标识符、常量和变量

all codes test under Python 3.7, 3.8, 3.9

标识符

规定

1. 只能由字母、数字、下划线组成
2. 字母大小写敏感
3. 必须以字母或下划线开头
4. 不能和关键字同名

- 查看关键字

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

单独下划线表示上次的运算结果

```
>>> 10
>>> _*20
200
```

常量

浮点数精度

查看浮点数的表达精度

```
import sys
sys.float_info.dig
```

查看浮点数间的最小间隔

```
sys.float_info.epsilon
```

最大浮点数

```
sys.float_info.max
```

运算符

算术运算符

运算符	简介
/	除法
//	整除
**	幂

位运算符

二进制的相关运算

位运算符	简介	优先级	示例	备注
~	按位反	1	~x	补码
&	按位与	3	x&y	都是1取1，此外取0
|	按位或	4	x|y	都是0取0，此外取1
^	按位异或	4	x^y	相同取0，不同取1
<<	左移位	2	x<<2	左移1位，等于十进制下×2
>>	右移位	2	x>>2	右移1位，等于十进制下÷2

身份运算符

- 作用：判断两个变量是否引用同一个对象

运算符	简介
is	同一对象返回True，否则False

运算符	简介
is not	

输入输出

输出的若干方法

简单print

```
print(\[obj,...]\[,sep='  ']\[,end='\n'[,file=sys.stdout])
```

1. sep：当输出多个对象时，作为分隔符使用。默认是一个空格

```
print(123,456,789,sep='#')
```

123#456#789#

2. end：全部对象输出结束后的结束符，默认是换行符
3. file：输出的位置，默认是命令提示框，也可以指定为特定文件地址

```
filel= open('data.txt','w')
print(123,'abc',file=filel)
filel.close()
```

format格式化输出

位置匹配

1. 不带编号，即“{}”

```
print('{} {}'.format('hello','world'))  # 不带字段
hello world
```

2. 带数字编号，可调换顺序，即“{1}”、“{2}”

```
>>> print('{0} {1}'.format('hello','world'))  # 带数字编号
hello world
>>> print('{0} {1} {0}'.format('hello','world'))  # 打乱顺序
hello world hello
>>> print('{1} {1} {0}'.format('hello','world'))
world world hello
```

3. 带关键字，即“{a}”、“{tom}”

```
>>> print('{a} {tom} {a}'.format(tom='hello',a='world'))  # 带关键字
10 world hello world
```

格式化形式基本类似于标准化输出，只是先导%变成了:

```
print('{tom:.2f}'.format(tom=3.994,a='world'))
```

数字进制转换

- 支持二进制
- 仅支持对数字进行操作，十进制仅支持整数

```
>>> print('{0:b}'.format(3))     #二进制
11
>>> print('{:d}'.format(20))     #十进制整数
20
>>> print('{:o}'.format(20))     #八进制
24
>>> print('{:x}'.format(20))     #十六进制
14
```

- :后面先加#则带进制前缀

```
>>> print('{0:#b}'.format(42))  #二进制
0b101010
>>> print('{:#d}'.format(42))   #十进制整数
42
>>> print('{:#o}'.format(42))   #八进制
0o52
>>> print('{:#x}'.format(42))   #十六进制（小写）
0x2a
>>> print('{:#x}'.format(42))   #十六进制（大写）
0X2A
```

数字格式转换

- 仅支持对数字进行操作

'c' - 字符。在打印之前将整数转换成对应的Unicode字符串。

'e' - 幂符号。用科学计数法打印数字。用'e'表示幂。

'g' - 一般格式。将数值以fixed-point格式输出。当数值特别大的时候，用幂形式打印。

'n' - 数字。当值为整数时和'd'相同，值为浮点数时和'g'相同。不同的是它会根据区域设置插入数字分隔符。

'%' - 百分数。将数值乘以100然后以fixed-point('f')格式打印，值后面会有一个百分号

```
>>> print('{0:b}'.format(3))
11
>>> print('{:c}'.format(20))
•
>>> print('{:e}'.format(20))
2.000000e+01
>>> print('{:g}'.format(20.1))
20.1
>>> print('{:f}'.format(20))
20.000000
>>> print('{:n}'.format(20))
20
>>> print('{:%}'.format(20))    #默认保留6位
2000.000000%
```

```
>>> print('{:.2%}'.format(20))  #仅保留两位小鼠
2000.000000%
```

左右对齐、保留位数和填充

左对齐	中间对齐	右对齐	（仅数字的右对齐）(不建议使用)
<(默认)	^	>	=(等价于>)

- 字符串的左右中对齐

```
>>> print('{} and {}'.format('hello','world'))  # 默认左对齐
hello and world
>>> print('{:10s} and {:>10s}'.format('hello','world'))  # 取10位左对齐，取10位右对齐
hello      and      world
>>> print('{:^10s} and {:^10s}'.format('hello','world'))  # 取10位中间对齐
  hello    and   world


```

- 保留小数位数：仅对数字类型操作

```
>>> print('{} is {:.2f}'.format(1.123,1.123))  # 取2位小数
1.123 is 1.12
>>> print('{0} is {0:>10.2f}'.format(1.123))  # 取2位小数，右对齐，取10位
1.123 is       1.12
```

- 填充

```
>>> '{:^30}'.format('centered')  # 中间对齐
'           centered           '
>>> '{:*^30}'.format('centered')  # 使用“*”填充
'***********centered***********'
>>>'{:0=30}'.format(11)  # 还有“=”只能应用于数字，这种方法可用“>”代替
'000000000000000000000000000011'
>>>'{:0>30}'.format(11)  # 与上面结果相同
'000000000000000000000000000011'
```

千位分割号

代码	介绍	例子
:,	加,做千分符	13,800,000
:_	加_做千分符	13_800_000

字符串前置f

f"xxxx"可在字符串前加f以达到格式化的目的，在{}里加入对象，此为format的另一种形式：

```

>>> a = "hello"
>>> b = "world"
>>> f"{a} {b}"
'hello world'
```

print+ 标准格式化输出（建议一般不要用这个方式）

```
print(控制字符串%(输出项1，输出项2，...,输出项n))
string = 控制字符串%(输出项1，输出项2，...,输出项n)
print(string)
```

