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

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

```python
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...

```python
#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
```

Python

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:

```python
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

```python
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）

```python
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

```python
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

```python
#三引号指定多行字符串
a='''woghwioewoihg
owhgw'''
print(a)
#输出结果为
woghwioewoihg
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。

```python
#字符串不能改变
s='wwwwwwww'
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

```python
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)
```

```python
#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

```python
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：
```

```python
([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

```python
#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

```python
#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()):
```

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

#如何构建有序字典，引入 collections 库：
import collections
unsorted_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

```python
#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

```python
#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）

```python
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，与其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

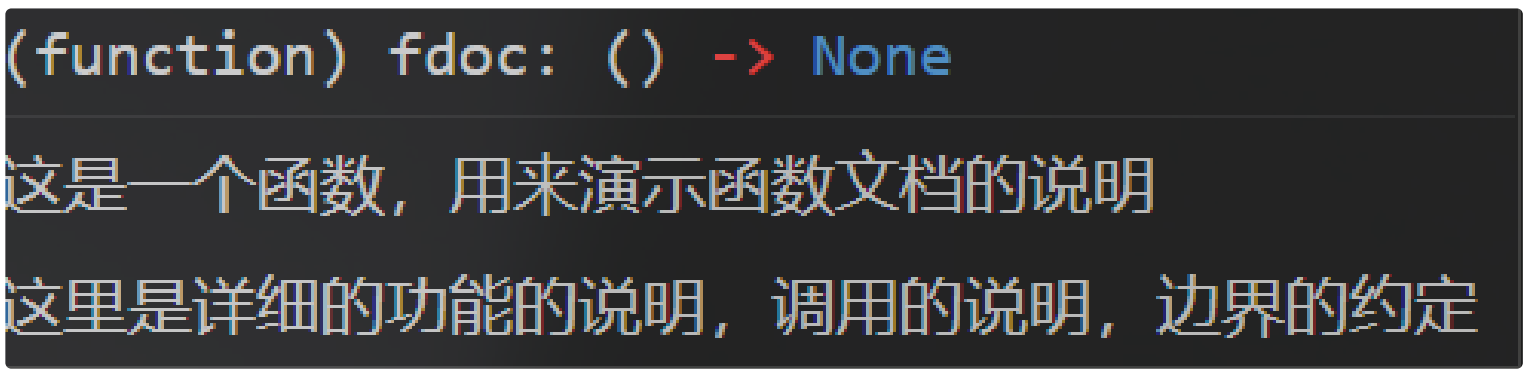
```python
def fdoc():
    '''这是一个函数，用来演示函数文档的说明

    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

```python
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

```python
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

```python
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

```python
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

```python
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='\'):
    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(*)
```

```python
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

```python
#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（只能为一个
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：

```python
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 0x0000017B28128E58>
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

```python
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

```python
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

```python
a=10
def fa():
    a=a+1
    print(a)
fa()
#以上代码无法运行，函数内部在定义 a，这时候编译器不知道等式右边的 a 是函数内部的还是外部的
```

Python

三.输入输出

1.format 函数

```python
#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： 语法更间洁，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

第五章：面向对象程序设计

1.面向对象概念：

一种程序设计范式，程序由对象组成，每个对象包含对用户公开的特定功能和隐藏的实现部分
对象是数据与相关行为的集合
不必关心对象的具体实现，只要能满足用户的需求即可

2.类：

对象的类型，用来描述对象
— 构造对象的模板
— 定义了该集合中每个对象所共有的属性和方法
— 由类构造对象的过程称之为实例化，或创建类的实例

一些重要概念

(1)多态
• 可以对不同类型的对象执行相同的操作
(2)封装
• 将数据和行为组合，并对外隐藏数据的实现方式
• 对象的状态：当下实例域的值的特定组合
(3)继承
• 通过扩展一个类来建立另外一个类

3.对象

(1)行为
通过可调用的方法定义　　#调用类里的函数
(2)状态
保存描述当前特征的信息
对象状态的改变必须通过调用方法实现
(3)标识
每个对象实例的标识应该是不同的

