Python基础知识

> Python是非常容易上手的，他的基础知识非常容易掌握。如果曾经学习过C++或者是Java的可以非常快的结束基础内容的学习。后文将会经常出现与C++的对比内容。

> Python的代码读起来会比较接近英文，大可以英文的阅读方式来理解Python。

> Python是一个基于面向对象的语言，面向对象将会是掌握Python基础的核心。这会花上一番精力。不用慌张，我们并不着急学习面向对象的知识。

> 这里真的只有基础知识，写在这里的基本都是最基础的使用方法。

1 基础语法

1.1 标识符/变量名

Python相比其他语言大大放宽了对变量的命名要求。你可以使用非ASCII字符作为变量名，你甚至可以使用中文作为变量名（但是我并不建议你这么做）。

- 第一个字符必须是字母表中字母或下划线_ 。
- 标识符的其他的部分由字母、数字和下划线组成。
- 标识符对大小写敏感。

```
# 以下是正确的命名方式
test
a # a与A是两个变量
A
b
q_2
__a__
变量

# 以下是错误的命名方式
123
1a2
a+b
?de
```

1.2 保留字/关键字

在C++中49个关键字（如if， for等）无法被用于标识符。

同样，在Python中也有保留字无法被用于标识符。

```
# 你可以使用如下代码来查看当前版本Python的保留字
import keyword
print(keyword.kwlist)
# ['False', 'None', 'True', '__peg_parser__', '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']
```

1.3 运算符

Python多数运算符和C++相同。

> 如果是初学者可以先大致浏览运算符这里，了解有哪些运算符，我们在后面会逐步学习运算符。
>
> 这里的内容主要是用来做参考。

算术运算符

运算符	描述
+	相加
-	相减
*	相乘
/	除
%	取模
**	幂
//	取整除（返回整数）

> 相比C++增加了**和//两个算术运算符
>
> ```
> 2 ** 3 # 8，2的3次幂
> 9 // 2 # 4，求商向下取整
> ```

取模运算符%永远是语言初学者很头疼的一个东西，但是这个东西相当有用。简单的讲，这是一个取余运算，用来计算两数相除的余数。

```
1 1 % 2 # 1，1除以2余1
2 2 % 2 # 0，2除以2余0
3
4 # 请根据以下例子总结取模运算的符号规律
5 5 % 2 # 1
6 5 % -2 # -1
7 -5 % 2 # 1
8 -5 % -2 # -1
```

比较运算符、

运算符	描述
==	相等
!=	不等于
>	大于
<	小于
>=	大于等于
<=	小于等于

比较运算符均会返回比较结果（结果为真则返回True，否则返回False）

```
1 a = (1 == 2) # a = False
2 b = (1 != 2) # b = True
3 c = 2
4 d = 3
5 e = (c > d) # e = False
```

赋值运算符

运算符	描述
=	赋值
+=	加法赋值（a += 1等价于a = a + 1）
-=	减法赋值（a -= 1等价于a = a - 1）
*=	乘法赋值（a *= 1等价于a = a * 1）
/=	除法赋值（a /= 1等价于a = a / 1）
**=	幂赋值（a **= 1等价于a = a ** 1）
//=	整除赋值（a //= 1等价于a = a // 1）

位运算符

运算符	描述
&	按位与
|	按位或
^	按位异或
~	按位取反
<<	左移动
>>	右移动

逻辑运算符

运算符	描述
and	布尔与
or	布尔或
not	布尔非

成员运算符

运算符	描述
in	在指定序列中找到值返回True，否则返回False
not in	在指定序列中没有找到值返回True，否则返回False

身份运算符

运算符	描述
is	判断两个标识符是否引用同一个对象
is not	判断两个标识符是不是引用自不同对象

1.4 运算符优先级（重要！）

和我们的四则运算逻辑相同，在计算机编程语言中运算符同样存在先后顺序

下表中运算符自上到下优先级依次降低

> 注意：（）的优先级最高，如果你确实无法明白你所写的式子的运算顺序，那就按照你需要的运算方式加括号
>
> ```
> # 例如
> (1+2)*3**2
> # 可以写为
> (1+2)*(3**2)
> # 当你不清楚你写的东西运算顺序是否正确的时候加括号就可以了
> ```

运算符	描述
**	指数 (最高优先级)
~，+，-	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
*，/，%，//	乘，除，求余数和取整除
+，-	加法减法
>>，<<	右移，左移运算符
&	位 'AND'
^，|	位运算符
<=，<，>，>=	比较运算符
==，!=	等于运算符
=，%=，/=，//=，-=，+=，*=，**=	赋值运算符
is，is not	身份运算符
in，not in	成员运算符
not，and，or	逻辑运算符

1.5 注释

在Python中单行注释使用#，多行注释使用'''或"""

> 注意：使用引号构成的多行注释实质是生成一个字符串

```
# 单行注释
# 这也是单行注释

'''多行

  注释'''

"""多行

  注释"""
```

1.6 缩进（重要！）

Python不像C++和Java那样使用{}来标识代码块。

Python使用缩进来区分代码块，同一代码块中的代码必须具有相同的缩进。

Python严格要求缩进规范，如果缩进出错会导致代码逻辑错误。

一般规定使用一个Tab（四个空格）来作为一次缩进。如果缩进距离不匹配会抛出 IndentationError

> 无法像C++或JS那样在代码排版上整活了(~￣▽￣)~

```python
# Python
a = 1
if a:
    print(2)
    if not a:
        print(3)
else:
    print(4)
```

上面这段Python代码等价与C++中

```cpp
// C++
a = 1;
if (a) {
    cout << 2 << endl;
    if (!a) {
        cout << 3 << endl;
    }
}
else {
    cout << 4 << endl;
}
```

1.7 单行语句

语句结尾不需要任何符号进行表示

1.8 多行语句

当单行语句过长时，可以使用\来实现多行语句

> 这个符号一般是用不上的。如果用上了那说明你这段代码可能需要进行优化。

```
simple = a + \
       b + \
       c
# 等价于
simple = a + b + c
```

需要注意，如果是()、[]、{}中的语句，可以直接换行而不需要加\进行标识。

（挖个坑，为什么会这样将会在后面讲到）

```
simple = [a,
          b,
          c]
# 等价于
simple = [a, b, c]
```

1.9 import

Python使用import和from...import来导入模块

如同其字面意思一样

import moduleName将导入整个模块

from moduleName import function从指定模块中导入指定内容

可以写为from moduleName import function1, function2, function3来导入多个内容

可以写为from moduleName import *来导入模块的全部内容

> import moduleName和from moduleName import *的区别体现在调用上。
>
> 例如模块中有个函数为function1。
>
> 那么import方式引入时要使用moduleName.function1来调用。
>
> from ... import *方式引入可以直接使用function1调用