- 示例

```
>>> print('pi的大小是 %5.3f.' % 3.1415926)
pi的大小是 3.142.
```

类似参照c语言的输出方法设定

格式符	简介	格式
d	带符号的十进制形式输出（正数不显示+）	
o	八进制无符号整数形式（不输出前导0）	%#o
x/X	十六进制无符号整数（不输出前导0x）	%#x
c	字符形式输出一个字符	
s	输出字符串	
f	小数形式输出，**默认6位**	%+-m.nf
e/E	标准指数形式，数字部分1位整数，6位小数	%+-m.ne
g/G	根据给定的值和精度，系统自动选择f或e中较紧凑的格式输出	%+-m.ng

没有二进制，二进制要用format或bin(num), bin(num) 带b前缀且取消不掉

确定具体格式的附加字符

附加	简介
m	输出数字的全部位数，不够位在左侧添加空格（右对齐）
n	规定数字的小数位
-	改为左对齐
+	正数前面添加+号
#	使得八进制和十六进制输出前导0、0x

循环结构

---

while

```
while 条件表达式:
    语句块
[else:]
    语句块
```

else部分可选，表达当条件不满足离开循环时需要执行的命令（只执行1次）

for

```
for 循环变量 in 序列对象:
    循环体
[else:]
    语句块
```

##

序列数据

列表

列表的基本操作

1. 求表长

```
len(list1)
```

2. 列表更新：赋值

```
list1[1] = 4
```

3. 删除元素或部分列表

```
list1 = [1,2,3,4]
del list1[0]    #将列表list1第0个元素
del list1[x:y]  #删除列表[x,y-1]的元素
del list1[:]    #清空列表list1，返回空列表
```

4. 列表合并

```
>>> list1 = [1,2]
>>> list2 = [3,4]
>>> list3 = list1 + list2
[1, 2, 3, 4]
```

5. 列表乘法

```
>>>list1 = [1,2]
>>>lsit2 = list1 * 3
[1,2,1,2,1,2]
```

6. 列表分片

```
>>>list1 = [1,2,3,4]
>>>list1[1:3]
[2,3]
```

7. 成员判断

```
>>>list1 = [1,2,3,4]
>>> 1 in list1
True
```

列表的方法

```
>>>list1 = [1,2,[1,2],3,4,5,3]
```

1. 检索元素

```
list.index(value[,start[,end]]
```

- 检索value，
- 可选：start和end是起始和结束位置
- 返回元素首次出现的位置；若没有，则返回ValueError

```
>>>list1.index(3)
3
>>>list1.index(-1)
ValueError: -1 is not in list
```

2. 统计元素出现的次数

```
list.count(element)
```

3. 在表尾添加元素

```
list.append(element)
```

4. 在表中插入新元素

```
list.insert(index, element)
```

- 在索引index位置插入element元素，之后的元素顺位后移

5. 两个列表的合并

```
list1.extend(list2)
```

6. 移出并返回元素

```
list.pop([index]) #删除第index个元素并返回，index默认-1
```

- 默认移出列表中的最后一个元素

7. 删除元素或部分列表

```
list1 = [1,2,3,4]
del list1[0]    #将列表list1第0个元素
del list1[x:y]  #删除列表[x,y-1]的元素
del list1[:]    #清空列表list1，返回空列表
```

8. 移出第一个被匹配的元素

```
list.remove(element)
```

- 移出列表中element第一个匹配的对象，后续元素顺次前移

9. 在表中插入新元素

```
list.insert(index, element)
```

- 在索引index位置插入element元素，之后的元素顺位后移

10. 列表逆置

```
list.reverse()
```

11. 列表排序

```
list.sort(reverse=False)
```

- reverse默认为False，正序排序；设置为True时，逆序排序
- 当使用sorted函数时，不改变原列表，而是产生一个排序后的新列表

12. 列表清除

```
list.clear()
```

- 等价于del list[:]

元组

元组的创建与删除

1. 元组一经创建，不可修改！
2. 元组的创建：当只有一个元素时，必须以,结尾。（避免引起歧义）

```
a = (1,)
```

用列表创建元组：

```
a = ([1,2,3])
```

3. 元组的删除

```
del a
```

4. 读取：[]形式

5. 切片

```
tuple1[start:end]
```

6. 求长度

```
len(tuple1)
```

7. 加法和乘法构造新元组

```
a = (1,2)
b = (3,4)
c = a + b
d = a * 3
```

- 但不同序列数据，如列表和元组，不能直接相加

8. 成员判断

```
>>> a = (1,2)
>>>2 in a
True
```

元组的方法

1. 检索

```
tuple.index(value[,start[,end]])
```

- 起止为可选参量

2. 计数

```
tuple.count(element)
```

元组和列表比较

1. 元组是不可变序列，运算速度比列表快
2. 元组是不可变序列，元组可以作为字典的键，而列表不可以
3. 内置函数可以让元组和列表互换

```
tuple()
list()
```

字典

映射类型，由键值对构成

字典的基本操作

1. 创建字典

   赋值方式创建字典

```
dict1 = {}
dict1['name'] = 'Tom'
```

   内置函数dict()创建

```
>>>dict2 = dict([(1,'a'),(2,'b'),(3,'c')])
>>>dict2
{1:'a',2:'b',3:'c'}
```

```
>>>dict3 = dict(a=1,b=2,c=3)
>>>dict3
{'a':1,'b':2,'c':3}
```

   内置函数.fromkeys()创建值为None的空字典

```
>>>dict4 = {}.fromkeys(['name','age'])
>>>dict4
{'name':None,'age':None}
```