4.类和类之间的关系

(1)依赖
一个类的方法操纵另一个类的实例
耦合度及其最小化
(2)聚合
类A包含类B的实例对象
(3)继承
类A由类B扩展而来
如果类A扩展类B，类A不但包含从类B继承的方法,还会拥有一些额外的功能

```
#类之间依赖的例子
import os
import sys

class Course:
  '''Course class
  properties: id, name, credits,lecturer
  methods: print_course, set_credicts
  '''
  ID=0
  def __init__(self,name,credits,lecturer):
    self.id=Course.ID
    self.name=name
    self.credits=credits
    self.lecturer=lecturer
    Course.ID+=1


  def print_course(self):
    print(f"{self.id}\t{self.name}\t{self.credits}\t{self.lecturer}")

class Student:
  def __init__(self,name):
    self.name=name
```

```
    self.courses=[]

  def choose_course(self,c:Course):
    self.courses.append(c)

  def total_credits(self):
    total=0.
    for c in self.courses:
      total+=c.credits
    return total

def main():
    cc=Course('c',2,'z')
    cp=Course('python',2,'z')
    cc.print_course()
    cp.print_course()
    s=Student('l')
    s.choose_course(cc)
    print(s.total_credits())
    print(Course.ID)
    Course.print_course(cc)

if __name__=='__main__':main()

#在类 Student 里的 choose_course()方法，将类 Course 的实例作为参数传进去
```

Python

5.自定义类

使用 class 语句来创建一个新类，class 之后为类的名称并以冒号结尾

类的文档信息可以通过 ClassName.doc 查看
类的定义要先执行才能生效
可以在函数中或 if 等中进行类定义
类会创建一个新的命名空间

```
#老师的示例代码
def fmin(self,x,y):
  return min(x,y)

class TestNumber:
  f=fmin

  def f_wighoutself(a,b):
    pass

tn=TestNumber()
print(type(fmin))
print(type(TestNumber.f))
print(type(tn.f))
print(type(tn.f_wighoutself))
#输出结果为：
<class 'function'>
<class 'function'>
<class 'method'>
<class 'method'>
#说明：实例化之前叫 function，实例化之后是实例对象的一个 method
```

Python

6.类变量与实例变量

(1)类变量
类变量在所有实例化的对象中是公用的
类变量定义在类中且在类方法之外
类变量通常不作为实例变量使用
(2)实例变量
每个实例独有
第一次使用时自动生成
与类变量重名时仍作为实例变量

```
class wh():
    name='wh'  #类变量
    def __init__(self):
        self.n='whh' #实例变量
print(wh.name)
print(wh().n)
#输出结果为：
wh
whh
#说明：类变量是类本身就有的，实例变量是实例化之后每个实例单独有的
#当命名不冲突时：
class wh():
    name='wh'  #类变量
```

```
    def __init__(self):
        self.n='whh' #实例变量
print(wh().name)
#输出结果为
wh
#当命名冲突时：
class wh():
    name='wh'  #类变量
    def __init__(self):
        self.name='whh' #实例变量
print(wh().name)
#输出结果为:
whh
```

Python

```
#类的数据属性：可以在外面定义类内部的变量
class wh():
    name='wh'  #类变量
    def __init__(self):
        self.name='whh' #实例变量
    def pg(self):
        print(wh.age)
wh.age=20
wh().pg()
#输出结果为：
20
```

Python

7.类的方法

通过实例调用一个方法就相当于将该实例插入到参数列表的最前面，并通过类对象来调用相应的函数

```
class D:
    def f(self,a,b):
        self.a=a
        self.b=b
        print(self.a,self.b)
        pass
d=D()
a,b=10,20
d.f(a,b)
D.f(d,a,b)
#输出结果为：
10 20
10 20
#说明：d.f(a,b)和D.f(d,a,b)等价，通过实例调用一个方法就相当于将该实例插入到参数列表的最前面，并通过类对象来调用相应的函数
```

Python

方法定义不一定只在类中， 也可以将一个函数对象赋值给类中的一个局部变量

```
#在外部定义函数时，需要加一个 self 参数才能在类中使用
def f1(self,x,y):
    return min(x,y)
class C:
    f=f1
c=C()
print(c.f(2,3))
print(type(c.f))
#输出结果为:
2
<class 'method'>
```

Python

第六章 单例模式、继承、运算符重载

1.设计模式

(1)开闭原则

对扩展开放，对修改关闭

(2)里氏代换原则

任何基类可以出现的地方，派生类一定可以出现：即基类可被派生类替换

(3)依赖倒转原则

针对接口编程，依赖抽象而不依赖具体接口

(4)隔离原则

使用多个隔离的接口，比使用单个接口要好

降低类之间的耦合度

(5)最小知道原则

一个实体应当尽量少地与其他实体发生作用

系统功能模块应相对独立

(6)合成复用原则

尽量使用合成/聚合的方式，而不是使用继承

2.单例模式：全局只有一个实例　　实现时确保某一个类只有一个实例存在

```python
#重写__new__方法来实现单例模式
class Singleton:
  _instance=None
  def __init__(self, name, volume):
    self.name=name
    self.volume=volume

  def __new__(cls,name,volume):
    if not Singleton._instance:
      Singleton._instance=object.__new__(cls)
      Singleton.__init__(Singleton._instance,name,volume)
    return Singleton._instance

slist=[Singleton('z',100) for i in range(10)]
for s in slist:
  print(hex(id(s)),end='\t')
  print(f"{s.name}\t{s.volume}")
#我只能看懂他重写了__new__,然后就可以了.....
#注：该实现在多线程场景下不安全
```

