目录

程序流程控制

程序的流程控制是通过一些程序结构来控制程序的执行顺序和流程。

一、程序结构

计算机程序是一条条按顺序执行的指令。顺序结构是是计算机程序的基础，但单一的顺序结构不能解决所有问题。计算机程序由三种基本结构组成：

- 顺序结构
- 分支结构
- 循环结构

1. 顺序结构

我们每天早上起床，穿衣服，洗脸，刷牙，叠被子，吃早餐，这些按照单一顺序进行的就是顺序结构。

顺序结构是程序执行的基本流程，它会按照代码从上往下的顺序依次执行。

```
a = 1
a += 1
print(a)
```

例如上面的代码，执行流程为

1. 定义变量a,并赋值为整数1
2. 变量a自加1
3. 调用函数print输出变量a的值

从上往下依次执行

2. 分支结构

出门时是否带伞要根据天气条件进行判断，如果天气好就不带，如果可能下雨或正在下雨就要带，这就是分支结构。

分支结构是程序根据条件判断结果而选择不同代码向前执行的一种方式，也叫条件分支。

分支结构包括：

1. 单分支
2. 二分支
3. 多分支

2.1. 单分支

在python中单分支结构通过`if`语句来实现，语法如下：

```
if <条件>:
    语句块
```

- 注意条件语句结束后要紧跟冒号
- 语句块是`if`条件满足后执行的一条或多条语句序列
- 语句块中语句通过缩进与`if`语句所在行形成包含关系
- 缩进按照规范为4个空格

`if`语句会首先计算`<条件>`表达式，如果结果为`True`则会执行所包含的语句块，结果为`False`则会跳过所包含的语句块。

`if`语句中的语句块的执行与否依赖于条件判断。但无论什么情况，控制都会 转到与if语句同级别的下一条语句。

案例：PM2.5 空气质量提醒（1）

根据输入的空气PM2.5的值，输出对应的提醒：

pm <35	空气质量优，快去户外活动
35<= pm < 75	空气良好，适量户外活动
75 <= pm	空气污染，请小心

input函数

input函数用来接收用户输入的文本信息，然后以字符串的形式返回，它接收字符串参数作为提示信息输出。

```
res = input('>>>:')
print(res)
```

```
pm = input('请输入空气PM2.5数值: ')
pm = float(pm)
if pm < 35:
    print('空气质量优，快去户外活动')
if 35 <= pm < 75:
    print('空气良好，适量户外活动')
if 75<= pm:
    print('空气污染，请小心')
```

2.2.二分支

python中二分支结构通过if-else语句来实现的，语法 格式如下：

```
if <条件>:
  <语句块1>
else:
  <语句块2>
```

- <语句块1>是在if条件满足后执行的一个或多个语句序列
- <语句块2>是if条件不满足后执行的语句序列
- 注意缩进，<语句块2>通过缩进与else所在行形成包含关系

二分支语句用于区分<条件>的两种可能True或者False，分别形成执行路径

案例：PM2.5空气质量提醒(2)

根据输入的空气PM2.5的值，输出对应的提醒：

pm <75　　空气没有污染，可以展开户外活动

pm >= 75　　空气污染，请小心

```
pm = input('请输入空气PM2.5数值: ')
pm = float(pm)
if pm <75:
    print('空气没有污染，可以展开户外活动')
else:
    print('空气污染，请小心')
```

三目运算

二分支结构还有一种更简洁的表达方式，语法格式如下：

```
<表达式> if <条件> else <表达式2>
```

适合代码块为1行语句时，这种方式也叫三目运算。

上面的代码可以改写为：

```
pm = input('请输入空气PM2.5数值: ')
pm = float(pm)
print('空气没有污染，可以展开户外活动') if pm <75 else print('空气污染，请小心')
```

对于简单判断，三目运算可以将多行语句写成一行，简洁明了。

2.3. 多分支

python 通过`if – elif – else`表示多分支结构，语法如下：

```
if <条件1>:
  <代码块1>
elif <条件2>:
  <代码块2>
...
else:
  <代码块3>
```

多分支是二分支的扩展，用于多个判断条件多条执行路径的情况。python依次执行判断条件，寻找第一个结果为True的条件，执行该条件下的代码块，同时结束后跳过整个`if–elif–else`结构，执行后面的语句。如果没有任何条件成立，则执行else下的代码块，且else语句是可选的，也即是说可以没有else语句。

案例：PM2.5空气质量提醒（3）

根据输入的空气PM2.5的值，输出对应的提醒：

pm <35	空气质量优，快去户外活动
35<= pm < 75	空气良好，适量户外活动
75 <= pm	空气污染，请小心

```
pm = input('请输入空气PM2.5数值: ')
pm = float(pm)
if pm < 35:
    print('空气质量优，快去户外活动')
elif 35 <= pm < 75:
    print('空气良好，适量户外活动')
else:
    print('空气污染，请小心')
```

思考题

编写一个根据体重和身高计算BMI值的程序，并同时输出国际和国内的BMI指标建议值

BMI的定义如下：

BMI = 体重（kg）÷ 身高^2（㎡）

标准如下：

```
height = input('请输入身高(单位米):')
weight = input('请输入体重（单位kg):')
height = float(height)
weight = float(weight)
bmi = weight/(height**2)
print('BMI数值为: {:.2f}'.format(bmi))
wto,dom = '',''
if bmi < 18.5:
    wto,dom = '偏瘦','偏瘦'
elif 18.5 <= bmi <24:
    wto,dom = '正常','正常'
elif 24 <= bmi <25:
    wto,dom = '正常','偏胖'
elif 25 <= bmi <28:
    wto,dom = '偏胖', '偏胖'
```

```
elif 28 <= bmi < 30:
    wto,dom = '偏胖','肥胖'
else:
    wto,dom = '肥胖','肥胖'
print('BMI指标为：国际{}，国内{}'.format(wto,dom))
```