2. 添加和修改

```
dict4['key'] = value
```

- 当键'key‘存在时，则修改其值为value
- 当'key'不存在时，则创建新的键值对'key':value

3. 删除

删除成员

```
del dict4['key']
```

删除字典

```
del dict4
```

4. 字典遍历

通过调用字典的三种方法分别进行不同的遍历

a. 遍历键值对

```
for key,value in dict1.items():
```

b. 遍历键

```
for key in dict1.keys():
```

c. 遍历值

```
for value in dict1.values():
```

字典的方法

1. 获取指定键的对应值

```
dict1.get(key)
```

- 键不存在时返回None

```
dict1.setdefault(key,value)
```

- 存在返回值，不存在则将键值对加入字典

2. 清空字典

```
>>>dict1.clear()
>>>dict1
{}
```

3. 字典复制

三类：引用、浅复制、深复制

a. 引用：两个字典名字都指向同一个地址，改变任何一个字典，另外一个也会改变

```
dict2 = dict1
```

b. 浅复制：只将字典的键值对拷贝了一份，但如果值是列表、元组等，这些列表和元组等仍然指向相同地址

```
dict2 = dict1.copy()
```

示例：

```
>>>dict1={'name':'Tom','tele':12345,'sex':
['M','F']}
>>>dict2 = dict1.copy()
>>>dict1['sex'].remove('F')
>>>dict2
{'name':'Tom','tele':12345,'sex':['M']}
```

c. 深复制：两个字典实现完全的独立

```
import copy
dict2 = copy.deepcopy(dict1)
```

4. 字典更新

```
dict1.update(dict2)
```

- 将字典dict2的键值对更新到字典dict1中；
- 若dict1中没有该键，则添加

```
>>>dict1={'name':'Tom','tele':12345,'sex':'M'}
>>>dict2={'name':'TOm','tele':1234567,'age':45}
>>>dict1.update(dict2)
>>>dict1
{'name':'Tom','tele':1234567,'sex':'M','age':45}
```

集合

对应数学上的集合，具有特点：

- 其中元素只可添加删除，不可修改
- 元素无序
- 元素不重复

集合的创建

直接创建可变集合

```
set1 = {a,b,c,d}
```

- 其中的重复元素会被自动忽略
- 由于其中元素不可变，所以不可以将集合、列表、字典作为对象插入集合中；但元组可以

创建空集合：

```
set1 = set()
```

- dict1 = {} 会创建字典

内置函数创建可变集合

```
>>>set1 = set([1,2,3,1])
>>>set1
{1,2,3}
```

- 由于其中元素不可变，所以不可以将集合、列表、字典作为对象插入集合中，但元组可以

*创建不可变集合

```
q_set = frozenset([1,2,3,4])
```

- 其中元素不可添加删除
- 由于其中元素不可变，所以不可以将集合、列表、字典作为对象插入集合中

集合的基本操作

1. 元素访问

   集合中元素没有顺序，**无法通过索引方式访问**，只能遍历

   ```
   for s in set1:
       print(s)
   ```

2. 可变集合添加元素

   ```
   set1.add(element)
   ```

   - 可以加入元组
   - 但不可以加入可变对象：列表、字典、集合等

3. 可变集合的合并

   ```
   set1.update(set2)
   ```

4. 可变集合删除元素

   ```
   set1.remove(element)
                          #删除，若不存在则抛出异常KeyError
   set1.discard(element)#删除，但不存在没有提示
   set1.pop()             #删除第一个元素并返回
   set1.clear()           #删除全部元素
   ```

集合的数学运算

```
set1 | set2 #并集
set1 & set2 #交集
set1 - set2 #差集
```

String相关

1. String对象是常量，所以索引仅能获取其单个字符的值，但无法修改

   修改可采用如下方案：修改‘strang’为‘string’

   ```
   string = 'strang'
   string = string[:2] + 'i' + string[4:]
   ```

2. 串比较是针对相应字符ASCII码值大小，顺次比较每个字符位置，遇到非相等的字符即返回相应结果，否则继续比较下一位直到结束。

   注：空字符串小于任何字符串，其长度为0.

   ```
   >>> '' < 's'
   True
   ```

常用的字符串操作

常用函数

```
len(string)      #求长度
str(string)      #转换为字符串形式
chr(s)               #返回unicode编码对应的单字符
ord(s)               #返回单字符对应的unicode编码
```

常用操作

串连接

1. 多个字符串直接写在一起
2. 空格分隔的多个字符串被直接合并
3. 加号运算符的合并
4. 乘法运算创建重复字符串

串分片

```
串名[start:end:step]
```

应用：

1. 除去最后一个字符返回其他

```
s[:-1]
```

2. 串逆置

```
s[::-1]      #即设置步长为-1
```

串比较

- 标准：根据相应字符的ASCII码大小，依次比较两个字符串中同一位置的字符，相等则向前推进，直到在某一次比较分出大小或结束
- 特殊：空串比任何串都笑，其长度为0

常用内置方法

判断型，返回BOOL值

```
str.isalpha()
str.isdigit()
str.isupper()
str.islower()
```

返回新的字符串的方法

```
str.upper()
str.lower()
str.capitalize()    #首字母大写，其余全部小写
str.strip()             #删除字符串两端空格，返回新串
```

查找

```
str.find(key, [start, end])     #查找key首次出现的位置并返回
```

替换

```
str.replace(old,new, [num])  #替换字符串中old为new，次数num可选
```

切分

```
str.split('cut') #以cut为分隔符切分字符串
```

- 注意：若str是空字符串‘’，则其分割总会返回一个包含空字符串的数组！

  ```
  ['']
  ```

format 方法

默认顺序

```
>>>"{} {}".format('hello','world')
'hello world'
```

指定位置

```
>>>"{0} {1} {0}".format('hello','world')
'hello world hello'
```

正则表达式基本流程

```python
import re

key = "test" # 要匹配的文本
p = r"(?<=)" # 表达式
pattern = re.compile(p) # 编译正则表达式
matcher = re.search(pattern, key) #返回第一个成功的匹配
```

1.特殊字符正则表达式

特殊字符	含义
^	匹配字符串开始位置
$	匹配字符串结束位置
()	括号中视为一个子表达式
*	匹配前面子表达式0次或多次
+	匹配前面子表达式1次或多次
?	匹配前面子表达式0次或1次
.	匹配除换行符外的任何单个字符
[...]	匹配中括号中包含的任意一个字符 1. 其中连字符-描述一个匹配的范围，若-出现在首位则代表普通字符 2.**特殊字符仅有反斜杠\保持其特殊含义**，用于转义字符。其他特殊字符+，*等作为普通字符匹配 3.脱字**^出现在首位表示匹配不包含其中的任意字符**，如果出现在中间就作为普通字符匹配
\	将普通字符变为特殊字符 解除元字符的特殊功能 引用序号对应的子组所匹配的字符串？
{M,N}	匹配前面的子表达式M～N次
A|B	匹配正则表达式A或B