> 请尽量避免使用from ... import *的方式来导入全部内容。
>
> 这么做可能导致同名函数被覆盖，无法得知自己调用的函数究竟是哪一个。

> 比如Python中原有函数round()，在你用from ... import *导入的包中同样包含了一个有着其他功能的round()函数，那么在后续的代码中这个round()就会存在歧义。

```
import os
os.listdir("C://") # 需要使用moduleName.function的方式来调用

import requests
a = requests.Session() # 使用moduleName.function的方式调用requests库中Session

from requests import Session
b = Session() # 可以直接调用Session

# a和b都可生成一个Session对象，效果相同
```

2 基本数据类型

Python的变量不需要声明。

但是所有变量在被使用前必须被赋值，只有被赋值了这个变量才会被创建。

Python是一个弱类型型语言，它不会显式的对数据类型进行区分，你也不必在意你创建的这个变量是什么类型的。Python将变量与其对应在内存中的数据类型分离开来。在Python中变量就是变量，变量只是指向内存中数据的一个符号。变量可以被指向任何一个数据，任何一种数据。

> 这里仍旧挖个坑，在后面将会详细说明这是如何实现的以及与C++的区别。

```
print(a) # 这个时候变量a并没有被创建，无法输出
        # 会抛出NameError: a is not defined

# 可以使用type()函数查看变量指向的数据类型
a = 1 # 变量a被创建，指向一个整型数据
type(a) # <class 'int'>

a = 1.2 # 重新指向一个浮点数据
type(a) # <class 'float'>

a = "string" # 重新指向一个字符串
type(a) # <class 'str'>
```

除了常规的单个变量赋值，在Python中也可以同时多个变量赋值。

```
a = b = c = 1
```

Python中还可以给多个对象指定多个变量。

> 请记好这种赋值方式，这种赋值方式在后面会隐式用到。
> 我会在元组详解这种赋值的实现原理。

```
a, b, c = 1, 2, 3
# a = 1
# b = 2
# c = 3
```

2.1 数字类型

python中数字有四种类型：整数、布尔型、浮点数和复数。

但是你并不需要刻意去记住他们，你只需要知道在Python中有这四种类型的数字即可。

> 注意：Python中的数字和C++中的完全不同。Python中的数字是没有溢出一说的。int和float仅仅是用于区分是否有小数部分。在Python中可以姑且认为单个变量可以记录的数字范围为$(-\infty, +\infty)$。

- int (整数), 如 1, 只有一种整数类型 int，表示为长整型。
- bool (布尔), 如 True。
- float (浮点数), 如 1.23、3E-2
- complex (复数), 如 1 + 2j、 1.1 + 2.2j

```
a = 1 # <class 'int'>
b = 1.2 # <class 'float'>
c = True # <class 'bool'>
d = 1 + 2j # <class 'complex'> 注意，复数中的实数部分和虚数部分均是float类型
```

除此之外，Python还可以使用十六进制和八进制表示整数

```
a = 0x00A # a = 10
b = 0o11 # b = 9
```

2.2 字符串(str)

在Python中你将经常与字符串打交道。但不用担心，Python的字符串十分友好。

在Python中，单引号与双引号的使用效果完全相同。在Python中是没有单个字符概念的，只有字符串。

如'G'是等价于"G"的。

> C++中单引号指单个字符，双引号指字符串

在Python中转义符仍为\。反斜杠可以用来转义，使用r可以让反斜杠不发生转义。 如r"this is a line with \n"则\n会显示，并不是换行。同样\\可以仍然可以输出\。

Python中的字符串可以使用+进行连接，用*进行重复连接

前文提到的'''和"""可以用于构成多行组成的字符串

```
a = 'abc'
b = "abc"
c = """abc
abc"""

a + b # abcabc
a * 3 # abcabcabc

# Python提供了一个len()函数用来获取序列的长度。当传入字符串时，将会返回字符串长度（字符数）
len(a) # 3
len(c) # 6
```

2.2.1 字符串访问

字符串可以使用[ ]进行索引。

$string[x]$这样来进行索引。

字符串中的每个值对应一个索引，在Python中有两种索引方式：从头索引、从尾索引

从头索引：下标从0开始，从前向后

从尾索引：下标从-1开始，从后向前

```
a = "abcdef"

# 给大家写下来方便对照
从尾索引:     -5   -4   -3   -2   -1
从头索引:    0  1  2  3  4
"abcdef":    a  b  c  d  e

a[0] # a
a[3] # d
a[-1] # e
a[-2] # d
```

> 尽管字符串是可以被索引的，但是字符串是不可以被局部更改的（即赋值操作）。
>
> ```
> a = "string"
> a[2] = "u"
> # 上面的索引赋值是不被允许的，Python解释器会报错
> # TypeError: 'str' object does not support item assignment
> ```

2.2.2 字符串截取

可以在索引上更进一步，不是索引其中的一个值，而是索引其中的一段，从而实现字符串的截取。

同样使用[ ]来进行索引截取

$variable[start:end:step]$，会截取下标范围为$[start, end)$的内容（请务必注意这个取值范围），$step$指明步长。

省略$start$的值代表从字符串的头开始。

省略end的值代表到字符串的尾结束。

省略: $step$部分则会去默认步长1。

这里你可以使用从头索引或者是从尾索引。但是请注意，可以二者混用。

让我们来举个栗子

```
a = "StringIsEasy"

# 这里给大家写出来方便对照
# 从尾索引: -12  -11  -10  -9   -8    -7   -6   -5   -4   -3   -2   -1
# 从头索引: 0  1  2  3  4   5    6  7  8  9   10   11
# a    :    S  t   r  i   n  g  I  s  E  a  s  y
```

```
7
8  a[1 : 2] # t
9  a[ : 3] # Str
10 a[1 : ] # tringIsEasy
11 a[-3 : -1] # as
12 a[-3 : ] # asy
13 a[1 : -1] # tringIsEas
14 a[0 : 4 : 2] # Sr
15 a[1 : 5 : 2] # ti
16 a[-5 : -1 : 2] # sa
```

2.2.3 Python中的转义字符

> 不需要全部记下，记下常用的即可。其他有需要可以直接去查文档。

转义字符	描述
\(在行尾时)	续行符【其实就是前面讲到的多行语句】
\\（常用）	反斜杠符号
\'（常用）	单引号
\"（常用）	双引号
\a	响铃
\b（常用）	退格（Backspace）
\000	空
\n（常用）	换行
\v	纵向制表符（非常不常用）
\t（常用）	横向制表符（Tab）
\r（常用）	回车，将 \r 后面的内容移到字符串开头，并逐一替换开头部分的字符，直至将 \r 后面的内容完全替换完成。
\f	换页
\yyy	八进制数，y 代表 0~7 的字符，例如：\012 代表换行。
\xyy	十六进制数，以 \x 开头，y 代表的字符，例如：\x0a 代表换行
\other	其它的字符以普通格式输出