3. 循环结构

工作日每天9:00到公司上班，17:30下班，周而复始，这就是循环结构。

python中循环结构有两种：

1. 条件循环也叫while循环
2. 遍历循环也叫for循环

3.1 条件循环

python中的条件循环通过while循环语句来实现，所以也叫while循环，语法格式如下：

```
while <条件>:
  代码块
```

- while关键字空格后接条件表达式末尾加上冒号组成while语句
- 代码块中的代码通过4个空格和while语句形成包含关系

while语句首先计算<条件>表达式，如果结果True，则执行对应代码块中的语句，执行结束后再次执行<条件>

表达式，再次判断结果，如果为True则循环执行，直到<条件>表达式为False时跳出循环，执行和while语句相同缩进的下一条语句。

当<条件>表达式恒为True时，形成无限循环，也叫死循环，需要小心使用。

```
# 依次打印0-9
i = 0
while i < 10:
    print(i)
    i = i + 1
```

```python
# 序列的循环
# 列表的元素循环
ls = [0,1,2,3,4,5,6,7,8,9]
index = 0
while index < len(ls):
    print(ls[index])
    index += 1
```

```python
# 散列的循环
# 集合没办法循环
dc = {'name': 'felix', 'age': 18}
index = 0
# 转换成列表
keys = list(dc.keys())
while index < len(keys):
    print(dc[keys[index]])
    index += 1
```

3.2 遍历循环

python中使用关键字`for`来实现遍历循环，也叫`for`循环，也叫迭代循环，语法格式如下：

```
for <循环变量> in <遍历结构>:
  代码块
```

- 关键字`for`+空格+`<循环变量>`+关键字`in`+`<遍历结构>`+冒号组成for语句
- 代码块通过缩进和for语句形成包含关系

for 循环会依次取出遍历结构中的元素，然后赋值给循环变量，每次遍历都会执行代码块，只到取出遍历结构中的所有元素。

所有可迭代对象都可以作为遍历结构进行for循环。

基本数据类型中序列数据类型，散列数据类型都可以进行迭代。

```python
# for循环来遍历可迭代对象非常方便
# 序列的迭代
# 列表的迭代
ls = [0,1,2,3,4,5,6,7,8,9]
for i in ls:
    print(i)
```

```python
# 散列的迭代
st = {1,2,3,4,5,6}
for i in st:
    print(i)
```

```python
dc = {'name': 'felix', 'age': 18}
# 字典key的迭代
for key in dc:
    print(key)
```

```python
# 字典值的迭代
for key in dc:
    print(dc[key])
```

```python
for value in dc.values():
    print(value)
```

迭代器（iterator）

迭代器是一个可以记住遍历位置的对象。for循环迭代本质上就是通过迭代器来实现的。

通过内建函数`iter`可以创建迭代器。

```python
iter('abc')
```

不是所有的数据类型都可以创建迭代器，凡是能够创建迭代器的对象称为可迭代对象，反之是不可迭代对象

range函数

内建函数`range`可以创建输出整数序列的迭代器。

```python
range(start, stop,step)
```

range(i,j)生成`i,i+1,i+2,...,j-1`,start默认为0，当给定step时，它指定增长不长。

```python
# 输出0-9
for i in range(10):
    print(i)
```

```python
# 输出1-10
for i in range(1,11):
    print(i)
```

```python
# 输出-1--10
for i in range(-11,-1,-1):
    print(i)
```

for循环经常和range函数配合用来指定循环次数。

3.3 循环控制关键字

循环有时候需要主动中断来提高程序执行效率。

```python
ls = [60,59,78,80,56,55]
# ls中存放的是所有学生的成绩
# 要判断是否有同学及格
for i in ls:
    if i >= 60:
        print('有同学及格')
```

可以发现上面的案例中，其实第一个成绩就及格了，但是程序继续循环下去，如果数据量小，效率差别不大，但数据量大时会影响程序的执行效率。在实际的代码编写中会有很多这种情况，这是就需要能够主动结束循环的能力。

break关键字

python中循环结构可以使用break跳出当前循环体，脱离该循环后代码继续执行。

```python
for i in ls:
    if i >= 60:
        print('有同学及格')
        break
index = 0
while index < len(ls):
    if ls[index] >= 60:
        print('有同学及格')
        break
```

注意break只会跳出当前循环

```python
for i in range(1,4):
    for j in range(1,4):
        if i==2:
            break
        print(i,j)
```

continue关键字

python中循环结构还可以使用continue关键字用来跳出当次循环，继续执行下一次循环。

```python
for i in range(10):
    if i%2 == 0:
        continue
    print(i)
```

```python
for i in range(10):
    if i%2 == 0:
        break
    print(i)
```

else关键字

循环结构还可以通过和else关键字进行配合，用来检测循环是否正常循环结束，还是break掉了。

```python
for i in range(10):
    if i%2 != 0:
        continue
    print(i)
else:
    print('循环正常结束')
```

```python
for i in range(10):
    if i%2 != 0:
        break
    print(i)
else:
    print('循环正常结束')
```

二、异常处理

在程序的编写过程中会出现各种错误，语法错误在程序启动时就会检测出来，它是程序正常运行的前提条件。程序中还有一种错误发生在程序运行后，可能是由于逻辑问题，又或者是业务发生了改变，为了能让用户有更好的体验，加强代码的健壮性，我们需要对这些错误进行处理，也叫异常处理。

在python中通过try-except语句进行异常处理。

回忆我们前面关于空气质量提醒的案例，当用户输入非数值时程序会发生什么?

```
pm = input('请输入空气PM2.5数值: ')
pm = float(pm)
if pm <75:
    print('空气没有污染，可以展开户外活动')
else:
    print('空气污染，请小心')
```

try-except语句的基本语法格式如下:

```
try:
  <语句块1>
except <异常类型1>:
  <语句块2>
except <异常类型2>:
    <语句块3>
```

语句块1中的代码如果发生异常，且异常与类型与对应excep语句中的异常类型相同则会被其捕获，从而执行对应的语句块

```
while True:
    try:
        pm = input('请输入空气PM2.5数值: ')
        pm = float(pm)
        break
    except ValueError as e:
        print(e) # 输出异常提示信息
        print('请输入整数')
if pm < 35:
    print('空气质量优，快去户外活动')
```

```
if 35 <= pm < 75:
    print('空气良好，适量户外活动')
if 75<= pm:
    print('空气污染，请小心')
```

除了try和except关键字外，异常语句还可以与else和finally关键字配合使用，语法格式如下：

```
try:
  <语句块1>
except <异常类型>:
  <语句块2>
...
else:
  <语句块3>
finally:
  <语句块4>
```

代码执行流程如下：

没有发生异常时，会执行else语句后的代码块,不管有没有发生异常，finally语句后的代码块一定会执行

```
try:
    a = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    index = input('请输入一个正整数>>>:')
    index = int(index)
    print(a[index])
except Exception as e:
    print(e)
    print('请输入一个正整数')
else:
    print('没有发生异常')
finally:
    print('我一定会执行')
```

三、函数和代码复用

看下面一段伪代码：

```
if cpu使用率 >80%:
    连接邮箱
    发送邮件
    关闭邮箱
if 内存使用率 >80%:
    连接邮箱
    发送邮件
    关闭邮箱
if 硬盘使用率 >80%:
    连接邮箱
    发送邮件
    关闭邮箱
```

思考这段代码有什么问题?