2. Re模块

```
import re
```

1. 正则表达式的编译

```
string = 'xxx'
pattern = re.compile(rstring)
```

- 返回表编译出的正则表达式供其他方法使用

替代方法：

```
pattern = r'xxx'
```

2. 起始位置匹配

```
re.match(pattern, string, flags=0)
```

- 从起始位置匹配pattern模式
- 返回：匹配成功返回(start,end)元组，起始位置没有匹配到则返回None

3. 首配搜索

```
re.search(pattern,string,flags=0)
```

- 在整个string中搜索
- 返回：搜索到返回一个re.Match Object；否则返回None

4. 找到全部

```
re.findall(pattern,string,flags=0)
```

- 找到全部匹配的字符串
- 返回：如果找到，则返回其组成的一个列表；没找到返回None

5. 替换

```
re.sub(pattern, repl string[,count])
re.subn(pattern, repl string[,count])
```

- 将string中pattern匹配的替换为repl；count表示最大替换次数，如果没有则全部替换
- 返回：re.sub返回替换后的字符，re.subn返回一个元组(替换后的字符, 替换的次数

6. 切分

```
re.split(pattern, string)
```

- 类似字符串的split()方法，但效率更高
- 返回一个切割后的字符串数组

3.常用正则表达式示例

实例	含义
[aeiou]	匹配一个小写元音字母
[^aeiou]	匹配一个除小写元音字母外的任何字符
[0-9]	匹配一个任何数字
[a-z]	匹配一个任何小写字母
[a-zA-Z0-9]	匹配一个大小写字母或数字
.	匹配一个除'\n'外的任何字符
\d	匹配一个数字字符
\D	匹配一个非数字字符
\A	匹配字符串的开始
\Z	匹配字符出的结束

- 其他参考Python相关教程

函数与模块

函数基础概念

函数的定义：

```
def 函数名([形式参数表]):
    ...
    [return 返回值]
```

- 在使用函数前，必须先定义或声明函数
- 定义格式说明：

    a. 如果没有return语句默认返回None
    b. 不需要指定返回值的类型
    c. 必须有圆括号和冒号

函数调用

```
函数名([实际参数表])
```

过程

1. 如果需要，为形参分配内存单元，后将实参的值或地址赋给形参
2. 为函数体内的变量分配内存单元，执行函数体内的语句
3. 若有return语句，则将返回值带回主调函数；若无return语句，则程序带回None返回；
4. 返回前会释放形参和函数内部变量所占的内存空间

函数参数

分类

1. 形式参数：定义函数时的参数
2. 实际参数：调用函数时的参数

实参向形参传递的方式

1. 值传递：函数调用时，会给形参分配与实参不同的内存单元，并将实参的值复制给形参；函数调用结束时，形参所占内存单元会被释放；
    - 数字
    - 字符串
    - 元组
2. 址传递：将实参的地址传递给形参。所以形参和实参占用同一段内存单元，在函数内形参的改变也意味着实参的改变。但该方式要求实参是可变对象：
    - 列表
    - 字典

函数的默认参数

若函数调用时没有传递实参，则按定义是的默认参数进行计算

函数的不定长参数

不定长参数会被包装进一个元组活字典，在可变参数前可以由若干个普通参数。

```
def function([普通参数][,*可变参数]):
    函数体
    [return 返回值]
```

上述定义中"*可变参数"部分会存放所有未命名的变量参数。调用时，会依次序将所有普通变量都复制后，剩下的参数将会收集在一个名叫"*可变参数"的元组中。

函数的返回值

返回多个值，可以用逗号分隔，默认为返回一个元组

变量作用域

- 全局变量：定义在所有函数外的变量，在程序执行时全程有效；
- 局部变量：定义在函数内部的变量，只能在内部使用；

性质：

1. 若函数内部有重名局部变量，优先局部变量；
2. 函数内部可以通过关键字申明接下来使用的是全局变量；

```
def function():
    global test
    test = 'c'
function()
print(test)
```

匿名函数

Python中特殊的函数声明方式，适用于简单的、能够在一行内表示的函数

```
lambda params:expr
```

- params：相当于声明函数时的参数列表
- expr：函数返回值的表达式，但表达式不能含有其他语句；

    a. 可以返回元组
    b. 允许调用其他函数

两数相加

```
lambda a,b:a+b
```

几个常用特殊函数

map

```
map(func, seq)
```

- 内置函数

- 将一个单参数函数一次作用到一个序列对象的每一个元素上，
- 返回一个map对象作为结果，其中每个元素是原序列中元素经过该函数处理后的结果，原序列没有改动

reduce

```
reduce(func,iterable[,initializer])
```

- 内置函数
- 将一个序列对象（列表、元组、字典、字符串等）数据进行如下操作：
    a. 用传给其的function函数对集合中1，2元素操作得到结果
    b. 再用结果与第3个元素操作
    c. 以此类推，最后得到一个结果
    d. [如果有initializer，则第一次为该值与集合中第1个元素作用]
- 返回最后得到的结果

应用：累加，累乘，按字典对象某个维度进行分组

filter

```
filter(func, iterable)
```

- 内置函数
- 单参数函数作用于序列上
- 返回值是使得函数返回值为True的元素组成的列表、元组或字符串

应用：过滤奇数或偶数

Python 内置库函数

数学类

函数名	功能描述
abs(a)	绝对值
round(a)	四舍五入法返回浮点数

函数名	功能描述
pow(a,b)	a的b次方
cmp(a,b)	比较大小
divmod(a,b)	返回整除结果和余数的数组
max([iterable])	最大值
min([iterable])	最小值
sum([iterable])	求和

类型和进制转换

函数名	功能描述
int()	转为整数integer
float()	转为浮点数float
long()	转为长整数long interger
str()	转为字符串
complex(3,9)	转为复数，返回3+9i
ord('c')	返回ASCII字符对应数值
chr(integer)	返回整数对应ASCII字符
unichr(integer)	返回整数独赢unicode字符
bin(integer)	十进制转二进制（带前缀）
hex/oct(integer)	十进制转十六/八进制（带前缀）