3.继承：继承其他类的类称为派生类，被其他类继承的类称为这些类的基类

基类必须与派生类定义在一个作用域内

派生类定义的执行过程与基类类似

如果在类中找不到请求调用的属性会搜索基类

如果基类由别的类派生而来，则会递归式搜索

```python
class People:
  """
  人的类，定义人相关的一些基本信息如姓名，身高，年龄等。

  """
  def __init__(self,name,height,age):
    self.__name=name
    self.__height=height
    self.__age=age

  def get_name(self):
    return self.__name

  def get_height(self):
    return self.__height

  def get_age(self):
    return self.__age

  def print_info(self):
    print('in People')
    print('Name:{},Age:{},Height:{}'.format(self.get_name(),self.get_age(),self.get_height()))

  def __add__(self,other):
    return self.get_height()+other.get_height()

class Speaker():
  """
  演讲家类
  """
  def __init__(self,topic):
    self.__topic=topic

  def get_topic(self):
    return self.__topic

  def speak(self):
    print('in Speaker')
    print("speak topic is {}".format(self.get_topic()))

class Student(People,Speaker):
  """
  学生类，继承人的类，同时添加一些新的属性，并覆盖方法

  """
  def __init__(self,name,height,age,topic,ID,major):
    People.__init__(self,name,height,age)
    Speaker.__init__(self,topic)
    self.__ID=ID
    self.__major=major

  def get_ID(self):
```

```
    return self.__ID

  def get_major(self):
    return self.__major

  def print_info(self):
    print('ID:{}, Name:{}, Major:{}, Age:{}, Height:{}'.format(self.get_ID(),self.get_name(),self.get_major

  def speak(self):
    #super(Student,self).print_info()
    #super(Student,self).speak()
    super().print_info()
    super().speak()

p1=People('z',175,40)
s1=Student('zjc',175,35,'python',33060828,'cs')
print(p1+s1)
s1.speak()
People.print_info(s1)
#输出结果为：
350
in People
Name:zjc,Age:35,Height:175
in Speaker
speak topic is python
in People
Name:zjc,Age:35,Height:175
#说明：第一个350，是在People()中定义了__add__,把两个实例的身高相加
#之后的s1.speak()用super()先后调用了父类People的print_info()方法和父类Speaker的speark()方法
```

Python

4.继承的检查

```
class wh():
    def __init__(self,a):
        self.a=a
        pass

class whh(wh):
    def __init__(self,b):
        self.b=b

c1=wh(2)
c2=whh(3)
print(isinstance(c2,wh)) #检查一个实例c2的类是不是继承了wh
print(issubclass(whh,wh)) #检查类whh是不是继承了wh
#输出结果均为True
```

Python

5.多继承：子类继承多个父类

```
class BaseClass:
  num_base_calls=0
  def call_me(self):
    print("calling method on Base Class")
    BaseClass.num_base_calls+=1

class LeftSubclass(BaseClass):
  num_left_calls=0
  def call_me(self):
    BaseClass.call_me(self)
    #super().call_me()
    print("calling method on Left Subclass")
    LeftSubclass.num_left_calls+=1

class RightSubclass(BaseClass):
  num_right_calls=0
  def call_me(self):
    BaseClass.call_me(self)
    #super().call_me()
    print("calling method on Right Subclass")
    RightSubclass.num_right_calls+=1

class Subclass(LeftSubclass,RightSubclass):
  num_sub_calls=0
  def call_me(self):
    LeftSubclass.call_me(self)
    RightSubclass.call_me(self)
    #super().call_me()
    print("print calling method on Subclass")
    Subclass.num_sub_calls+=1
s=Subclass()
s.call_me()
#print(s)

print("\tsub_call:{}\n\
```

```
  left_call:{}\n\
  right_call:{}\n\
  base_call:{}".format(Subclass.num_sub_calls,LeftSubclass.num_left_calls,RightSubclass.num_right_calls,Bas
 #此时输出结果为：
calling method on Base Class
calling method on Left Subclass
calling method on Base Class
calling method on Right Subclass
print calling method on Subclass
        sub_call:1
        left_call:1
        right_call:1
        base_call:2
#Subclass()的call_me()方法分别调用了父类LeftSubclass.call_me()和RightSubclass.call_me()
#每个父类的call_me()又调用他们各自的父类Baseclass()的call_me,所以基类Baseclass()的call_me()被调用两次
#当改为用super时，可以保证每个基类只调用一次（两个父类都有call_me(),按定义类时从右往左调用）
#此时输出结果为：
calling method on Base Class
calling method on Right Subclass
calling method on Left Subclass
print calling method on Subclass
        sub_call:1
        left_call:1
        right_call:1
        base_call:1
```