1. 函数的概念

函数是一段具有特定功能的，可重用的语句组，用函数名来表示并通过函数名进行完成功能调用。

函数也可以看作是一段具有名字的子程序，可以在需要的地方调用执行，不需要再每个执行地方重复编写这些语句。每次使用函数可以提供不同的参数作为输入，以实现对不同数据的处理；函数执行后，还可以以反馈相应的处理结果。

函数是一种功能抽象。

2. python中函数的定义

Python定义一个函数使用`def`关键字，语法形式如下：

```
def <函数名>(<参数列表>):

    <函数体>

    return <返回值列表>
```

实例：生日歌

过生日时要为朋友唱生日歌，歌词为：

```
Happy birthday to you!

Happy birthday to you!

Happy birthday, dear<名字>

Happy birthday to you!
```

编写程序为Mike和Lily输出生日歌。最简单的方式是重复使用print()语句

```
# 最简单的方式
print('Happy birthday to you! ')
print('Happy birthday to you! ')
print('Happy birthday, dear Mike! ')
print('Happy birthday to you! ')

print('Happy birthday to you! ')
print('Happy birthday to you! ')
print('Happy birthday, dear Lily! ')
print('Happy birthday to you! ')
```

以函数的方式

```
# 定义函数
def happy():
    print('Happy birthday to you! ')

def happyB(name):
    happy()
    happy()
    print('Happy birthday, dear {}! '.format(name))
    happy()

# 调用函数
happyB('Mike')
print()
happyB('Lily')
```

3. 函数的调用过程

程序调用一个函数需要执行以下四个步骤：

1. 调用程序在调用处暂停执行
2. 在调用时将实参复制给函数的形参
3. 执行函数体语句
4. 函数调用结束给出返回值，程序回到调用前的暂停处继续执行

上面的happyB函数的调用过程

4. 函数的参数

定义函数时()里的参数叫形参（形式参数），它只是一个变量名，供函数体中的代码调用。

函数调用时，传入()里的参数叫实参（实际参数），它是实际的数据，会传递给形参，供函数体执行。

4.1 形参

定义函数时，形参根据功能不同，可以定义几种类型。

4.1.1 必须参数

在定义函数时，如果要求调用者必须传递实参给这个形参，它就是必须参数。

直接定义在函数名后的()中的形参就是必须参数。

例如上面的happyB函数中的name。

案例：

定义一个函数接收两个数，然后打印它们的和

```
1
2 def add(x,y):
3     print(x+y)
```

```
1 add(1)  # 调用时必须传递实参给必须参数，否则报错
```

4.1.2 默认参数

在定义函数时，某些形参有可能在调用时不用接收实参，这种情况可以定义为默认参数。

在函数名后()中，以参数名=默认值的形式定义的形参就是必须参数。

注意：默认参数必须定义在必须参数的后面

案例：

定义一个函数，它接收两个参数content和times，

content是函数要打印的内容

times是函数打印的次数，如果不传递times默认打印1次

```
# 定义
def my_print(content, times=1):
    for i in range(times):
        print(content)
# 调用
my_print('happy birthday! ')
my_print('happy birthday! ', 2)
```

调用函数时传递实参给默认形参会覆盖默认值。

4.1.3 不定参数

在定义函数时，不确定在调用时会传递多少个实参时，可以定义不定参数。

不定参数根据传递实参的不同（详见4.2实参）有分为两种。

位置不定参

在函数名后的()中，在形参前加 *号可以定义位置不定参，通常它会定义为 *args。

它用来接收函数调用时，以位置参数传递过来的超过形参数量的多余的实参。

注意：不订参必须定义在默认参数后面

位置不定参数会将所有多余的位置实参创建成元组。

```python
def func(a, *args):
    print(args,type(args))

func(1,2,3,4)
```

案例：

定义一个函数，接收2个以上的数，打印它们的和。

```python
def add(x,y,*args):
    x += y
    for i in args:
        x += i
    print(x)
add(1,2,3,4)
```

关键字不定参

在函数名后的()中，在形参前加 **号可以定义关键字不定参，通常它会定义为 **kwargs。

它用来接收函数调用时，以关键字参数传递过来的超过形参数量的多余的实参。

注意：不订参必须定义在默认参数后面

关键字不定参数会将所有多余的关键字实参创建成字典。

```python
def func(a, **kwargs):
    print(kwargs,type(kwargs))

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

4.2 实参

调用函数是传递实参有两种方式。

4.2.1 位置参数

调用函数时，传递实参时默认会按照形参的位置一一对应，这种实参传递叫做位置参数。

案例

定义一个函数实现打印一个数的n次幂。

```
def my_power(x, n):
    print(x**n)

my_power(3,2)
my_power(2,3)
```

4.2.2 关键字参数

调用函数时，传递实参时以形参名=实参的形式传递参数，叫做关键字参数。

这是不用考虑参数的位置。

注意：关键字参数必须写在位置参数后面。

案例

使用关键字参数调用上面的案例

```
my_power(x=3,n=2)
my_power(n=2,x=3)
my_power(3,n=2)
my_power(n=2,3)
```

4.2.3 *,**在传递实参时的用法

***解包**

在传递实参时，可以通过*对迭代对象进行解包。

```
def fun(a,b,*arg):
    print(a,b,arg)
ls = [1,2,3,4,5,6]
fun(*ls) # => fun(1,2,3,4,5,6)
```

