定义在一个 -> 后面，紧随着一个表达式，在冒号与 ->之间

```
def f(ham: 42, eggs: str = 'spam') ->"Nothing to see here":
    pass
```

Python

第四章：模块和输入输出

一.模块（module）：一个Python文件，以.py结尾，包含了对象定义和可执行语句

模块能定义函数，类和变量，模块里也能包含可执行的代码

目的是：1.更有逻辑和层次地组织Python代码

2.相关的代码进行单独的组织会使代码更容易理解并被复用

1.__name__：模块的模块名（字符串）可以由全局变量__name__得到

__name__是python的一个内置类属性，它天生就存在于一个python 程序中
直接运行python程序时，__name__的值为“__main__”
而在其它程序中导入.py文件运行时，__name__的值为文件名，即模块名
也就是说调用别的模块时，别的模块的__name__的值为模块名

Python

2.import

当解释器遇到import语句时，如果模块在当前的搜索路径就会被导入

搜索路径为当前目录或存储在sys.path的路径

模块在解释器会话中只导入一次，如果更新，需要重启解释器或者重新加载

不建议使用 from xxx import *

3.可执行语句

模块可以包含可执行语句，这些语句一般用来初始化模块，其仅在第一次被导入的地方执行一次

4.全局变量

(1).模块私有符号表，并被模块内所有的函数作为全局符号表使用

(2).是模块内部可以使用全局变量，而无需担心与其他模块全局变量冲突

(3).可通过模块访问其全局变量：modname.itemname

5.加速和优化

在__pycache__目录下以module.version.pyc 名字缓存模块编译后的版本，编译后的模块是跨平台的

为了减少编译模块的大小，可在Python命令行中使用-O或者-OO“优化”

-O 参数删除了断言语句

-OO 参数删除了断言语句和 __doc__字符串

注意：仅在确定无误的场合使用这一选项，并非加速代码的执行速度，程序逻辑可能依赖__doc__等

6.包：就是模块的集合，在电脑中，包就是一个文件夹，里面的每一个.py文件就是一个模块

为了让Python将目录当做内容包，必须包含init.py文件，一般空白即可

包的导入：

1.可只导入包里的特定模块

import a.b.c

2.通过完整的名称来引用

a.b.c.fun()

3.from a.b import c

c.fun()

4.from a.b.c import fun

fun()

注意：

from package import *

如果包中的init.py代码定义了一个名为__all__的列表，就会按照列表中给出的模块名进行导入

如果没有定义__all__，则不会导入所有子模块

包内引用：

用点号标明关联导入当前和上级包

— from . import b

— from .. import d

— from ... import e

意思是文件夹里套文件夹的时候，每多一个点可以向上一级

二.命名空间

1.三种类型：

(1)内置名称:Python语言内置的名称，比如函数名abs和异常名称Exception等

(2)全局名称:模块中定义的名称，如函数、类、其它导入的模块、模块级的变量和常量等

(3)局部名称:函数中定义的名称，如函数的参数和局部变量等，或类中定义的函数及变量等

查找顺序:局部的命名空间->全局命名空间->内置命名空间

2.生存时间

Python中只有模块(module)，类(class)以及函数(def、lambda)才会引入新的作用域

其它的代码块（如if/elif/else/、try/except、for/while等）并不会引入新的作用域

在这些语句内定义的变量，在代码块外也可访问
3.作用域

四种作用域

1.L(Local)

- 最内层，包含局部变量，比如一个函数或方法内部

2.E(Enclosing)

- 包含了非局部(non-local)也非全局(non-global)的变量，
比如嵌套函数等，一个函数A 中包含了另一个函数B，那么对于B，A中的作用域就为 nonlocal

3. G(Global)

- 当前模块的最外层，比如当前模块的全局变量

—4.B(Built-in)

- 包含了内建的变量/关键字等

搜索规则顺序： L —> E —> G —>B

```
a=10
def fa():
    a=a+1
    print(a)
fa()
#以上代码无法运行，函数内部没有定义 a
```

Python ▾

三.输入输出

1.format 函数

```
#通过位置定位
print('a1 = {} a2= {} a3= {}'.format('first','second','third'))
print('a1 = {1} a2= {0} a3= {2}'.format('first','second','third'))
#输出结果为：
a1 = first a2= second a3= third
a1 = second a2= first a3= third

#通过关键字参数
print('your name is {name} , age is {age}'.format(age=87,name='jack'))
#输出结果为：
your name is jack , age is 87
#通过对象属性
class p():
    def __init__(self):
        self.name='wh'
        self.age=20
p=p()
print('name = {p.name} age = {p.age}'.format(p=p))
#输出结果为：
name = wh age = 20

dt = {'k1': 1, 'k2': 2, 'k3': 3}
print('k1: {0[k1]}; k2: {0[k2]}; k3: {0[k3]}'.format(dt))
print('k1: {k1}; k2: {k2}; k3: {k3}'.format(**dt))
#输出结果为：
k1: 1; k2: 2; k3: 3
k1: 1; k2: 2; k3: 3

#通过下标
s1='whh'
s2='hxx'
print('{0[1]} {0[2]} {1[2]} {1[0]}'.format(s1,s2))
#输出结果为：
h h x h

#对齐与填充
name='whh'
number=100
print('{0:10} ==> {1:10d}'.format(name,number))
#输出结果为：
whh          ==>        100

#浮点小数输出
print('常量 PI 的值近似为 {0:.3f}'.format(math.pi))
#输出结果为：
常量 PI 的值近似为 3.142。

#format()函数
- 格式化输出
• 进制及其他显示
- b ：二进制
- d ：十进制
- o ：八进制
- x ：十六进制
- !s ：将对象格式化转换成字符串
- !a ：将对象格式化转换成 ASCII
print('{!a}'.format('天津 tianjin'))
```


- !r : 将对象格式化转换成 repr

#f-strings: 语法更简洁，速度也更快

year = 2016

event = 'Ceremony'

print(f'Results of the {year} {event}')

#输出结果为：

Results of the 2016 Ceremony

Python ▾