- 如果需要考虑转换为**大写形式**(对于16进制),或者**取消前缀**需要使用format方法：

```
>>>format(255, '#x'), format(255, 'x'), format(255, 'X')
#分别对应带前缀,无前缀小写，无前缀大写
('0xff', 'ff', 'FF')
```

- 如果要转换浮点数，使用方法（仅支持转换为16进制）

```
float.hex()
```

- 转换为2进制参考

```
```

```python
# Python program to convert float
# decimal to binary number

# Function returns octal representation
def float_bin(number, places=3):
    # split() seperates whole number and decimal
    # part and stores it in two seperate variables
    whole, dec = str(number).split(".")

    # Convert both whole number and decimal
    # part from string type to integer type
    whole = int(whole)
    dec = int(dec)

    # Convert the whole number part to it's
    # respective binary form and remove the
    # "0b" from it.
    res = bin(whole).lstrip("0b") + "."

    # Iterate the number of times, we want
    # the number of decimal places to be
    for x in range(places):
        # Multiply the decimal value by 2
        # and seperate the whole number part
        # and decimal part
        whole, dec = str((decimal_converter(dec)) *
2).split(".")

        # Convert the decimal part
        # to integer again
        dec = int(dec)

        # Keep adding the integer parts
        # receive to the result variable
        res += whole

    return res


# Function converts the value passed as
# parameter to it's decimal representation
def decimal_converter(num):
    while num > 1:
```

```
        num /= 10
    return num


# Driver Code

# Take the user input for
# the floating point number
n = input("Enter your floating point value : \n")

# Take user input for the number of
# decimal places user want result as
p = int(input("Enter the number of decimal places of the
result : \n"))

print(float_bin(n, places=p))
```

序列操作

函数名	功能描述
sorted([iterable])	返回一个正序序列，不改变原序列
reversed([iterable])	返回一个逆序序列，不改变原序列

eval: 动态执行Python语句

```
eval(expression[, globals[, locals]])
```

If you pass in an **expression**(string form) to `eval()`, then the function parses it, compiles it to bytecode, and evaluates it as a Python expression.

1. **Parse** `expression`
2. **Compile** it to bytecode
3. **Evaluate** it as a Python expression
4. **Return** the result of the evaluation

```
>>> eval("2 ** 8")
256
>>> eval("1024 + 1024")
2048
>>> eval("sum([8, 16, 32])")
56
>>> x = 100
>>> eval("x * 2")
200
```

- All the names passed to `globals` in a dictionary will be available to `eval()` at execution time.

  DIY globals 字典是被允许的
- the dictionary contains the variables that `eval()` uses as local names when evaluating `expression`.

eval的应用场景

1. 逻辑判断语句

```
>>> x = 100
>>> y = 100
>>> eval("x != y")
False
```

2. 数学表达式

```
>>> import math
>>> # Area of a circle
>>> eval("math.pi * pow(25, 2)")
1963.4954084936207
```

3. 执行函数、调用方法，输出文本等

```
>>> import subprocess
>>> eval("subprocess.getoutput('echo Hello, World')")
'Hello, World'
```

- 不支持：statement类型语句：如if，

其他参考文章：https://www.cnblogs.com/benjamin77/p/10048503.html

文件

文件概述

基本概念

1. 文件：一组存储在外部存储介质（磁盘，光盘，U盘等）上的信息集合，可以包含任何数据内容；
2. 操作系统是以文件的形式来管理外部存储介质上的文件，并以“**文件名.扩展名**”的形式来标识文件；
3. 从文件内容的表现形式分类
    - 文本文件：ASCII码，字符流；存在字符编码要与二进制转换，速度慢
    - 二进制文件：二进制码，字节流；读写效率高

Python文件操作流程

1. 建立/打开文件
2. 文件读写
3. 关闭文件

文件的打开与关闭

模式	功能描述
rt	只读方式打开文本文件，只允许读数据
wt	只写方式打开或建立文本文件，只允许写数据
at	追加打开一个文本文件，并在文件结尾添加数据
rb	只读方式打开二进制文件，只允许读数据

模式	功能描述
wb	只写方式打开二进制文件，只允许写数据
ab	追加打开一个二进制文件，并在文件结尾添加数据
rt+	读写方式打开一个文本文件，允许读和写
wt+	读写方式打开或建立一个文本文件，允许读和写
at+	读写方式打开一个文本文件，允许读或在文件结尾添加数据
rb+	读写方式打开一个二进制文件，允许读和写
wb+	读写方式打开或建立一个二进制文件，允许读和写
ab+	读写方式打开一个二进制文件，允许读或在文件结尾添加数据

打开文件

```
1  文件对象 = open(<文件名>,<打开模式>)
```

关闭文件

```
1  文件对象.close()
```

文件的基本操作

文件读

.read()

```
1  string = 文件对象.read([size])
```

- 从用读模式打开的文件中读取指定的字节数。
- 文本文件返回字符串，二进制文件返回字节流。
- size为负数或空时，读取到文件末尾

. readline()

```
string =  文件对象.readline([size])
```

- 从用读模式打开的文件中从当前位置读取到行末，多用于文本文件
- 若size默认大于本行字符长度，则读取到本行末尾（含‘\n’）

.readlines()

```
string = 文件对象.readlines([size])
```

- 从用读模式打开的文件中从当前位置读取多行数据，多用于文本文件
- size不填则读取到文件末尾；size表示读取的总长度，若小于本行字符长度，则读取到本行末尾
- 返回一个每行内容组成的列表，每行都包括‘\n’

文件写

.write()

```
文件对象.write(字符串)
```

- 向写模式打开的文件的当前位置写入字符串
    a. 文本文件则写入字符串
    b. 二进制文件则写入字节流
- 返回写入的字符数或字节数
- **不在字符串的结尾添加换行符‘\n’**