Python

6.运算符重载：对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型

运算符重载不能改变其本来寓意
运算符重载只是一种“语法上的方便”

```
class Point:

  def __init__(self,x,y):
    self.x=x
    self.y=y

  def __add__(self,other):
    return Point(self.x+other.x,self.y+other.y)

  def __sub__(self,other):
    return Point(self.x-other.x,self.y-other.y)

  def __str__(self):
    return "({},{})".format(self.x,self.y)

  def __lt__(self,other):
    return self.x<other.x

  def __gt__(self,other):
    return self.x>other.x

  def __le__(self,other):
    return self.x<=other.x

  def __ge__(self,other):
    return self.x>=other.x

  def __eq__(self,other):
    return self.x==other.x

  def __ne__(self,other):
    return self.x!=other.x

  def __call__(self):
    '''
    Point类的实例可调用，也称可调用对象
    '''
    print('我不是函数，别调了')

p1=Point(1,2)
print(p1)    #这里是重载了__str__，使得实例可以打印
p2=Point(3,4)
print(p1+p2)    #这里重载了__add__,使得两个实例可以相加
p3=p2-p1    #这里重载了__sub__,使得两个实例可以相减
print(p3)
print(p1<p2)  #这里重载了__lt__
plist=[p1,p2,p3]
plist=sorted(plist,reverse=True)  #重载了所有比较运算符，可以对实例对象进行比较
print('\n'.join([str(p) for p in plist]))
p1()
p2()
p3()
#输出结果为：
(1,2)
(4,6)
```

```
(2,2)
True
(3,4)
(2,2)
(1,2)
我不是函数，别调了
我不是函数，别调了
我不是函数，别调了
```

Python

第七章 异常处理

本章工厂模式的内容见设计模式汇总

1.异常与处理

输入数据或设备状态不会一直理想并正确
应避免程序的错误或外部环境的影响给用户造成不佳的使用检验
向用户通告可能的错误
保存所有的已有结果
允许用户以妥善的形式退出程序

2.异常捕获：try……except

执行过程
执行 try 子句(在 try 和 except 关键字之间的部分)
— 无异常发生时 except 子句被忽略
— 如 try 子句在执行过程中发生异常，则该子句其余部分会被忽略
— 如异常匹配 except 指定的异常类型，则执行对应的 except 子句
— 如发生异常在 except 子句中没有与之匹配的分支，则传递到上一级 try 语句
— 如最终仍找不到对应的处理语句，则作为未处理异常，终止程序运行并显示提示信息
— 包含多个 except 子句时最多只有一个分支会被执行
— 处理程序应只针对对应的 try 子句中的异常进行处理，而不是其他的 try 的处理程序中的异常
— 一个 except 子句也可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组

— 最后一个 except 子句可以省略异常名称以作为通配符使用
慎用，会隐藏一个实际的程序错误信息
— 可以使用这种方法打印一条错误信息,然后重新抛出异常（允许调用者处理这个异常）

```python
#一个最简单的异常处理的例子
try:
    a=2
    a+='3'
except TypeError as err:
    print('TypeError:',err)
#输出结果为：
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
#注意这里 except...as... 后的 err 内容为报错的实际内容

#最后一个 except 子句可以省略异常名称以作为通配符使用
import sys
try:
    a=2
    a+='3'
except OSError as err:
    print('OSError:',err)
except :
    print("Unexpected error:", sys.exc_info()[0])
    print('Errortext:',sys.exc_info()[1])
    raise
#输出结果为：
Unexpected error: <class 'TypeError'>
Errortext: unsupported operand type(s) for +=: 'int' and 'str'
#以下是 raise 出来的异常
Traceback (most recent call last):
  File "c:\Users\10653\Desktop\大三上\现代程序设计\考试复习.py", line 451, in <module>
    a+='3'
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
#说明：sys.exc_info()返回一个元组，元组的三个元素分别为：
#1.exc_type 是异常的对象类型
#2.exc_value 是异常的值
#3.exc_tb 是一个 traceback 对象，其中包含出错的行数、位置等
```