这里讲到了对引号和单引号的转义。

在Python中有种写法可以让大家在字符串中直接写引号

如果你的字符串中包含单引号，使用双引号包裹字符串

如果包含双引号，使用单引号包裹字符串

如果两者都有，使用三引号包裹字符串（什么？你问三引号是啥？就是前面提到的"""和'''）

> 当然我不推荐这么做，我个人建议在写代码的时候统一使用一种引号包裹字符串，该转义的就转义，不要怕麻烦，良好的代码习惯很重要。

2.2.4 字符串格式化

和C++一样，Python同样支持字符串格式化。

在Python中有两种格式化字符串的方式。

> 注意：Python中的字符串格式化并不局限于输出，只要表达式正确任何时候都可以对字符串进行格式化

类C++格式化

> 这种格式化方式是最严格的，但是比较繁琐。是否使用可以根据个人喜好以及应用需求来决定。

这种字符串格式化的方式和C++的 printf() 用法是相同的

```
1  a = "String %s Easy %d"%("Is", 200)
2  # a = "String Is Easy 200"
```

我们称出现在字符串中的 %s 、 %d 为格式化符号，他们按照出现的顺序一一对应后面括号中的值。例如 %s 对应 "Is" ， %d 对应 200

不同的格式化符号对应不同类型的值。例如 %s 对应字符串， %d 对应整型数值。之后的表会详细说明每一种格式化符号对应的数据类型。

如果格式化符号与对应数据的数据类型不相同的话会抛出错误。

符 号	描述
%c	格式化字符及其ASCII码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f和%e的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

我们还有一些操作符用来辅助格式化符号

符号	功能
*	定义宽度
-	用做左对齐
+	在正数前面显示加号(+)
	在正数前面显示空格
#	在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X')
0	显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)
.n	n 是小数点后的位数(如果可用的话)

我们着重讲一下最后一种，这种是最常用的。

```
a = 1.23456789
"%f"%(a) # 1.2345689
"%.1f" # 1.2，这里.1指仅输出到小数点后一位
"%.5f" # 1.23456，输出到小数点后5位
"%.10f" # 1.2345678900，输出到小数点后10位，不存在的部分用0补齐
```

> 看到这里你肯定想问如果我想让字符串可以输出一个%怎么办。上面的操作符中存在一个%，你只需要书写%%即可让字符串在输出时有一个%，这点和转义符\相同（\\可以输出\）。

str.format()格式化方式

format()是字符串的一个方法。它可以被用于格式化字符串。它的灵活程度远高于前面提到的方法。

- 基础用法

我们只需要使用{ }在字符串中留下空位即可。这时候会按照出现的先后顺序依次将format中的值依次填入。

```
"{ } is { }".format("string", "easy") # "string is easy"
```

我们也可以自己确定填入的顺序，这时候只需要在{ }中加入索引即可。这个索引来自于format， format会为接收到的值从0开始依次编号。

```
"{0} is {1}".format("string", "easy") # "string is easy"
"{1} is {0}".format("string", "easy") # "easy is string"
# 而且这个时候可以让一个值在字符串中被使用多次
"{0} and {0} is {1}".format("string", "easy") # "string and string is easy"
"{1} and {0} is {1}".format("string", "easy") # "easy and string is easy"
```

我们甚至可以将我们的值参数化，让他们作为参数被填入字符串中。

```
# 直接构建参数
"{n} is {adj}".format(n="string", adj="easy") # "string is easy"

# 构建字典 -----> 有关字典的详细内容请见后面，这里不会详解，如果不懂可以先跳过
word = {
    "n" = "string",
    "adj" = "easy"
}
"{n} is {adj}".format(word) # "string is easy"

# 使用列表索引 -----> 有关列表的详细内容见后面，这里不会详解，如果不懂可以先跳过
word = ["string", "easy"]
"{ 0[0] } is { 0[1] }".format(word) # "string is easy"
```

既然我们可以使用参数化的值，我们同样可以为format传入对象。
(这会涉及到面向对象相关知识，可以先了解一下)

```
class Value:
    def __init__(self):
        self.value = 66666

test = Value()
"My brain is { 0.value }".format(test)
```

- 数字格式化

有关数字格式化，这点基本继承了前面讲到的类C++格式化中的数字格式化方式，但是需要将%替换为:

```
a = 1.23456789
"{ :.4f }".format(a) # 1.2345
```

详细格式如下

数字	格式	输出
3.1415926	{:.2f}	3.14
3.1415926	{:+.2f}	+3.14
-1	{:+.2f}	-1.00
2.71828	{:.0f}	3
5	{:0>2d}	05
5	{:x<4d}	5xxx
10	{:x<4d}	10xx
1000000	{:,}	1,000,000
0.25	{:.2%}	25.00%
1000000000	{:.2e}	1.00e+09
13	{:>10d}	13
13	{:<10d}	13
13	{:^10d}	13

其中 > 、 < 、 ^ 分别为右对齐、左对齐和居中对齐，对齐时所使用的填充字符根据其前面的符号决定。例如 {:x<4d} 会使用 x 进行填充。

2.2.5 字符串下常用方法

> 这里仅仅作为文档参考，看不看都行，用到的时候会详细讲。

序号	方法及描述
1	capitalize() 将字符串的第一个字符转换为大写
2	center(width, fillchar) 返回一个指定的宽度 width 居中的字符串，fillchar 为填充的字符，默认为空格。
3	count(str, beg= 0,end=len(string)) 返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
4	bytes.decode(encoding="utf-8", errors="strict") Python3 中没有 decode 方法，但我们可以使用 bytes 对象的 decode() 方法来解码给定的 bytes 对象，这个 bytes 对象可以由 str.encode() 来编码返回。
5	encode(encoding='UTF-8',errors='strict') 以 encoding 指定的编码格式编码字符串，如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
6	endswith(suffix, beg=0, end=len(string)) 检查字符串是否以 obj 结束，如果 beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True, 否则返回 False.
7	expandtabs(tabsize=8) 把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8 。
8	find(str, beg=0, end=len(string)) 检测 str 是否包含在字符串中，如果指定范围 beg 和 end ，则检查是否包含在指定范围内，如果包含返回开始的索引值，否则返回-1
9	index(str, beg=0, end=len(string)) 跟find()方法一样，只不过如果str不在字符串中会报一个异常。
10	isalnum() 如果字符串至少有一个字符并且所有字符都是字母或数字则返 回 True，否则返回 False
11	isalpha() 如果字符串至少有一个字符并且所有字符都是字母或中文字则返回 True, 否则返回 False
12	isdigit() 如果字符串只包含数字则返回 True 否则返回 False..
13	islower() 如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False
14	isnumeric() 如果字符串中只包含数字字符，则返回 True，否则返回 False
15	isspace() 如果字符串中只包含空白，则返回 True，否则返回 False.
16	istitle() 如果字符串是标题化的(见 title())则返回 True，否则返回 False