.writelines()

```
文件对象.writelines(列表)
```

- 向用写模式打开的文件当前位置写入列表中的所有元素，多用于文本文件
- 不会自动加入换行符，需要在每个元素末尾自己添加

文件定位

.seek()

```
文件对象.seek(offset,whence=0)
```

- 参数offset为相对于whence所指示位置的字节偏移量；
- whence表示所指示的位置，默认值为0

WHENCE	位置
0	相对于文件开始位置
1	相对于文件读写位置
2	相对于文件结尾

.tell()

```
文件对象.tell()
```

- 返回文件当前位置，相对于文件开始的位置

文件的相关模块

常用的有os和shutil两个模块

os.mkdir()

```
import os
os.mkdir(path)
```

- 按path指定的路径创建单级目录，若目录存在则抛出异常

os.makedirs()

```
import os
os.makedirs(path)
```

- 按path指定的路径递归地创建多级目录，若目录存在则抛出异常

os.listdir()

```
import os
os.listdir("D:\\")
```

- 列出path即D盘根目录下的所有文件和目录
- 返回列表

os.getcwd()

```
import os
print(os.chdir())
```

- 显示当前目录

os.chdir()

```
import os
os.chdir(path)
```

- 改变当前工作目录到path

os.rmdir()

```
import os
os.rmdir(path)
```

- 按path指定路径删除目录（若目录非空则抛出异常）

os.remove()

```
import os
os.remove(filename)
```

- 按filename删除指定文件，若文件不存在则抛出异常

os.rename()

```
import os
os.renmae(old,new)
```

- 将old路径+文件名1更改为new所对应的文件名（路径+文件名2）
- 路径不能更改

shutil.copyfile()

```
import shutil
shutil.copyfile(源文件，目标文件)
```

- 复制文件到目标文件位置（文件路径+文件名称）

Python计算生态

---

random库

time库

```
import time, datetime
```

用于获取系统时间的内置库

time库常用函数

语法	功能	示例
time.time()	获取**时间戳**(计算机内部时间值)，从1970年1月1日后经过的浮点秒数	15558934.2954187
time.ctime()	当前时间结果的字符串	Mon May 27 09:46:13 2019

语法	功能	示例
time.gmtime()	获取当前时间，返回time.struct_time对象	time.struct_time对象格式见下
time.strfttime(tpl,ts)	返回按照tpl模板形式输出ts表示的时间；ts为时间戳	
time.strptime(str,tpl)	将时间字符串str按照tpl模板变成时间戳	
time.perf_counter()	返回CPU级别精确时间计数值（秒）；常用于统计程序运行时间	
time.sleep(s)	让程序休眠s浮点秒	

模板字符串的格式化字符

串格式字符	说明	范围
%Y	年份	0000-9999
%m	数字月份	01-12
%B	全拼月份	January-December
%b	缩写月份	Jan-Dec
%d	日期	01-31
%A	全拼星期	Monday-Sunday
%a	缩写星期	Mon-Sun
%H	24小时制	00-23
%h	12小时制	01-12
%p	上/下午	AM,PM
%M	分钟	00-59
%S	秒	00-59

time.struct_time对象

```
time.struct_time(tm_year=1,tm_mon=1,tm_mday=1,tm_hour=0,
tm_min=0,tm_sec=0,tm_wday=1,tm_yday=2,tm_isdst=-1)
```

使用：

```
import time
t = time.gemtime() #获取当前时间
print(t.tm_year)
# 2020 输出当前年
```

datetime库

定义的特殊常量和类：

- MINYEAR = 1
- MAXYEAR = 9999
- date类：表示日期的类
- time类：表示时间的类
- datetime类：表示日期和时间的类
- timedelta类：表示两个datetime对象的差值
- tzinfo类：表示时区相关信息

```
#使用时应提前导入上述类
from datetime import date, time, datetime, timedelta, tzinfo
from datetime import *
```

date类

```
datetime.date(year, month, day)
```

- print返回格式：year-month-day

构造函数

构造函数	描述
datetime.date(year, month, day)	初始化构造方法，需要年月日三个参数
datetime.date.today()	用今天日期构造
datetime.date.fromtimestamp(t)	使用时间戳构造对象，可通过time.time()获取
datetime.date.fromordinal(n)	使用日期序数构造对象，从公元1年1月1日开始的序数

方法

方法	说明
timetuple()	返回time.time_struct对象格式的日期
toordinal()	fromordinal(n)的逆过程
weekday()	返回星期0-6
isoweekday()	返回标准星期1-7
isoformat()	返回标准格式YYYY-MM-DD
strftime(tpl)	自定义格式输出，参考日期模板
replace(year, month, day)	传入参数year,month,day对应的新日期

time类

```
datetime.time(hour, minute, second, microsecond, tzinfo, fold)
```

- print返回格式：hour:minute:second[.microsecond]

构造函数

构造函数	描述
datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None, fold=0)	初始化构造方法

方法

方法	说明
isoformat()	返回标准时间格式HH:MM:SS.mmmmmm+zz:zz
strftime(tpl)	自定义格式输出，参考日期模板
utcoffset()	返回timezone较utc的偏移量
tzname()	timezone名称
replace()	通过传入相应参数修改时间

datetime类

```
datetime.datetime(year, month, day[,
hour[,minute[,second[,microsecond[, tzinfo]]]]])
```

- 即除了年月日外，其他参数都为可选参数
- print返回格式：YYYY-MM-DD hour:minute:second[.microsecond]

构造函数

构造函数	描述
datetime.datetime(year,month,day,hour=0, minute=0, second=0, microsecond=0, tzinfo=None, fold=0)	初始化构造方法，年月日三个参数为必须
datetime.now()	
datetime.utcnow()	返回utc时区标准datetime对象
datetime.combine(date, time)	用date和time对象组合构造对象

方法

方法	说明
timetuple()	返回time.time_struct对象格式的日期时间
utctimetuple()	返回time.time_struct对象格式的utc日期时间（自带utc转换）
astimezone()	输出带时区部分的形式 +08:00
date()	返回date部分的date类
time()	返回time部分的time类
isoformat(sep)	标准格式化，日期和时间分隔符为sep, 默认yyyy-mm-ddThh:mm:ss
ctime()	Sat Sep 8 00:00:00 2018格式
strftime(tpl)	自定义格式输出，参考日期模板
replace()	替换其中的一些值

datetime类

用于datetime类的日期时间加减运算

```
datetime.timedelta(year, month, day[,
hour[,minute[,second[,microsecond[, tzinfo]]]]])
```

构造函数

构造函数	描述
datetime.timedelta(year=0,month=0,day=0,hour=0, minute=0, second=0, microsecond=0, tzinfo=None, fold=0)	初始化构造方法，默认全部是0