Python

try … except 语句可以带有一个 else 子句，该子句只能出现在所有 except 子句之后
— 当 try 子句没有抛出异常时，需要执行一些代码，可以使用 else 子句
— 使用 else 子句比在 try 子句中附加代码要好，因为这样可以避免 try … except 意外地截获本来不属于它们保护的那些代码抛出的异常

异常处理并不仅仅处理那些直接发生在 try 子句中的异常
还能处理子句中调用的函数（甚至间接调用的函数）里抛出的异常

3.异常抛出:raise

raise 语句允许程序员强制抛出一个指定的异常
要抛出的异常必须是一个异常实例或异常类（继承自 Exception 的类）

```
def test_return():
  try:
    print('in try')
    raise Exception
    return 'return from try'
  except Exception:
    print('in except')
    return 'return from except'
  else:
    print('in else')
    return 'return from else'
  finally:
    print('in finally')
    return 'return from finally'

  return 'return from the last line'

print(test_return())
#输出结果为：
in try
in except
in finally
return from finally
'''说明：首先在 try 中输出 in try，然后 try 抛出了一个异常被 except 捕获，输出 in except
然后因为 try 有异常，else 分支不执行，且无论 try 子句有无发生异常，finally 子句都会执行'''
'''关于返回值的说明：不要在 try, except, else 子句里写返回值，尤其是有 else 子句时，try 中的 return 会导致 else 不执行
如果 finally 有返回值，其将会覆盖原始的返回值'''
```

Python

4.自定义异常：直接或间接的从 Exception 类派生

异常类的命名多以“Error”结尾
– 通常为了保持简单，异常类只包含属性信息，以供异常处理
– 如果一个新建的模块中需要抛出几种不同的错误，通常的作法是为该模块定义一个异常基类，然后针对不同的错误类型派生出对应的异常子类
– 发生异常时一般应通过参数提供更细致、具体的异常信息，可作为异常类的属性存在
如下例中，AccountNegativeDepositError 和 AccountBalanceNotEnoughError 类中有 user，m，balance 属性，使异常的信息更具体

```
class AccoutError(Exception):
  def __init__(self,user):
    self.user=user

class AccountNegativeDepositError(AccoutError):
  def __init__(self,user,m):
    self.user=user
    self.m=m
    self.message="{} deposit negative amount: {}".format(user,m)

class AccountBalanceNotEnoughError(AccoutError):

  def __init__(self,user,balance, m):
    self.user=user
    self.m=m
    self.balance=balance
    self.message="{}'s balance {} is smaller than the withdraw amount of {}. Loan is suggested.".format(use

class Account:

  def __init__(self,user,balance):
    self._user=user
    self._balance=balance

  def set_balance(self,balance):
    self._balance=balance

  def get_balance(self):
    return self._balance

  def get_user(self):
    return self._user

  def deposit(self,m):
    if m<0:
```

```
      raise AccountNegativeDepositError(self.get_user(),m)
    else:
      self.set_balance(self.get_balance()+m)

  def withdraw(self,m):
    if self.get_balance()<m:
      raise AccountBalanceNotEnoughError(self.get_user(),self.get_balance(),m)
    else:
      self.set_balance(self.get_balance()-m)

account=Account('zjc',100)
try:
  #account.deposit(-100)
  account.withdraw(10000)
except AccountNegativeDepositError as ande:
  print(ande.message)
except AccountBalanceNotEnoughError as abnee:
  print(abnee.message)
except AccoutError:
  print('noname exception')
else:
  print('no except...')
  print(account.get_balance())
#输出结果为：
zjc's balance 100 is smaller than the withdraw amount of 10000. Loan is suggested.
'''说明：首先定义了AccoutError类继承Exception，然后AccountNegativeDepositError和AccountBalanceNotEnoughError继承了AccoutError，分别对存款金额为负数和取的钱大于存款定义了异常'''
#注意：定义的Error中必须有message才可以被raise出来
```

Python

在异常名（列表）之后可以为 except 子句指定一个变量
• 该变量将绑定于一个异常实例
• 附属的参数存储在 args 属性中
– 异常实例一般会定义__str__()
• 可以直接访问并打印参数而不必引用参数
• 并不鼓励仅这么做，更好的做法是给异常传递参数
• 如果要传递多个参数，可以传递一个元组
• 一旦异常发生，能够在抛出前绑定所有指定的属性
– 比如构建一个 message 属性，包含异常的详细信息

```
try:
    raise Exception('arg1','arg2')
except Exception as inst:
    print(type(inst))
    print(inst.args)
    print(inst)
#输出结果为：
<class 'Exception'>
('arg1', 'arg2')
('arg1', 'arg2')
#说明：上述代码在raise时传给了异常两个参数，后续对except语句指定了一个变量
```