序号	方法及描述
17	isupper() 如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
18	join(seq) 以指定字符串作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
19	len(string) 返回字符串长度
20	[ljust(width, fillchar]) 返回一个原字符串左对齐,并使用 fillchar 填充至长度 width 的新字符串，fillchar 默认为空格。
21	lower() 转换字符串中所有大写字符为小写.
22	lstrip() 截掉字符串左边的空格或指定字符。
23	maketrans() 创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
24	max(str) 返回字符串 str 中最大的字母。
25	min(str) 返回字符串 str 中最小的字母。
26	[replace(old, new , max]) 把 将字符串中的 old 替换成 new,如果 max 指定，则替换不超过 max 次。
27	rfind(str, beg=0,end=len(string)) 类似于 find()函数，不过是从右边开始查找.
28	rindex(str, beg=0, end=len(string)) 类似于 index()，不过是从右边开始.
29	[rjust(width,, fillchar]) 返回一个原字符串右对齐,并使用fillchar(默认空格）填充至长度 width 的新字符串
30	rstrip() 删除字符串末尾的空格或指定字符。
31	split(str="", num=string.count(str)) 以 str 为分隔符截取字符串，如果 num 有指定值，则仅截取 num+1 个子字符串
32	[splitlines(keepends]) 按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。
33	startswith(substr, beg=0,end=len(string)) 检查字符串是否是以指定子字符串 substr 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查。
34	[strip(chars]) 在字符串上执行 lstrip()和 rstrip()

序号	方法及描述
35	swapcase() 将字符串中大写转换为小写，小写转换为大写
36	title() 返回"标题化"的字符串,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
37	translate(table, deletechars="") 根据 str 给出的表(包含 256 个字符)转换 string 的字符, 要过滤掉的字符放到 deletechars 参数中
38	upper() 转换字符串中的小写字母为大写
39	zfill (width) 返回长度为 width 的字符串，原字符串右对齐，前面填充0
40	isdecimal() 检查字符串是否只包含十进制字符，如果是返回 true，否则返回 false。

2.3 列表(list)

列表是一种有序序列。

> 这里的有序指内部元素有先后顺序（一般是添加的先后顺序。索引序号越小的顺序越靠前），而不是常规理解中的大小顺序等。

在使用Python的过程中，列表将会经常出现，你会十分依赖列表。

Python的列表属于广义表，它可以包含任何数据类型元素，它可以嵌套包含，它的长度是动态的。

Python的列表使用[ ]来包裹，在列表内使用,来分割元素。

Python的列表同样可以进行索引，使用[ ]进行索引和截取（这点十分类似我们前面所讲的字符串）。

列表同样支持从头索引和从尾索引。而且可以二者混用。

$List[pos]$

$List[start:end:step]$

```
a = ["a", 1, "t", 1.2, [1, 2]]

# 这里构造了一个列表
从尾索引:     -5   -4   -3    -2     -1
从头索引:    0   1   2   3    4
   a   :   [   "a"     1    "t"   1.2   [1,2]    ]

```

```
# 索引会返回对应元素
a[1] # 1
a[2] # "t"
a[4] # [1, 2]

# 截取的话将会返回列表
a[ : 2] # ["a", 1]
a[-3 : ] # ["t", 1.2, [1, 2]]
a[1 : -1] # [1, "t", 1.2]

# 与字符串不同的是，列表是可以进行索引赋值实现部分内容更改的
a[2] = 2.5 # a = ["a", 1, 2.5, 1.2, [1, 2]]

# 在字符串时我们提到的len()函数，同样可以用在列表上用来获取列表中的元素个数
len(a) # 5
len(a[4]) # 2，这时候是获取a中索引为4的元素的长度，这个元素是列表[1,2]，它的长度是2
```

同样的，使用 + 可以实现两个列表的连接； * 可以实现列表的重复连接。

```
a = [1, 2, 3]
b = [4, 5, 6]
a + b # [1, 2, 3, 4, 5, 6]
a * 2 # [1, 2, 3, 1, 2, 3]
```

前面我们提到列表是动态的，我们可以使用列表的 append() 方法和 pop() 方法来增删元素。

```
# list.append(Element)
a = [1, 2, 3]
a.append("a") # a = [1, 2, 3, "a"]
# list.pop( position=-1 )
# pop()方法的默认会返回最后一个元素并在列表中删除，可以传入索引来删除特定位置的元素
a.pop() # a = [1, 2. 3]
a.pop(0) # a = [2, 3]
```

同样，我们也在这里列出列表的常用方法。

序号	方法
1	list.append(obj) 在列表末尾添加新的对象
2	list.count(obj) 统计某个元素在列表中出现的次数
3	list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
4	list.index(obj) 从列表中找出某个值第一个匹配项的索引位置
5	list.insert(index, obj) 将对象插入列表指定位置
6	[list.pop(index=-1]) 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
7	list.remove(obj) 移除列表中某个值的第一个匹配项
8	list.reverse() 反向列表中元素
9	list.sort(key=None, reverse=False) 对原列表进行排序
10	list.clear() 清空列表
11	list.copy() 复制列表

2.4 元组(tuple)

元组使用( )进行包裹，使用,分割元素。元组也是广义表，元组的几乎所有的特性均与列表相同，但是元组一旦被创建，其元素就不可以被改变，这点和字符串很像。

```
# 注意，创建元组时，如果元组的元素只有一个，也要加,
# 这点与列表不同
# 如果不加,会被认为是运算
t = (1) # <class "int">
t = (1, ) # <class "tuple">

a = ("a", 1, "t", 1.2, [1, 2])

# 这里构造了一个列表
从尾索引:     -5    -4    -3    -2     -1
从头索引:    0   1   2   3   4
   a   :  (  "a"    1   "t"  1.2  [1,2]   )

# 索引会返回对应元素
a[1] # 1
a[2] # "t"
```

```
a[4] # [1, 2]

# 截取的话将会返回列表
a[ : 2] # ()"a", 1)
a[-3 : ] # ()"t", 1.2, [1, 2])
a[1 : -1] # (1, "t", 1.2)

# 在字符串时我们提到的len()函数，同样可以用在列表上用来获取列表中的元素个数
len(a) # 5
len(a[4]) # 2，这时候是获取a中索引为4的元素的长度，这个元素是列表[1,2]，它的长度是2
```

我们仍然可以使用 + 来连接两个元组来构成一个新元组，使用 * 来重复连接一个元组

```
a = (1, 2, 3)
b = (4, 5, 6)

a + b # (1, 2, 3, 4, 5, 6)
a * 2 # (1, 2 ,3, 1, 2, 3)
```