```
1 2 (3, 4, 5, 6)
```

****解包**

在传递实参时，可以通过**对字典对象进行解包。

```

```

```python
def fun(a,b, **kwargs):
    print(a,b,kwargs)
d = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
fun(**d) # => fun(a=1,b=2,c=3,d=4)
```

```
1 2 {'c': 3, 'd': 4}
```

4.3 返回值

函数还有一个很重要的功能就是返回结果。

python中使用`return`关键字来退出函数，返回到函数被调用的地方继续往下执行。

`return` 可以将0个，1个，多个函数运算完的结果返回给函数被调用处的变量。

函数可以没有返回值，也就是说函数中可以没有`return`语句，这时函数返回`None`,例如上面我们定义的那些函数。

`return`会将多个返回值以元组的形式返回。

案例：

定义一个函数接收2个或多个数值，并返回它们的和。

```python
def add(x,y,*args):
    x += y
    for i in args:
        x += i
    return x

res = add(1,2)
print(res)
```

```
3
```

定义一个函数接收被除数x和除数y，返回它们的商和余数。

```python

def my_mod(x,y):
    res1 = None
    res2 = None
    if x < y:
        res1 = x
        res2 = 0
```

```
    else:
        i = 0
        while x >= y:
            x = x-y
            i += 1
        res1 = i
        res2 = x

    return res1, res2
res = my_mod(10,3)
print(res)
```

```
(3, 1)
```

5. lambda函数

简单来说，lambda函数用来定义简单的，能够在一行内表示的函数。

语法格式如下：

```
lambda arg1,arg2,... : expression
```

案例

```
f = lambda x,y : x + y
res = f(1,2)
print(res)
```

```
3
```

lambda函数一般不会直接定义，通常是作为参数传递给其他函数作为参数使用。

6. 变量作用域

python中一个变量能够被访问的范围叫做作用域。根据作用域的大小简单的分为全局变量和局部变量。

6.1 全局变量

python是解释型编程语言，解释器在运行一个python程序时会在计算机内存中申请一块内存用来运行这个程序。全局变量在这块内存空间中都可以被访问和修改。

直接定义在函数外的变量就是全局变量，在程序运行的全过程有效。

6.2 局部变量

定义在函数里，的变量就是局部变量，它只在它定义的函数里起作用，一旦函数执行完毕它就不存在了。

案例

```
a = 1 # 全局变量

def fun():
    print(a)
fun()
```

```
上面的案例说明全局变量能够在函数里访问。
```

```
def fun():
    a = 2  # 局部变量
    print(a)

fun()
print(a)
```

```
2
2
```

上面的案例说明局部变量在函数外部不能访问

```
a = 1
def fun():
    a += 1  # 尝试直接在函数内部修改全局变量
    print(a)
fun()
```

上面的案例说明在函数内部不能直接修改全局变量

6.2 globals关键字

有时候需要在函数内部修改全局变量。

使用`globals`关键字可以在函数内部修改全局变量

案例：

```
a = 1 # 全局变量

def fun():
    global a  # 申明 a 是全局变量
    a += 1
fun()
print(a)
```

```
2
```

四、python内建函数

python解释器提供了70多个内置函数。

```
import builtins
print(dir(builtins))
```

小写的就是内置函数。

常用内建函数

abs()

abs（*x*）¶

返回数字的绝对值。参数可以是整数或浮点数。如果参数为复数，则返回其大小。如果*x*定义__abs__()，则 abs(x)返回x.__abs__()。

```
abs(-1)  # 返回 1
```

all()

all（*可迭代*）¶

返回True如果的所有元素*迭代*是真实的（或者如果可迭代为空）。相当于：

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

```
ls = [1,2,3,4]
all(ls) # -> False
```

```
True
```

bin()

`bin（x）`

将整数转换为以“ 0b”为前缀的二进制字符串。结果是有效的Python表达式。如果*x*不是Python `int`对象，则必须定义一个`__index__()`返回整数的方法。

```
bin(3)
```

```
'0b11'
```

```
bin(-10)
```

```
'-0b1010'
```

chr()

`chr（i）`

接收一个字符的unicode整数，返回这个字符。例如，`chr(97)`返回字符串`'a'`，而`chr(8364)`返回字符串`'€'`。这是函数`ord()`的反函数。

```
chmr(97)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-19-b23a96a0e283> in <module>
----> 1 chmr(97)
```

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

```
chr(8364)
```

```
'€'
```

dir()

dir（[*对象*]）¶

尝试返回该对象的有效属性列表。

divmod()

divmod（*a*，*b*）

使用两个（非复数）数字作为参数，并在使用整数除法时返回一对包含其商和余数的数字。

```
divmod(10,3)
```

```
(3, 1)
```

enumerate()

enumerate（*iterable*，*start = 0* ）

返回一个枚举对象。*iterable*必须是序列， 迭代器或其他支持迭代的对象。__next__()由enumerate()返回的迭代器的方法 返回一个元组，该元组包含一个计数（从*start开始*，默认为0），以及通过对*iterable进行迭代而*获得的值。

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
list(enumerate(seasons))
```

```
list(enumerate(seasons, start=1))
```

经常在迭代对象时又想知道迭代计数可以这样写：

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
for index, item in enumerate(seasons):
    print(index, item)
```

help()

help（[*对象*] ）¶

调用内置的帮助系统。

hex()

hex（*x* ）¶

将整数转换为以“ 0x”为前缀的小写十六进制字符串。

```
hex(10)
```

id()

id（*对象*）¶

返回对象的“身份”。这是一个整数，可以保证在此对象的生存期内唯一且恒定。具有不重叠生命周期的两个对象可能具有相同的id() 值。

CPython实现细节： 这是对象在内存中的地址。

input()

`input`（[*提示*] ）¶

如果存在*提示*参数，则将其写入到标准输出中，而无需尾随换行符。然后，该函数从输入中读取一行，将其转换为字符串（将尾随换行符分隔），然后将其返回。

isinstance()

`isinstance`（*object*，*classinfo* ）

接收一个对象和一个类，如果对象是这个类的对象或者这个类的子类（直接，间接）的对象则返回True，否则返回False。