Python

5.预定义的清理行为

```
class DBError(Exception):

  def __init__(self,message):
    self.message=message

class DBhelper():

  def __init__(self,server_address,server_port,user,pwd):
    self._server_address=server_address
    self._server_port=server_port
    self._user=user
    self._pwd=pwd
    self._conn=None

  def get_server_address(self):
    return self._server_address

  def get_server_port(self):
    return self._server_port

  def __enter__(self):
    print("in __enter__: will try to connect the db server")
    return self

  def connect(self):
    try:
      print("in connect: connecting to the server")
      self._conn='success'#simulate the connection
      raise DBError('connect refused by the server')
```

```
    except DBError as dbe:
      print("in except: dbe.message")

  def close(self):
    print('in close: the connection is closed')
    #self._conn.close()
    pass

  def __exit__(self,type,value,trace):
    print("in __exit__: will close all the resource occupied")
    self.close()

with  DBhelper('127.0.0.1',3306,'zjc','123') as db:
  db.connect()

#输出结果为：
in __enter__: will try to connect the db server
in connect: connecting to the server
in except: dbe.message
in __exit__: will close all the resource occupied
in close: the connection is closed
#说明：类中定义了__enter__，会在进入是调用
#类中定义了__exit__，会在退出时调用
#注意：必须是with...as 才能生效，__exit__定义必须有四个参数
```

Python

6.异常处理的原则

(1)异常处理不能代替简单的测试
- 耗时，成本高
- 只在异常情况下使用异常处理

(2)不要过分地细化异常
- 代码量膨胀

(3)利用异常层次结构
- 自定义类

(4)不要压制异常(except 里什么也不做)
(5)早抛出、晚捕获

7.断言

用于判断一个表达式，并在表达式 False 的时候触发异常

```
assert 1==2
#等价于
if not 1==2:
    raise AssertionError
```

Python

第八章 装饰器

有关代理模式的内容见设计模式汇总

1.高阶函数：接收另一个函数作为参数的函数

```
#高阶函数的例子
def compose(x,y,f):
  return f(x)+f(y)
f=abs
print(compose(1,-2,f))
#输出结果为：
3

#reduce 函数用法
def f(x,y):
  return x*10+y
print(reduce(f,[1,2,3,4]))
#输出结果为：
1234
#说明：用传给 reduce 中的函数 function（有两个参数）先对集合中的第 1、2 个元素进行操作
#得到的结果再与第三个数据用 function 函数运算，直到集合的最后一个元素
#上式等价于(((1*10+2)*10+3)*10)+4

#map 与 reduce 函数结合
def cton(s):
  digits = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}
  return digits[s]
print(reduce(f,map(cton,'1234')))
#输出结果为
1234
#说明：首先由 map 函数将'1234'拆成 1，2，3，4 分别传入 cton 函数，然后再用 reduce 函数通过 f 函数计算

#filter 函数：过滤掉不符合条件的元素，返回由符合条件元素组成的新列表。
```

```python
#接收两个参数，第一个为函数，第二个为序列，序列的每个元素作为参数传递给函数进行判断
#然后返回 True 或 False，最后将返回 True 的元素放到新列表中
def odd(d):
  if d%2==0:
    return False
  else:
    return True
print(list(filter(odd,list(range(10)))))
print([x for x in range(10) if odd(x)])
#输出结果为：
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]

#返回值是函数：
def lazy_sum(*args):
    def sum():
        ax=0
        for n in args:
            ax=ax+n
        return ax
    return sum
f=lazy_sum(1,3,5,7,9)
print(f())
#输出结果为
25
#外函数返回内函数，内函数返回的是 ax，相当于外函数返回 ax

#偏函数：为了简化多参数函数的调用，可通过固定某参数来返回新函数，以实现简化调用
def growth(step=1,limit=200):
    g=0
    while(True):
        if g+step>limit:
            break
        g+=step
    return g
print(growth())
print(growth(step=3))
growth3=functools.partial(growth,step=3)
print(growth3)
print(growth3())
print(growth3(limit=300))
#输出结果为：
200
198
functools.partial(<function growth at 0x000001B6E37F9048>, step=3)
198
300
#说明：functools.partial：第一个参数为一个函数，紧接着后面可以固定这个函数的一些参数，将固定之后的结果当成一个新函数返回
```