填坑了填坑了o(*≧▽≦)ツ┏┓，记不记得在前面基本数据类型讲赋值的时候在多对象赋值的时候有a, b, c = 1, 2, 3这种赋值方式。

实现原理：

1, 2, 3首先形成元组(1, 2, 3)。然后将提供的变量与元素按照顺序一一对应进行赋值。最后得到a = 1, b = 2, c = 3。

如果这里仅仅是a = 1, 2, 3，那么a将会得到上面提到的元组。

列表也是有这个特性的， a, b, c = [1, 2, 3]可以得到相同的结果。

2.5 字典(dict)

字典也是一个有序序列。

字典使用{ }包裹，使用,来分割元素。与列表和元组不同的是，这里的每个元素都是一个键值对。

> Python的字典格式完全照搬json数据格式，如果你了解json数据格式的话这里可以快速浏览一下。

2.5.1 键值对

键值对由键(key)和值(value)组成，通过:进行连接。

一般形式为$key:value$。

2.5.2 组成字典

一个字典由若干组键值对组成。值(value)可以为任何数据类型，任何数据内容。但是键(key)的数据内容必须是唯一的，而且一般约定键的数据类型为字符串。

> 虽然不限定键的数据类型（必须是可进行hash的数据类型），但是在调用过程中会变得非常离谱。通常为了代码的可读性并不会使用字符串以外的数据类型作为键。
>
> ```
> a = {
> (123, 111) : "test"
> }
> print(a[(123,111)])
>
> # 运行结果
> # test
> ```

```
a = {
    "str" : "string", # 一个键值对
    "num" : 1000,
    "list" : [1, 2],
    "tuple" : (1, 2),
    "dict" : {
        "num" : 10000 # 注意：这里重新构成了一个字典，所以这个键值对与上面那个键也为num的键值对不冲突
    }
}

# 这里进行一个错误举例
b = {
    "str" : "string",
    "str" : "test", # 错误，键已经存在
}
```

2.5.3 字典的索引与增删

字典可以索引，但是无法截取。

字典的索引和前述的方式并不相同，虽然它仍然使用[ ]进行索引，但它使用键作为索引的值。

```
a = {
    "str" : "string", # 一个键值对
    "num" : 1000,
    "list" : [1, 2],
    "tuple" : (1, 2),
    "dict" : {
        "num" : 10000 # 注意：这里重新构成了一个字典，所以这个键值对与上面那个键也为num的键值对不冲突
    }
}

a["str"] # "string"
a["list"] # [1, 2]

# 如果你索引了一个并不存在的键并试图获取它那薛定谔的值的时候，那么Python会抛出异常
a["undefined"] # KeyError: 'undefined'

# 同样可以len()函数来获取字典的长度，但是我也不是很清楚你这么做有啥实际意义emmmm
len(a) # 5
```

字典可以动态的增删元素，修改键值对的值

```
a = {
    "str" : "string", # 一个键值对
    "num" : 1000,
    "list" : [1, 2],
    "tuple" : (1, 2),
    "dict" : {
        "num" : 10000 # 注意：这里重新构成了一个字典，所以这个键值对与上面那个键也为num的键值对不冲突
    }
}

# 当你对已有的键进行索引并进行赋值时，即修改了这个键值对的值
a["str"] = "not string" # 原有的键值对"str":"string"变为"str":"not string"
```

```
13
14 # 当你对不存在的键进行索引并进行赋值时，即新建了这个键值对
15 a["new"] = "create" # 在字典中新增键值对"new":"create"
16
17 # 注意字典不能使用+进行连接，也不能使用*进行重复连接
```

2.5.5 字典常用方法

序号	函数及描述
1	radiansdict.clear() 删除字典内所有元素
2	radiansdict.copy() 返回一个字典的浅复制
3	radiansdict.fromkeys() 创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值
4	radiansdict.get(key, default=None) 返回指定键的值，如果键不在字典中返回 default 设置的默认值
5	key in dict 如果键在字典dict里返回true，否则返回false
6	radiansdict.items() 以列表返回一个视图对象
7	radiansdict.keys() 返回一个视图对象
8	radiansdict.setdefault(key, default=None) 和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default
9	radiansdict.update(dict2) 把字典dict2的键/值对更新到dict里
10	radiansdict.values() 返回一个视图对象
11	[pop(key,default]) 删除字典给定键 key 所对应的值，返回值为被删除的值。key值必须给出。 否则，返回default值。
12	popitem() 随机返回并删除字典中的最后一对键和值。

2.6 集合

> 集合用的并不多。但是因为它的特性的原因，在某些特定环境下有很大的用途。

集合使用{ }或set()来创建，创建空集合必须使用set()而不能用{ }，{ }用于创建空字典。
请务必区分字典和集合。

> 集合的元素可以是任何数据，字典必须是键值对

集合是一个无序的不重复序列。

集合通常可以用于去重判断。

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
# {'orange', 'banana', 'pear', 'apple'}，这是basket实际的内容，重复的元素仅会被保留一个

# 在很久之前的1.3节的成员运算符中我们提到过in和not in运算符就可以用在这里（不限于这里）
# in用来判断一个元素是否存在于当前序列中
# not in 用来判断一个元素是否不存在于当前序列中
"apple" in basket # True
"peach" in basket # False

# 同样可以使用len()函数获取集合的长度
len(basket) # 4
```

使用add()方法为集合添加元素（仍然会进行去重操作）

使用remove()方法从集合中移除元素（如果元素不存在会抛出异常）

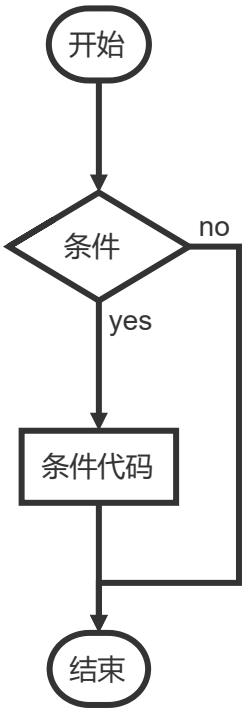
集合的常用方法

方法	描述
add()	为集合添加元素
clear()	移除集合中的所有元素
copy()	拷贝一个集合
difference()	返回多个集合的差集
difference_update()	移除集合中的元素，该元素在指定的集合也存在。
discard()	删除集合中指定的元素
intersection()	返回集合的交集
intersection_update()	返回集合的交集。
isdisjoint()	判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。
issubset()	判断指定集合是否为该方法参数集合的子集。
issuperset()	判断该方法的参数集合是否为指定集合的子集
pop()	随机移除元素
remove()	移除指定元素
symmetric_difference()	返回两个集合中不重复的元素集合。
symmetric_difference_update()	移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。
union()	返回两个集合的并集
update()	给集合添加元素

3 条件控制与循环

3.1 条件控制

条件控制通常用来实现下面这个流程图。



在Python中仅提供了 if 来实现条件控制

> 在Python中是没有 switch 的

3.1.1 if语句

Python中的 if 语句块如下所示