```
1 isinstance(1, int)
```

issubclass

`issubclass`（*class*，*classinfo* ）¶

如果`class`是`classinfo`的子类（直接，间接）则返回True，否则返回False。

```
1 issubclass(bool, int)
```

iter()

`iter`（*object* [，*sentinel*] ）

返回一个迭代器对象,如果对象不支持迭代则抛出异常。

len()

`len`（*s* ）

返回对象的长度（项目数）。参数可以是序列（例如字符串，字节，元组，列表或range）或集合（例如字典，集合）。

map()

`map`（*函数*，*可迭代*，... ）

返回一个迭代器，该迭代器将*函数*应用于*iterable*的每个项目，并产生结果。

```python
# 将列表中的元素进行2次幂运算，生成新的列表
ls = [1,2,3,4,5]
ls_new = []
for item in map(lambda x: x**2, ls):
    ls_new.append(item)

# 列表生成式
ls_n = [x for x in map(lambda i: i**2, ls)]
```

max()

max（*迭代*，**[，键，默认*]）

max（*arg1*，*arg2*，**args*[，*键*]）

返回可迭代的最大项目或两个或多个参数中的最大一个。

如果提供了一个位置参数，则它应该是可迭代的。返回迭代器中最大的项。如果提供了两个或多个位置参数，则返回最大的位置参数。

```python
ls = [1,2,3,4]
max(ls)
```

```python
max(3,4,5)
```

min()

min（*arg1*，*arg2*，**args*[，*键*]）

返回可迭代的最小项或两个或多个参数中的最小项。

如果提供了一个位置参数，则它应该是可迭代的。返回iterable中的最小项。如果提供了两个或多个位置参数，则返回最小的位置参数。

```python
ls = [1,2,3,4]
min(ls)
```

```python
min(3,4,5)
```

oct()

`oct（x）` ¶

将整数转换为以“ 0o”为前缀的八进制字符串。结果是有效的Python表达式。

```
oct(8)
```

```
oct(-56)
```

ord()

`ord（c）` ¶

返回一个字符的unicode码的十进制整数。例如， `ord('a')`返回整数`97`，`ord('€')`（欧元符号）返回`8364`。这是的反函数`chr()`。

pow()

`pow（x，y）`

返回x的y次幂。

```
pow(2,3)
```

reversed()

`reversed（seq）` ¶

返回一个反向迭代器。

```
ls = [1,2,3,4]
[x for x in reversed(ls)]
```

round()

`round（number[，ndigits]）` ¶

返回数字舍入到小数点后的n位精度。如果*ndigits*省略或为`None`，则返回与其输入最接近的整数。

```
round(1.3)
```

```
round(1.5)
```

```
round(1.923, 2)
```

sorted()

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

从*iterable*中的项目返回一个新的排序列表。

有两个可选参数，必须将其指定为关键字参数。

*key*指定一个自变量的函数，该函数用于从*iterable*中的每个元素中提取一个比较键（例如`key=str.lower`）。默认值为`None`（直接比较元素）。

*reverse*是一个布尔值。如果设置为`True`，则对列表元素进行从大到小的排序。

sum()

`sum`（*迭代*），求和。

```
sum([1,2,3])
```

zip()

`zip`（* *iterables* ）¶

制作一个迭代器，聚合每个可迭代对象中的元素。

```
x = [1, 2, 3]
y = [4, 5, 6]
zipped = zip(x, y)
list(zipped)
```

```
[(1, 4), (2, 5), (3, 6)]
```

更多方法详见官方文档

五、面向对象

前面我们讲到基本数据类型用来表示最常见的信息。但是信息有无穷多种，为了更好的表达信息，我们可以创建自定义数据类型。

1. 类

1.1 类的概念

一种数据类型就是类。例如整数，浮点数，字符串。

1.2 类的定义

python中通过关键字`class`可以定义一个自定义数据类型，基本语法如下:

```
class 类名:
    属性
    方法
```

注意：python中类名规则同变量，一般使用大驼峰来表示。

案例

例如：创建一个`Point`类用于表示平面坐标系中的一个点

```
class Point:
    """
    表示平面坐标系中的一个点
    """
```

```
### 2. 对象

#### 2.1 对象的概念

某种数据类型的一个具体的数据称为这个类的一个对象或者实例。通过类创建对象叫做实例化。

所谓的面向对象，就是把一些数据抽象成类的思想。

python是一门面向对象的编程语言，python中一切皆对象。

前面学习的函数也是python中的一个类，定义的某个函数就是函数类的一个具体实例。
```

```
def func():
    pass
type(func)
```

2.2 实例化

除了基本数据类型实例化的过程中用到的特殊的语法规范外，所有自定义类型进行实例化都是通过调用类名来实现的，非常简单，语法如下：

```
类名(参数)
```

看起来和调用函数一样。

案例

给上面创建的Point类创建一个实例。

```
point = Point()
type(point)
```

3. 属性

数据的特征称为属性。

3.1类属性

类的特征称为类属性。

3.1.1类属性的定义

直接在类中定义的变量（与class语句只有一个缩进），就是类属性。

案例：

给Point类创建一个name属性用来表示名称。

```
class Point:
    """
    表示平面坐标系中的一个点
    """
    name = '点'
```

3.1.2 类属性的访问

类属性可以直接通过类名和对象以句点法访问，语法格式如下：

```
类名.类属性名
对象.类属性名
```

案例：

```
Point.name  # 直接通过类名访问类属性
point=Point()  # 创建一个实例
point.name     # 通过对象访问类属性
```

注意：如果不存在属性则抛出AttributeError的异常

3.2 对象属性

对象的特征称为对象属性。

3.2.1 对象属性的定义

对象属性一般定义在构造方法中，详见下面构造方法一节。

通过句点法对象.对象属性以赋值的方式可以直接定义对象属性。

案例：

平面坐标系中的每个点都有x坐标和y坐标，通过类Point创建一个对象表示点（x=1，y=2）

```
point = Point()
# 通过赋值直接定义对象属性
point.x = 1
point.y = 2
```

注意：在定义对象属性时如果和类属性同名，那么通过对象将无法访问到类属性。

3.2.2 对象属性的访问

通过句点法对象.对象属性可以访问对象属性。

案例：

访问上面案例中point的x坐标和y坐标