Python

2.闭包：

```python
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i
        fs.append(f)
    return fs
f1, f2, f3 = count()
print(f1())
print(f2())
print(f3())
#输出结果为：
3
3
3

def count():
    def f(j):
        def g():
            return j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()
    return fs
f1,f2,f3=count()
print(f1())
print(f2())
print(f3())
#输出结果为：
1
2
3
```

```
#见https://www.cnblogs.com/yssjun/p/9887239.html（语言难以描述了属于是）
```

Python

3.装饰器：在不修改原始代码的前提一下增强或扩展既有功能

```
import time
def log(func):
  def wrapper(*args,**kwargs):
    print("call "+func.__name__)#额外功能
    return func(*args,**kwargs)#原有功能
  return wrapper

@log
def now():
  print(time.strftime('%Y-%m-%d',time.localtime(time.time())))
now() #相当于log(now)()
#输出结果为
call now
2022-01-08
```

Python

如果装饰器本身需要参数，则需要通过高阶函数实现

```
import time
def log(text):
    def decorator(func):
        def wrapper(*args,**kwargs):
            print(text+' '+'call '+func.__name__)
            return func(*args,**kwargs)
        return wrapper
    return decorator

@log('日志:')
def now():
    print(time.strftime('%Y-%m-%d',time.localtime(time.time())))
now()
print(now.__name__)
#输出结果为：
日志: call now
2022-01-08
wrapper
#函数now的名字变成了wrapper
#如何保持函数名称不发生变化：
from functools import wraps
def log(text):
    def decorator(func):
        @wraps(func)
        def wrapper(*args,**kwargs):
            print(text+' '+'call '+func.__name__)
            return func(*args,**kwargs)
        return wrapper
    return decorator

@log('日志:')
def now():
    print(time.strftime('%Y-%m-%d',time.localtime(time.time())))
now()
print(now.__name__)
#输出结果为：
日志: call now
2022-01-08
now
```

Python

4.装饰器类：用类方法实现装饰器

```
from functools import wraps
class Log:
    def __init__(self,logfile='out.log'):
        self.logfile=logfile
    def __call__(self,func): #重写__call__
        @wraps(func)
        def wrapper(*args,**kwargs):
            info="INFO: "+func.__name__+" was called"
            print(info)
            return func(*args,**kwargs)
        return wrapper
@Log('test.log')#@Log()
def myfunc():
    pass
myfunc()#相当于Log(‘test.log’)(myfunc)()
#输出结果为：
INFO: myfunc was called
#用类实现装饰器，装饰器需要的参数用__init__定义，重写__call__方法，使类的实例变成可调用对象，来实现装饰器
```

Python

5.装饰器的顺序：从下往上装饰，从上往下调用

```python
@a
@b
@c
def f ():
  pass
f() #相当于 f=a(b(c(f)))
#例子
def first(func):
    def wrapper(*args,**kwargs):
        print('这是在上面的装饰器')
        return func(*args,**kwargs)
    return wrapper

def second(func):
    def wrapper(*args,**kwargs):
        print('这是在下面的装饰器')
        return func(*args,**kwargs)
    return wrapper

@first
@second
def wh():
    pass
wh()
#输出结果为：
这是在上面的装饰器
这是在下面的装饰器
```

Python

Property 装饰器

使实例方法的使用如同实例属性
@property 读取属性
@ 方法名.setter 修改属性
@ 方法名.deleter 删除属性

```python
class Student:
  def __init__(self,id):
    self._id=id
  @property
  def id(self):
    return self._id
  @id.setter
  def id(self,id_value):
    if(id_value<0):
      pass
    else:
      self._id=id_value
  @id.deleter
  def id(self):
    del self._id
s=Student(12)
print(s.id)
s.id=18
print(s.id)
#输出结果为：
12
18
#说明：通过如上的写法，可以把 s.id 这个实例方法（也就是一个函数）变成好像一个实例对象来进行赋值之类的操作
```

Python

6.类方法与静态方法

实例方法需要通过 self 参数隐式的传递当前类对象的实例
用 @classmethod 修饰的方法需要通过 cls 参数传递当前类对象，称为类方法
用 @staticmethod 修饰的方法定义与普通函数一样，称为静态方法
等价于类外的普通函数，但可能仅为该类服务，因此搁进类中，往往写在类外可能更合适
类方法和静态方法均可通过类对象或实例对象调用

```python
class Finder:
  def __init__(self,path):
    self._path=path
  def get_path(self):
    return self._path
  @staticmethod
  def list_files(dir):
    print("{} contains files as listed:".format(dir))
    pass
  @classmethod
  def generate_finder(cls,root):
```

```
    print("class info: {}".format(cls))
    return cls(root)
 #输出结果为：
<class 'function'>
<class 'function'>
<class 'method'>
<class 'function'>
<class 'method'>
<class 'method'>
test contains files as listed:
class info: <class '__main__.Finder'>
test
finder_2 contains files as listed:
#说明：1-3行和4-6行一样，意思是用@staticmethod装饰过的类内方法会变成function，并且他们都支持类对象调用和实例对象调用
#用@staticmethod装饰过的类内方法就是普通函数，往往放在外面比较好。
```