方法

- 两个timedelta对象之间的加、减操作
- 对一个timedelta对象取正、负、绝对值操作
- 两个timedelta对象比较大小
- 一个timedelta对象乘、除一个整数的操作
- 一个datetime对象加键一个timedelta对象

tzinfo类

表示时区，略

异常处理

Python常见错误

编程环境中的相关问题

1. py文件中输出需要print语句，但shell中不需要

```
>>> ls = ['hello'] + [1,2,3]
>>> ls
['hello',1,2,3]
```

语法错误

Python和C语言代码的书写差异

1. Python中没有++和--运算，必须用+=1和-=1实现
2. if和while中的条件不需要括号，用空格与其分开即可
3. 分号;仅在多条语句放在同一行中时需要，每行语句结束时不需要分号

```
x=1; y=2; z=3
```

4. Python中需要表达式的地方不能出现语句，如不能再while循环的条件测试中插入赋值语句

```
while (x=next()) != None: #错误
```

编程错误

1. 改变不可变类型的数据
    - 元组
    - 字符串

    正确的做法是通过切片、联结等方法构建一个新的对象，并赋给原来的变量

```
s = 'hello'
s = s[0].upper() + s[1:]
```

2. 向空列表中加入元素要用.append(element)的方法
3. 注意改变对象的方法会返回None，而不是改变后的对象
    - list.append()
    - list.sort()

```
ls = [5,2,9,1,4]
for item in ls.sort() # 错误
```

正确的做法是把调用方法和遍历分开进行：

```
ls = [5,2,9,1,4]
ls.sort()
for item in ls:
```

4. 类型转换不明确

如在字符串和数字类型变量相加时，会因为不知道该统一成什么类型而报错：

```
age = 18
print('I am' + age + 'years old.' ) ###错误
Typeerror: can only concatenate str (not "int") to str
```

正确的做法是将数字转变为str后再相加：

```
age = 18
print('I am' + str(age) + 'years old.' ) ###错误
```

5. 对象赋值问题

赋值语句不会创建对象的副本，仅仅是创建引用：

- 改变原对象中的值时，新的对象其中的值也会被改变；反之同理

```
>>>L = [1,2,3]
>>>M = ['X',L,'Y']
>>>M
['X',[1,2,3],'Y']
```

如果不希望L和M引用同一个内存对象，则需要通过一些方法创建一个新的副本避免共同引用：

a. 数字或列表用分片

```
new = seq[:]              #正序
reverse_new = seq[::-1] #逆序
```

b. 字典用复制方法

```
new = Old_dict.copy()
```

6. 函数变量作用域问题

Python中如果没有声明，默认函数中的变量是局部变量

- 而由于没有声明，只有在赋值的时候才认为该局部变量被创建，可能导致前面的其他操作报错

```
>>> x = 0
>>> def func():
    print(x)
    x = 9
>>>func()
UnboundLocalError: local variable 'x' referenced before assignment
```

可以通过在函数中提前声明x是全局变量来解决

```
>>> x = 0
>>> def func():
    global x
    print(x)
    x = 9
>>>func()
9
```

Python异常处理

常见异常的继承关系

- 异常：影响程序正常向下执行的错误

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration

+-- StandardError
| +-- BufferError
| +-- ArithmeticError
| | +-- FloatingPointError
| | +-- OverflowError
| | +-- ZeroDivisionError
| +-- AssertionError
| +-- AttributeError
| +-- EnvironmentError
| | +-- IOError
| | +-- OSError
| | +-- WindowsError (Windows)
| | +-- VMSError (VMS)
| +-- EOFError
| +-- ImportError
| +-- LookupError
| | +-- IndexError
| | +-- KeyError
| +-- MemoryError
| +-- NameError
| | +-- UnboundLocalError
| +-- ReferenceError
| +-- RuntimeError
| | +-- NotImplementedError
| +-- SyntaxError
| | +-- IndentationError
| | +-- TabError
| +-- SystemError
| +-- TypeError
| +-- ValueError
| +-- UnicodeError
| +-- UnicodeDecodeError
| +-- UnicodeEncodeError
| +-- UnicodeTranslateError
+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning

+-- BytesWarning
+– ResourceWarning

异常处理

基本流程

```
try:
    ...
except XXXError:
    ...
[except:]
[else:]
[finally:]
    ...
```

1. 程序执行try子句中的代码，若没有发生异常，直接跳到else执行其中语句；没有else的话则直接结束try...except，执行try后面的语句；
2. 如果执行中发生异常，则跳过try中剩下的部分。依次判断异常类型是否与except关键字后的异常相匹配：
    a. 如果匹配，则执行该except子句的代码；
    b. 如果不匹配，则继续顺次向后寻找；
    c. 如果except后面没有跟任何异常类型，则默认其可以处理所有类型的异常（甚至真正的程序错误和sys.exit()调用）；
    - 注释：except子句会捕获该类型异常及其子类型的所有错误，但有多个except子句匹配时，**只会执行第一个被匹配的**。
        - 所以顺序很重要！！！

```
filename = 'alice.txt'
try:
    f = open(filename)
except IOError:
    print("IOError")
except FileNotFoundError:
    print("FileNotFoundError")
else:
    print("Error Unexpected!")
    f.close()
```

因为FileNotFoundError是IOError的子类，所以第二条except语句永远不会被执行

3. 如果任何一条except语句捕捉到错误，则在执行完except中的语句后，会执行finally中的语句（如果有的话）
    - finally语句通常用于释放外部资源

raise抛出异常和异常传递

```
try:
    raise NameError('Hello?')
except NameError:
    print('NameError')
```

通过raise进行异常传递：

```
def faulty():
    raise Exception("Error")
def vomit_exception():
    try:
        faulty()
    except:
        raise       #不处理，继续向上层try吐出异常

try:
    vomit_exception()
except:
    print("Exception handled!")