```
print(point.x)
print(point.y)
```

访问对象属性时，首先会检查对象是否拥有此属性，如果没有则去创建对象的类中查找有没有同名的类属性，如果有则返回，如果都找不到则抛出AttributeError的异常

4. 方法

定义在类中的函数称为方法。通过调用的方式的不同，分为对象方法，类方法和特殊方法。

4.1 对象方法

通过对象调用的方法成为对象方法。

4.1.1 对象方法的定义

为了讲清楚对象方法的定义和调用，我们先看下面的案例。

案例：

定义函数my_point,它接收一个Point对象，然后打印这个点的x，y坐标。

```
def my_point(point):
    print('({},{})'.format(point.x, point.y))

p = Point()
p.x = 1
p.y = 2
my_point(p)

```

定义函数distance，它接收两个Point对象，然后返回这两个点的距离。

```
def distance(p1, p2):
    return ((p1.x-p2.x)**2 + (p1.y-p2.y)**2)**0.5

p1 = Point()
p2 = Point()
p1.x = 1
p1.y = 2
p2.x = 3
p2.y = 4
res = distance(p1,p2)
print(res)
```

观察上面的两个函数，发现它们都接收一个或多个Point的对象作为参数。为了显式的加强这样的联系，我们可以将它们定义在Point的类中。

```python
class Point:
    """
    表示平面坐标系中的一个点
    """
    name = '点'

    def my_point(point):
        print('({},{})'.format(point.x, point.y))

    def distance(p1, p2):
        return ((p1.x-p2.x)**2 + (p1.y-p2.y)**2)**0.5
```

4.1.2 对象方法的调用

对象方法向属性一样，可以通过句点法进行调用。

```
类名.方法名(参数)
对象.方法名(参数)
```

通过类名调用方法时，和普通函数没有区别

```python
Point.my_point(point)
res = Point.distance(p1, p2)
print(res)
```

通过对象调用方法时，对象本身会被隐式的传给方法的第一个参数

```python
point.my_point()
res = p1.distance(p2)
print(res)
```

因此，定义对象方法会习惯性的把第一个形参定义为`self`，表示调用对象本身

```python
class Point:
    """
    表示平面坐标系中的一个点
    """
    name = '点'

    def my_point(self):
        print('({},{})'.format(self.x, self.y))

    def distance(self, p2):
        return ((self.x-p2.x)**2 + (self.y-p2.y)**2)**0.5
```

4.2 类方法

略，详见官方文档

4.3 特殊方法（魔术方法）

在类中可以定义一些特殊的方法用来实现特殊的功能，也称为魔术方法。这些方法一般都以双下划线__开头

__init__

__init__又叫构造方法，初始化方法，在调用类名实例化对象时，构造方法会被调用，类名括号()后的参数会传递给构造方法，对象属性一般在这个方法中定义。

案例：

上面案例中的Point类实例化后，需要手动创建对象属性x和y，这显然容易出错和不规范，正确的做法应该是在构造方法中定义属性x和y

```python

class Point:
    """
    表示平面坐标系中的一个点
    """
    name = '点'

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

    def my_point(self):
```

```
        print('({},{})'.format(self.x, self.y))

    def distance(self, p2):
        return ((self.x-p2.x)**2 + (self.y-p2.y)**2)**0.5

# 实例化
p1 = Point(1, 2)
p2 = Point(x=3, y=4)

```

__str__

__str__方法在对象被print函数打印时被调用，print输出__str__方法返回的字符串。

案例：

上面案例中Point类里的my_print方法可以去掉，定义一个__str__方法

```

class Point:
    """
    表示平面坐标系中的一个点
    """
    name = '点'

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

    def distance(self, p2):
        return ((self.x-p2.x)**2 + (self.y-p2.y)**2)**0.5

    def __str__(self):
        return '({},{})'.format(self.x, self.y)
p = Point(1,2)
print(p)
```

更多的特殊方法详见官方文档

5. 类的继承

略，见[官方文档

六、模块和包

如果你从Python解释器退出并再次进入，之前的定义（函数和变量）都会丢失。因此，如果你想编写一个稍长些的程序，最好使用文本编辑器为解释器准备输入并将该文件作为输入运行。这被称作编写 *脚本* 。随着程序变得越来越长，你或许会想把它拆分成几个文件，以方便维护。你亦或想在不同的程序中使用一个便捷的函数， 而不必把这个函数复制到每一个程序中去。

1. 模块

1.1 概念

为支持这些，Python有一种方法可以把定义放在一个文件里，并在脚本或解释器的交互式实例中使用它们。这样的文件被称作 *模块* ；模块中的定义可以 *导入* 到其它模块或者 *主* 模块（你在顶级和计算器模式下执行的脚本中可以访问的变量集合）。

模块是一个包含Python定义和语句的文件。文件名就是模块名后跟文件后缀 `.py` 。

案例：

使用你最喜欢的文本编辑器（当然不建议windows记事本）在当前目录下创建一个名为`fib.py`的文件，文件中含有以下内容。

```python
# 斐波那契数列 模块

def fib(n):    # 打印斐波那契数列到数n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

1.2 导入模块

通过关键字import可以在代码中导入写好的模块，语法如下：

```
import 模块名
```

案例：

现在进入解释器，并用一下代码导入fibo.py

```
import fibo
```

在当前变量表中，不会直接定义模块中的函数名，它只是定义了模块名。接下来，通过模块名fibo就可以访问其中的函数

```
fibo.fib(10)
fibo.fib2(10)
```

import语句有一个变体，它可以把模块中的名称（函数名，变量，类名）直接导入到当前模块的变量表里，语法如下：

```
from 模块名 import 名称
```

案例：

```
from fibo import fib, fib2
```

这并不会把被调模块名引入到局部变量表里，而是将函数名fib,fib2引入，现在可以直接访问这两个函数了。

```
fib(10)
fib2(10)
```

还有一个变体可以导入模块中定义的所有名称，语法如下：

```
from 模块名 import *
```

这会导入所有非以下划线（_）开头的名称。 在多数情况下，Python程序员都不会使用这个功能，因为它在解释器中引入了一组未知的名称，而它们很可能会覆盖一些你已经定义过的东西。