Python

第十章：抽象类

1.抽象类的type()函数创建

type()函数：并非仅仅返回对象的类型
Python使用type()函数创建类对象
函数和类不是编译时定义的,而是在运行时动态创建的

```
#- type()函数依次传入3个参数：类名称、继承的父类集合(tuple)属性、（数据或方法）字典
def say_hello(self):
  print(f"hi,I am {self.name}")

def __init__(self,name):
  self.name=name

People=type('People',(object,),dict(say_hello=say_hello,__init__=__init__))
p=People('zjc')
p.say_hello()
#输出结果为：
hi,I am zjc
#注意，上述type在传参时，第二个是(object,) 这说明只有一个继承的父类时也要用元组形式传参
#第三个参数是dict(say_hello=say_hello,__init__=__init__)，一定要传字典！！！
```

Python

2.元类：metaclass

先定义metaclass，然后创建类，最后创建实例
类可以看成metaclass创建的“实例
通过关键字参数metaclass来指定元类来创建类

```
def add(self,value):
  self.append(value)

class ListMetaClass(type):
  def __new__(cls,name,bases,attrs):#将ListMetaClass 的__new__中增加一个叫add的函数
    attrs['add']=add
    return type.__new__(cls,name,bases,attrs)

class Mylist(list,metaclass=ListMetaClass):
  pass

l=Mylist()
l.add(1)
l.add(2)
print(l)
#这是在list中增加一个为add函数，称为Mylist，Mylist和list相比多了一个add方法，效果和append一样
```

Python

3.抽象类：

特殊的类，只能被继承，不能被实例化
从不同的类中抽取相同的属性和行为
抽象类中有抽象方法
不能被实例化，只能被继承
子类必须实现抽象方法

```
#抽象类的实现：借助abc模块
import abc
class Fruit(metaclass=abc.ABCMeta):#class Fruit(abc.ABC)
  @abc.abstractmethod
  def harvest(self):
    pass

  @abc.abstractmethod
```

```
  def grow(self):
    pass

class Apple(Fruit):

  def harvest(self):
    print("从树上摘")

  def grow(self):
    print("种苹果树")

  def juice(self):
    print("做苹果汁")

class Watermelon(Fruit):
  def harvest(self):
    #super().harvest()
    print("从地里摘")

  def grow(self):
    print("用种子种")

@Fruit.register   #注册抽象类
class Orange:
  def harvest(self):
    print("从树上摘")

  def grow(self):
    print("种橘子树")
```

```
a=Apple()
a.grow()
w=Watermelon()
w.harvest()

o=Orange()
o.grow()

print(isinstance(o,Fruit))
print(issubclass(Orange,Fruit))
print([sc.__name__ for sc in Fruit.__subclasses__()]) #no orange
#输出结果为：
种苹果树
从地里摘
种橘子树
True
True
['Apple', 'Watermelon']
#说明：从输出的最后一行来看，注册抽象类不是抽象类Fruit的子类
```

Python

4.接口

(1)Python 中没有专门的支持，但可以约定任何方法都只是一种规范，具体的功能需要子类实现
(2)与抽象类的区别
- 抽象类中可以对一些抽象方法做出基础实现
- 接口中所有方法只是规范，不做任何实现

(3)使用原则
- 继承抽象类应该尽量避免多继承
- 继承（实现）接口鼓励多继承

5.泛函数

```
from functools import singledispatch
class Student:
  pass
@singledispatch
def log(obj):
  print(obj)

@log.register(str)
def _(text):
  print(text.title())

@log.register(int)
def _(num):
  ns=[]
  while num:
    ns.append(num%10)
    num//=10
  for n in ns:
    print(n,end='')
  print()

@log.register(dict)
def _(d):
```

```python
  print("{} kvs load...".format(len(d)))

@log.register(tuple)
@log.register(list)
def _(l):
  print('不会打印列表或者元组...')

@log.register(Student)
def _(s):
  print("额，学生类也打不了。。。")

log('china')
log(1234)
log([1,2,3,4])
log({'a':1,'b':2,'c':3})
log((1,))
log(Student())
#输出结果为：
China
4321
不会打印列表或者元组...
3 kvs load...
不会打印列表或者元组...
额，学生类也打不了。。。
```

Python

6.适配器模式见设计模式总结