```
if condition1 :
    <statement>
elif condition2 :
    <statement>
else :
    <statement>
```

其中 elif 和 else 是可选的。

> 注意：通过 : 来表明后面将会跟随条件满足时要执行的代码块。必须严格满足缩进。

这里给出一段示例：

```
a = 1
b = 2

if a > b :
    print("a>b")
elif b > a :
    print("b>a")
else :
    print("b==a")
# 最后的结果为："b>a"
```

如果你学过C++的话，你应该记得C++有且只有一个三元运算符：... ? ... : ...。它的功能和条件判断十分的相似。如果你仔细看过前面1.3节的话，你应该会发现在Python中并没有三元运算符。这么好用的东西Python给删除了吗?

并没有，在Python中，原有的三元运算符被替换成了if的一种特殊形式（个人认为阅读起来更加符合正常语序了）。

```
a = 1 if 1 < 2 else 2 # 如果1 < 2那么a = 1否则a = 2

# 和C++的三元运算符一样，这样简写的条件判断同样允许嵌套，建议为了方便理解最好使用括号区分层次
a = 1 if  1 < 2 else (2 if 2 < 3 else 3)
```

3.1.2 常用判断操作符

我们在1.3节中讲到的比较运算符和逻辑运算符会是这里的常客，成员运算符也经常会出现

可以给大家举点栗子

```
a = 1
b = 2
c = 3
d = 1
e = [1, 2, 3, 4, 5]

if (a > b) and (a == d) : # 当a大于b且a等于d时进入代码块
    print("a")
elif (b > a) and (a == c) :
    print("b")
elif c in e: # 如果c在序列e中，进入代码块
    print("c")
else :
```

```
    print("d")
# 运行结果："c"
```

这里并没有说明逻辑运算符or。不管你有没有学过C++我都要说一嘴。

进行逻辑判断时是在遵循括号顺序的前提下从左向右依次执行。

比如(1 < 2) or (1 == 2)，那么会先判断1 < 2，再判断1 == 2。

从逻辑与与逻辑或的真值表上我们可以知道，只有全为真逻辑与才为真，只有全为假逻辑或才为假。

那么对于解释器来说，只要逻辑与碰到一个假即可判定整个条件为假，只要逻辑或碰到一个真即可判定整个条件为真。这样做可以加快执行速度。

简单的说，只要我们碰上一个可以完全确定整个式子真值的条件时，后续的条件判断均不会被执行。

用上面的那个例子，我们已经判断出(1 < 2)为真了，对于逻辑与来说整体已经为真了，这个时候后面第二个条件就不会被执行而直接返回真。

下面这个例子可以更好的说明这个特性。

```
a = [1, 2]
b = 2
if (a[0] < b) or (a.pop()) :
 print("success")
print(a) # [1, 2]
```

毫无疑问一定会输出success的，那么a的值呢?

如果一步不落的执行的话，a应该仅有一个元素1。但是当我们真实去运行程序的时候我们会发现，a中仍有两个元素，也是就说， a.pop()这句话并没有被执行。因为第一句(a[0] < b)已经可以确定整个式子的真值为真，不在需要后续的判断了。

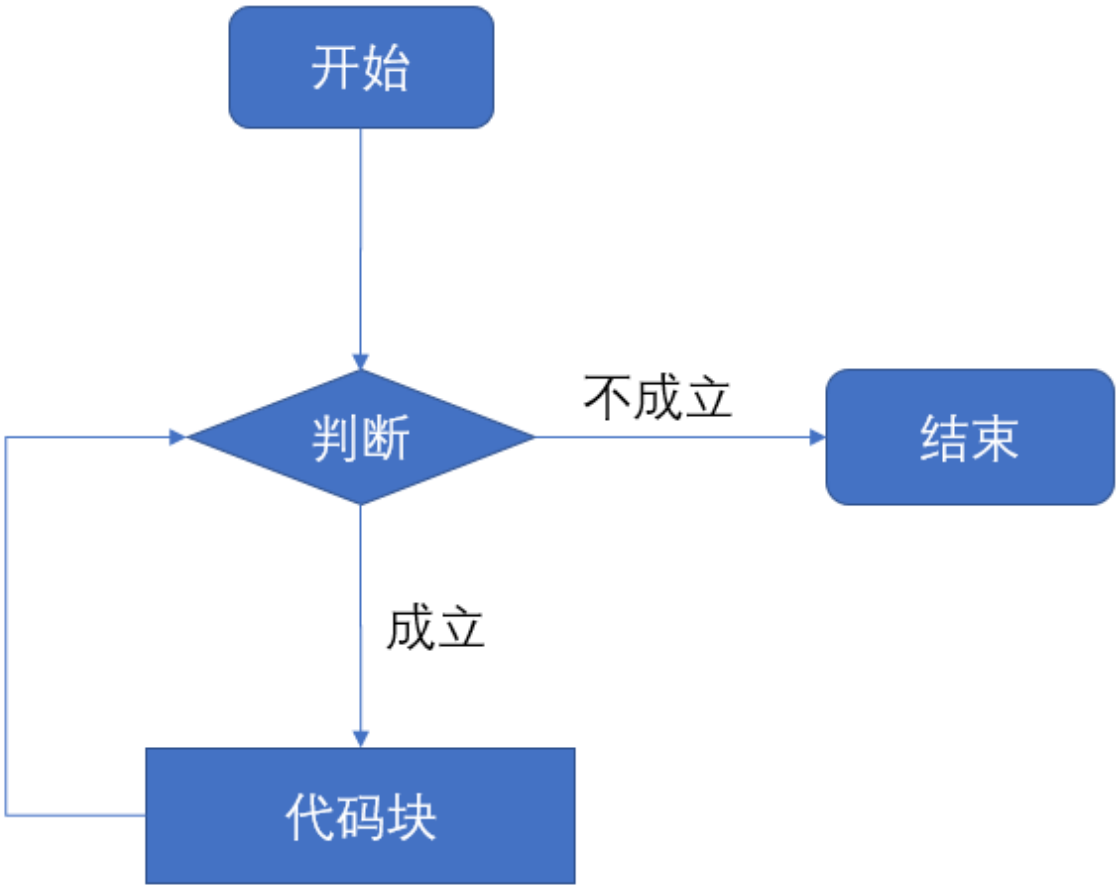
3.2 循环

循环，顾名思义，重复进行相同的操作。在Python中有两大循环语句： while和for

3.2.1 while

Python中的while和C++中的while用法相近。如果你已经掌握了C++中while的用法，那么大可以直接去看后面的示例

在Python中， while的流程如下图所示



先给出一段示例代码