注意通常情况下从一个模块或者包内导入 * 的做法是不太被接受的， 因为这通常会导致代码的可读性很差。不过，在交互式编译器中为了节省打字可以这么用。

如果模块名称之后带有 as，则跟在 as 之后的名称将直接绑定到所导入的模块。语法如下：

```
import 模块名称 as 新名称
from 模块名称 import 名称 as 新名称
```

案例：

```
import fibo as fib
fib.fib(10)
```

这会和 import fibo 方式一样有效地调入模块， 唯一的区别是它以 fib 的名称存在的。

这种方式也可以在用到 from 的时候使用，并会有类似的效果：

```
from fibo import fib as fibonacci
fibonacci(10)
```

注意：出于效率的考虑，每个模块在解释器会话中只被导入一次。

2. 包

模块的问题解决了代码过长不便于维护问题，但是如果不同人编写的模块名相同怎么办？为了避免模块名冲突，python又引入了用目录来组织模块的方法，称为包。

为了避免fibo.py与其他模块冲突，我们可以选择一个顶层包名，例如：my_fibo，然后创建名为my_fibo的文件夹，将模块fibo.py放入该文件夹下。然后通过import 包名.模块名的方式引入，只要顶层包名不起冲突，模块就不会起冲突。现在fibo模块的引入就是这样的：m

```
import my_fibo.fibo
```

但是这样导入引用时也要用全名

```
my_fibo.fibo
```

也可以通过结合from引用

```
from my_fibo import fibo
```

这样可以是用fibo调用模块中的函数

```
 fibo.fib(10)
```

也可以直接导入所需的函数或变量

```
from my_fibo.fibo import fib
```

这样可以直接调用fib函数

```
fibo.fib(10)
```

请注意，当使用 `from package import item` 时，item可以是包的子模块（或子包），也可以是包中定义的其他名称，如函数，类或变量。 `import` 语句首先测试是否在包中定义了item；如果没有，它假定它是一个模块并尝试加载它。如果找不到它，则引发 `ImportError` 异常。

相反，当使用 `import item.subitem.subsubitem` 这样的语法时，除了最后一项之外的每一项都必须是一个包；最后一项可以是模块或包，但不能是前一项中定义的类或函数或变量。

注意每一个包根目录下都有一个`__init__.py`的文件，这个文件必须存在，因为python就是通过它来是吧普通文件夹和包。`__init__.py`中

3 标准模块（标准库）

Python附带了一个标准模块库实现一些常用功能，甚至有些模块内置在python解释器中。例如内置函数都不需要导入而直接使用。

下面介绍一下常用标准库。

Math库

1. 介绍

math库是python内置的数学类函数库

math库不支持复数类型，复数类型请使用cmath模块中的同名函数。

math库一共提供了4个数学常数和44个函数

44个函数共分为4类，包括：16个数值表示函数，8个幂对函数，16个三角对数函数和4个高等特殊函数

2. 数学常数

- `math.pi`¶

  数学常数 π = 3.141592...，精确到可用精度。

```
import math
math.pi
```

- `math.e`

  数学常数 e = 2.718281...，精确到可用精度。

```
math.e
```

- `math.tau`

  数学常数 τ = 6.283185...，精确到可用精度。Tau 是一个圆周常数，等于 2π，圆的周长与半径之比。更多关于 Tau 的信息可参考 Vi Hart 的视频 Pi is (still) Wrong。吃两倍多的派来庆祝 Tau 日 吧！*3.6 新版功能.*

- `math.inf`

  浮点正无穷大。（对于负无穷大，使用 `-math.inf` 。）相当于 `float('inf')` 的输出。*3.5 新版功能.*

- `math.nan`

  浮点“非数字”（NaN）值。 相当于 `float('nan')` 的输出。

3. 数值表示函数

- `math.ceil(x)`¶

返回 *x* 的上限，即大于或者等于 *x* 的最小整数。如果 *x* 不是一个浮点数，则委托 `x.__ceil__()`，返回一个 `Integral` 类的值。

- `math.fabs(x)`¶

  返回 *x* 的绝对值。

- `math.factorial(x)`

  以一个整数返回 *x* 的阶乘。 如果 *x* 不是整数或为负数时则将引发 `ValueError`。

- `math.floor(x)`

  返回 *x* 的向下取整，小于或等于 *x* 的最大整数。如果 *x* 不是浮点数，则委托 `x.__floor__()` ，它应返回 `Integral` 值。

4. 幂对函数

- `math.log`(*x*[, *base*])¶

  使用一个参数，返回 *x* 的自然对数（底为 *e* ）。使用两个参数，返回给定的 *base* 的对数 *x* ，计算为 `log(x)/log(base)` 。

- `math.log1p`(*x*)

  返回 *1+x* (base *e*) 的自然对数。以对于接近零的 *x* 精确的方式计算结果。

- `math.log2`(*x*)

  返回 *x* 以2为底的对数。这通常比 `log(x, 2)` 更准确。*3.3 新版功能.*参见 `int.bit_length()` 返回表示二进制整数所需的位数，不包括符号和前导零。

- `math.log10`(*x*)

  返回 *x* 底为10的对数。这通常比 `log(x, 10)` 更准确。

- `math.pow`(*x*, *y*)

  将返回 `x` 的 `y` 次幂。特殊情况尽可能遵循C99标准的附录'F'。特别是， `pow(1.0, x)` 和 `pow(x, 0.0)` 总是返回 `1.0` ，即使 `x` 是零或NaN。 如果 `x` 和 `y` 都是有限的， `x` 是负数， `y` 不是整数那么 `pow(x, y)` 是未定义的，并且引发 `ValueError` 。与内置的 ** 运算符不同， `math.pow()` 将其参数转换为 `float` 类型。使用 ** 或内置的 `pow()` 函数来计算精确的整数幂。

- `math.sqrt`(*x*)

  返回 *x* 的平方根。

其他详见官方文档

random库

1. 介绍

该模块实现了各种分布的伪随机数生成器。

使用random库主要目的是生成随机数，因此，只需要查阅该库的随机数生成函数，找到符合使用场景的函数使用即可。这个库提供了不同类型的随机数函数，所有函数都是基于最基本的`random.random()`函数扩展而来。