```

自定义异常类

用户可以在继承Exception异常类的基础上自定义异常类型：

```
class InputError(Exception):
    def __init__(self, msg):
        self.msg = msg
    def __str__(self):      #当类被当作字符串调用时输出的内容
        return self.msg
raise InputError("Invalid Input!")
```

进阶：面向对象的程序设计

概述

面向过程和面向对象

1. 面向过程：分析解决问题所需要的步骤，然后用函数一步步的实现；
2. 面向对象：把构成问题的事物分解成若干个对象，建立对象不是为了完成一个具体的步骤，而是为了描述这个事物在整个问题解决中的行为；

注：如果没有复用性、扩展性等需求，不需要面向对象进行程序设计

面向对象的基本概念

1. 对象：现实世界中客观存在的事物；在面向对象的程序设计中 的对象是现实世界中客观事物在程序设计中的抽象，有自己的特征和行为。
    - 任何对象都由属性和方法组成
2. 属性：对象的特征，用变量表示
3. 方法：对象的行为，用函数表示
4. 抽象：从众多的事物中抽取出具有共同的、本质的特征作为一个整体，是共同特征的集合
5. 封装：通过抽象所得到的数据信息和操作进行结合，使其成为一个有机的整体。进而对内执行操作，对外隐藏这些细节和数据信息。
6. 继承：新创建的类的一些特征（包括属性和行为）可以从其他已有的类获得。新类称为子类，而已有类称为父类/基类
    - 子类可以继承父类全部的属性和方法，也可以只继承一部分
    - 子类可以从多个父类处继承属性和方法
    - 子类还可以重写父类的属性和方法
7. 多态性：不同对象收到相同的信息，但产生了不同的行为

类与对象

类

```
class 类名:
    属性1 = 值1
    ...
    方法1 = 函数1
    ...
```

- 类名一般采用“驼峰式”命名法

对象的创建与使用

对象是其类的实例化

```
对象名= 类名(<参数列表>)
```

访问实例对象的属性和方法

```
对象名.属性名
对象名.函数名(<参数>)
```

self参数和__init__函数

self参数

- Python中，类的方法都要包含self参数（一般是第一个参数）；
- 这个参数表示类的实例对象本身，用于对对象自身的引用
- 类的外部通过**对象名**调用方法时不需要传递该参数
- 类的外部通过**类名**调用方法时，需要传递实例对象的名称给这个参数

__init__函数

- 初始化函数，在对类的实例化时会被调用，以完成一些属性的初始化

```python
class Bird:
    def __init__ (self,weight):
        self.weight = weight
    def introduction(self):
        print("my weight is ", self.weight)

bird = Bird(20)
bird.introduction()
Brid.introduction(bird) # 类方法调用，需要传self
```

__del__方法

- 当某个实例对象的所有引用都被清除后，实例所占用的空间会被自动释放
- 在释放空间之前，Python会执行该方法（如果有），以进行特殊操作
- del 命令可以调用该方法

属性和方法

属性

类属性和对象属性

1. 类属性：类所有实例化对象共享的属性，在内存中只存在一个副本，直接定义在类中。
    - 公有的类属性可以在类外通过类名或对象访问
2. 对象属性：**定义在方法之中**，且有对象名前缀（通常是self.），**只能通过对象名访问**。

```python
class Person:
    #类属性
    count = 0
    def __init__(self,name,gender='M'):
        #对象属性
        self.name = name
        self.gender = gender
```

公有属性和私有属性

- 没有下划线开头的是共有属性
- 有双下划线开头的是私有属性，在类外使用使用遵循如下格式：

```
类（对象）名._类名__私有属性名
```

实例

```
class Base:
    a = 10      #公有类属性
    __x= 20     #私有类属性
    def __init__(self, value):
        print(Base.__x)
        self.__value = value    #私有对象属性
```

```
>>> b = Base(5)
20
>>> print(b._Base__x)
20
```

总结

	公有属性	私有属性
类属性	所有实例对象共用	所有实例对象共用；不能直接调用
对象属性	每个实例对象都不同	每个实例对象都不同；不能直接调用

方法

公有方法和私有方法

定义

```
class Methods:
    def publicMethod(self):
        pass
    def __privateMethod(self):
        pass
```

- 私有方法定义在前面需要加两个英文下划线__

均可以访问属于类和对象的成员，区别在于调用方法不同：

- 公有方法

```
对象名.公有方法(<参数>)
类名.公有方法(对象名)
```

- 私有方法

```
对象名._类名__私有方法名()
类名._类名__私有方法名(对象名)
```

实例方法、类方法和静态方法

实例方法即一般方法；类方法和静态方法需要通过@语句标注，他们都可以通过类名和对象名调用，但：

- 类方法**不能直接访问属于对象的成员，只能访问属于类的成员**

```
类名.类方法(<参数>)
对象名.类方法(<参数>)
```

- 静态方法两者成员都不能访问

```
类名.静态方法(<参数>)
对象名.静态方法(<参数>)
```

```
class A:
    #类属性
    explanation = 'this is my programs.'
```

```
    #实例方法
    def normalMethod(self,name):
        self.explanation = name

    #类方法
    @classmethod
    def classMethod(cls,explanation):
        cls.explanation = explanation
    #静态方法
    @staticmethod
    def staticMethod(explanation):
        print('static Method called!')
```

总结

	公有方法	私有方法
实例方法	都可以	不能直接访调用方法
类方法	不能访问实例对象成员	不能访问实例对象成员；不能直接调用方法
静态方法	不能访问对象和类成员	不能放为对象和类成员；不能直接调用方法

继承和派生

继承

单继承派生类

```
class SubClass(BaseClass):
    #类定义部分
```

多继承派生类

```
class SubClass(BaseClass1,BaseClass2,...):
    #类定义部分
```

说明

1. 子类会通过继承得到父类所有的公有方法；
    a. 私有方法不会获得
    b. 排在前面的父类同名方法会被后面的覆盖
    c. **但构造方法，前面的会遮住后面的**
2. 子类重新定义同名方法，会进行重写
3. 子类中调用父类方法为【父类名.方法名(<参数>)】

Python实现多态性

多态性

- 定义：具有不同功能的函数可以使用相同的函数名称；
- 面向对象方法中的多态性：向不同对象发送同一条消息，不同对象在接受时会产生不同的行为（方法）；
    - 每个对象可以用自己方式响应相同的信息；

python的实现

通过接受相同形式参数的不同的方法来实现

致谢

感谢闫宏飞老师在2020年秋季学期计算概论课让我有时间整理Python基础及相关代码！

2020年11月