```
a = 1
b = 2

# while部分开始
while a < b : # 当a < b时条件成立进入while代码块
    a += 1 # a = a + 1
# while部分结束

print(a) # 与while有相同缩进的下一个语句
```

当碰到while语句时，Python会对后面的判断语句进行判断。这时候a=1， b=2， a<b成立，进入while后跟随的代码块（这个代码块和前面if的代码块一样，严格要求相比他自己的while有一次缩进）。依次执行代码块中的内容，这时候a的值变为2。当代码块执行结束后，会重新回到while语句进行判断。这时候a<b不成立， while循环结束。执行与while有着相同缩进的下一个语句。

为了防止你们没看懂我上面啰啰嗦嗦说的一堆，我来举个很简单的例子。

让我们来计算一下从1加到100，即$1+2+3+...+99+100$。这是个使用循环的典型例子。

```python
a = 1
b = 0

# 我们令a从1开始自增至100
# 让b从0开始每次均与自增之前的a相加，相当于得到一个式子0 + 1 + 2 + 3 + ... + 99 + 100，即我们要达到的目标
# 那么我们就可以使用循环让a从1开始，每次循环自己加一并这样循环100次即可。只需要在a > 100成立时，即a < 101这个条件不成立时停止循环即可。

while a < 101 : # 只要a < 101成立，即a还在[1, 100]这个范围时进入while对应的代码块
    b += a # b = b + a
    a += 1 # a = a + 1

print(b) # b = 5050
```

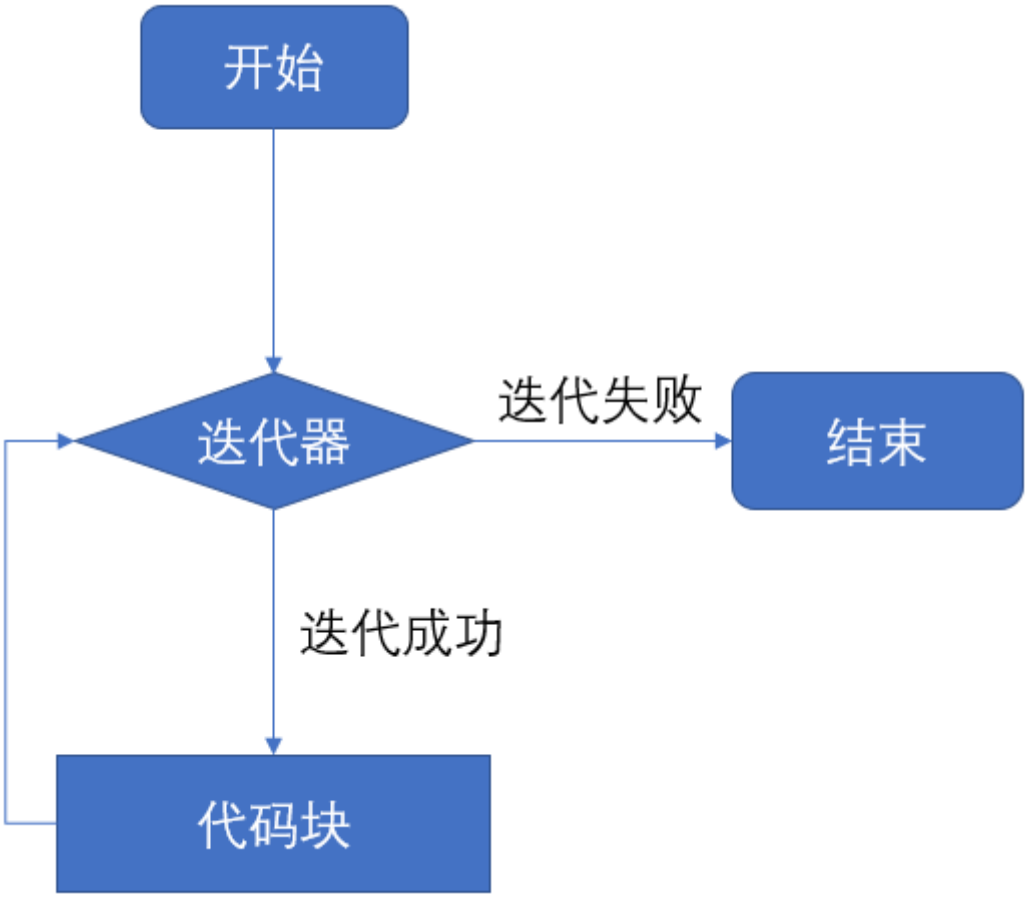
3.2.2 for

Python中的for算是独树一帜，和C++中的for完全不一样。Python中的for更容易被使用英语语序阅读。

下面给出for的流程图。和while相比，唯一的不同就是判断被换成迭代器了。

在Python中for通常搭配in进行使用，即for .... in ...。注意，这里的in并不是成员运算符。

> 迭代器这个概念这里不会详解，后面会专门有章节给出详解。大家不必深究迭代器是什么东西。
>
> 我保证就算这里你不知道迭代器是个啥你也能学会（因为我就是这样的(*￣3￣)╭）



在Python中while是基于判断进行的循环，而for是基于迭代进行循环。

我们先举一个小小的例子来看看这个基于迭代循环究竟是什么。

```
a = ["apple", "banana", "pear", "peach"]

for fruit in a:
    print(fruit)

print("示例结束了~")
# 这段代码的输出结果是
# apple
# banana
# pear
# peach
# 示例结束了~
```

你有没有发现啥?

如果没有我们可以再来一个例子

```
b = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

for weekday in b :
```

```
    print(weekday)

print("示例结束了~")

# 代码运行结果
# Mon
# Tue
# Wed
# Thu
# Fri
# Sat
# Sun
# 示例结束了~
```

通过上面两个示例我们可以发现， for x in y这个循环类型实质是按照有序序列y的从头索引的元素顺序依次遍历了每个元素，每次循环讲当前遍历到的元素赋值给x。

比如我们第一个例子。第一次循环我们遍历到序列第零个元素"apple"，赋值给变量fruit， print()函数输出apple；第二次循环遍历到序列第一个元素"banana"，赋值给变量fruit， print()输出banana。依次类推，继续依次输出pear和peach。第二个例子同理。

> 我们可以利用这种方式实现列表、元组、字典等有序序列的遍历。
> 集合也是可以使用for .. in ..进行遍历的，但是由于集合本身是无序的，每次遍历得到的元素的顺序是不同的。

> 这里只是为了方便理解使用有序序列代替了迭代器。注意：迭代器 ≠ 有序序列

同样我们可以使用for实现固定次数的循环（实际应用环境中不是很常用）

在Python中我们有一个range(n)函数，将range()函数放在y的位置可以实现从0到n-1的遍历

```
for i in range(5) : # 遍历范围为[0, 5)
    print(i)

# 0
# 1
# 2
# 3
# 4
```

当然我们也可以不从0开始，这时候我们要传入start和end两个参数

```
for i in range(2, 5) : # 遍历范围为[2, 5)
    print(i)