2.整数随机函数

- `random.randrange`(*stop*)¶
- `random.`randrange`(*start*, *stop*[, *step*])

从 `range(start, stop, step)` 返回一个随机选择的元素。 这相当于 `choice(range(start, stop, step))`，但实际上并没有构建一个 range 对象。位置参数模式匹配 `range()` 。不应使用关键字参数，因为该函数可能以意外的方式使用它们。*在 3.2 版更改:* `randrange()` 在生成均匀分布的值方面更为复杂。 以前它使用了像`int(random()*n)`这样的形式，它可以产生稍微不均匀的分布。

- `random.randint(`*a*, *b*`)`

  返回随机整数 *N* 满足 `a <= N <= b`。相当于 `randrange(a, b+1)`。

2.序列随机函数

- `random.``choice(`*seq*`)`¶

从非空序列 *seq* 返回一个随机元素。 如果 *seq* 为空，则引发 `IndexError`。

- `random.``shuffle(`*x*[, *random*]`)`¶

将序列 *x* 随机打乱位置。

3. 实数随机函数

- `random.random()`¶

  返回 [0.0, 1.0) 范围内的下一个随机浮点数。

更多函数详见官方文档

datetime库

1. 介绍

Python提供了有一个处理时间的标准函数库datetime,它提供了一些列由简单到复杂的时间处理方法。datetime库可以从系统中获得时间，并以用户选择的格式输出。

datetime库以类的方式提供多种日期和时间表达方式：

- `datetime.date`:日期表示类， 可以表示年月日等
- `datetime.time`:时间表示类，可以表示小时，分钟，秒，毫秒等
- `datetime.datetime`: 日期和时间表示类，功能覆盖`date`和`time`类
- `datetime.timedelta`: 时间间隔类

注意：这些类型的对象都是不可变的。

2.datetime类

`class datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None)¶`

year, *month* 和 *day* 参数是必须的。 *tzinfo* 可以是 None 或者是一个 tzinfo 子类的实例。其余的参数必须是在下面范围内的整数：

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <= 指定年月的天数`,
- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

如果参数不在这些范围内，则抛出 ValueError 异常。

```
from datetime import datetime
dt = datetime(2020,2,20,12,12,12) # 创建一个datetime对象
```

其他的构造方法，类方法：

- *classmethod* `datetime.now`(*tz=None*)¶

返回表示当地当前时间的 datetime 对象。

如果可选参数 *tz* 为 None 或未指定，这就类似于 today()，但该方法会在可能的情况下提供比通过 time.time() 时间戳所获时间值更高的精度（例如，在提供了 C gettimeofday() 函数的平台上就可以做到这一点）。

如果 *tz* 不为 None，它必须是 tzinfo 子类的一个实例，并且当前日期和时间将被转换到 *tz* 时区。

此函数可以替代 today() 和 utcnow()。

```
datetime.now()
```

```
对象属性（只读）
```

```
dt = datetime.now()
dt.year
dt.month
dt.day
dt.hour
dt.minute
dt.second
dt.microsecond
dt.tzinfo
```

支持的运算：

运算	结果
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 – timedelta`	(2)
`timedelta = datetime1 – datetime2`	(3)
`datetime1 < datetime2`	比较 `datetime` 与 `datetime`。(4)

1. datetime2 是从 datetime1 去掉了一段 timedelta 的结果，如果 `timedelta.days` > 0 则是在时间线上前进，如果 `timedelta.days` < 0 则是在时间线上后退。
2. 计算 datetime2 使得 datetime2 + timedelta == datetime1。
3. 从一个 `datetime` 减去一个 `datetime` 返回一个timedelta。
4. 当 *datetime1* 的时间在 *datetime2* 之前则认为 *datetime1* 小于 *datetime2*。

3. timedelta类

`timedelta`对象表示两个 date 或者 time 的时间间隔。

class `datetime.timedelta`(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

所有参数都是可选的并且默认为 `0`。 这些参数可以是整数或者浮点数，也可以是正数或者负数。

只有 *days*, *seconds* 和 *microseconds* 会存储在内部。 参数单位的换算规则如下：

- 1毫秒会转换成1000微秒。
- 1分钟会转换成60秒。
- 1小时会转换成3600秒。

- 1星期会转换成7天。

并且 days, seconds, microseconds 会经标准化处理以保证表达方式的唯一性，即：

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (一天的秒数)
- `-999999999 <= days <= 999999999`

下面的例子演示了如何对 *days*, *seconds* 和 *microseconds* 以外的任意参数执行“合并”操作并标准化为以上三个结果属性：

```
from datetime import timedelta
delta = timedelta(
     days=50,
     seconds=27,
     microseconds=10,
     milliseconds=29000,
     minutes=5,
     hours=8,
     weeks=2
)
# Only days, seconds, and microseconds remain
delta
```

4. 时间格式转换

时间的表现形式一般有三种：

1. 字符串

   ```
   '2020-02-20 12:12:12'
   '2020年2月20日12时12分12秒'
   ```

2. 整数时间戳

   距离格林威治时间`1970-01-01 00:00:00`的秒数

   ```
   1582016031
   ```

3. `datetime`对象

字符串转datetime对象

classmethod datetime.strptime(*date_string*, *format*)¶

返回一个对应于 *date_string*，根据 *format* 进行解析得到的 datetime 对象。

```
from datetime import datetime
time_string = '2020-02-20 10:10:10'
dt = datetime.strptime(time_string, '%Y-%m-%d %H:%M:%S')
```

datetime对象转字符串

datetime.strftime(*format*)¶

返回一个由显式格式字符串所指明的代表日期的字符串。

```
from datetime import datetime
dt = datetime.now()
dt.strftime('%Y-%m-%d %H:%M:%S')
```

格式化字符串规则见官方文档

时间戳转datetime对象

classmethod datetime.fromtimestamp(*timestamp*, *tz=None*)¶

返回对应于 POSIX 时间戳例如 time.time() 的返回值对应的datetime对象。

```
import time
from datetime import datetime
datetime.fromtimestamp(time.time())
```

datetime对象转时间戳

datetime.timestamp()

返回对应于 datetime 实例的 POSIX 时间戳。

from datetime import datetime
dt = datetime.now()
dt.timestamp()

更详细内容见[官方文档