# 2
# 3
# 4
```

3.2.3 循环嵌套

和前文中的if一样，循环也是可以进行嵌套的。

> for和while可以进行混合嵌套，请不用担心。

```
for i in range(5) : # 第一层循环
    for j in range (6) : # 第二层循环
        print("i = {}, j = {}".format(i, j))
```

3.2.3 break和continue

> 如果你学过C++，那么你可以跳过这个章节。

- break

break用来跳出当前这一层循环。

这句话并不好理解。咱可以举个栗子。

```
for i in range(100) :
    print(i)
    if i == 10 :
        break

# 0
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
# 10
```

这段程序的意思是从0到100的遍历，每次输出当前遍历的到值，当遍历的值为10时我们执行break操作。

从输出上我们可以看到，程序并没有像正常的for循环那样一直输出到这个循环的最后（即输出到99）。他在遍历到10的时候就结束了。

这就是break的作用，它可以直接结束当前这一层的循环，不论这层循环是否完成。

```
for i in range(3) :
    for j in range(10) :
        if j == 5:
            break
        for k in range(10) :
            print("i = {}, j = {}, k = {}".format(i, j, k))
```

通过运行上面这段程序可以发现在输出中，不论i为多少，j永远不会超过5。也就是说break每次都仅仅是结束了它所在的那层循环，并不会影响外层循环。而内层循环也会被一并结束。

- continue

continue的使用方法和break很类似。不同的是continue仅仅会跳过这层循环的当前这次循环，而不是结束这层循环。

```
for i in range(3) :
    for j in range(10) :
        if j == 5:
            continue
        for k in range(10) :
            print("i = {}, j = {}, k = {}".format(i, j, k))
```

让我们复用前面那段代码，并把其中的break改成continue。这次运行我们会发现，仅仅会缺失j=5的情况。也就是只要continue执行，这层循环中continue之后的代码都会被跳过，直接回到循环的判断/迭代器部分进行下一次循环。

4 函数

> 如果可以理解什么是函数可以直接跳过前面这段函数的解释
>
> 相信你肯定学过高数/工数/数分，下面这段将基于高数中的函数部分进行展开
>
> 这里用离散数学中的知识来理解会更好。

在高数中我们会这样定义一个一元函数：
对于集合$\{x_i\}$和$\{y_i\}$，存在一个从x到y的映射关系$f(x)$，使对每一个x_i，都有唯一的y_i与之对应。

由上面的一元函数定义我们可以推出多元函数$f(x_1, x_2, x_3, \ldots, x_n)$的定义。

在Python中，函数也是这么一个意思。我们知道我们在数学中常用的函数$f(x_1, x_2, x_3, \ldots, x_n)$是一个有着特定运算功能的模块，在Python中的函数也是一个具有一定特定运算功能的模块。

例如我们之前使用print()和range()还有len()都是函数。他们都有输入（x_i），也会有输出（y_i），他们都是可以完成特定功能的模块。

4.1 定义一个函数

在Python中我们使用def来定义一个函数，按照如下格式进行定义。

> 和前面一样，严格要求对应的功能代码块相比def缩进一次。
>
> 在Python的函数中不需要指明返回值类型，这和变量不需要声明类型是一个道理

```
def functionName(args, kwords) :
    <function part>
    <function part>
```

举个例子。

```
# 输出传入的内容
def test(arg1) :
    print(arg1)

# 我们只需要像调用print()函数那样调用我们自己定义的函数即可
test("test")

# 输出：test
```

4.2 函数参数

4.2.1 参数

我们称传入的变量为参数。我们在声明函数的时候就可以声明函数的参数。Python不限制函数参数的个数。

```
def test(arg1, arg2, arg3) :
    print("arg1:{}, arg2:{}, arg3:{}".format(arg1, arg2, arg3))
```

在调用时我们需要按照顺序依次传入参数。这非常的关键，这将决定函数得到的参数的值是否正确。

Python提供了一种可以指定传入的方式。用参数名=变量名 来指定传入参数。如我们调用上面那个我们写的自定义函数。

> 尽管可以指定传入，我仍然建议大家在写的时候尽量按照函数定义的顺序传入。

```
year = "2021"
month = "07"
day = "20"

# 正常调用
test(year, month, day)
# 输出：arg1:2021, arg2:07, arg3:20

# 指定传入
test(arg1=year, arg2=month, arg3=day)
# 输出：arg1:2021, arg2:07, arg3:20
```

4.2.2 带有默认值的参数

可以为参数指明默认值。只需要在定义这个参数的时候赋值为所需要的默认值即可。当在传入参数的时候没有为这个参数传入值的话就会使用默认值。

```
def test(arg1, arg2, arg3="这是默认值") :
    print("arg1:{}, arg2:{}, arg3:{}".format(arg1, arg2, arg3))

year = "2021"
month = "07"
day = "20"

# 正常调用
test(year, month, day)
# 输出：arg1:2021, arg2:07, arg3:20

# 使用默认值
test(year, month)
# 输出：arg1:2021, arg2:07, arg3:这里是默认值
```

请注意：带有默认值的参数必须放在正常参数的后面，例如上面的arg3，在定义时它是不能被写在arg1和arg2的前面的，必须写在他们的后面。

在Python中为了方便函数调用我们一般规定函数在传入时对于正常的参数直接写变量名传入即可，对于有默认值的参数采用指定的方式传入。

```
year = "2021"
month = "07"
day = "20"

# 这里使用带默认值的那个函数
test(year, month, arg3=day)
```

因为参数带有默认值意味着我们不一定会用到这个参数，我们也不一定会用到所有的带有默认值的参数。如果不指定的话可能会导致传参错误。

```
def test(arg1, arg2, arg3="默认值", arg4="默认值") :
 print("arg1:{}, arg2:{}, arg3:{}, arg4:{}".format(arg1, arg2, arg3, arg4))

year = "2021"
month = "07"
day = "20"

test(year, month, day)
# 输出：arg1:2021, arg2:07, arg3:20, arg4:默认值

test(year, month, arg4=day)
# 输出：arg1:2021, arg2:07, arg3:默认值, arg4:20
```

像上面这个例子中的，因为参数默认是按照顺序传入，如果我们想使用arg4但是不使用arg3，那么使用前一种传参方式会使得参数传入错误。

4.3 return

return一般用法为： return variable。用于退出函数并返回变量的值。
其中variable部分是可选的。这意味你可以单纯的退出函数而不返回任何值。

```
def returnTest() :
    print("before return")
    return
    print("after return")

returnTest()

# 运行结果
# before return
```

return可以在任何位置作为函数出口来退出函数，在逻辑上位于return之后的所有语句均不会被执行（函数已经退出了）

在Python中，你可以返回一组变量。使用return variable_1, variable_2, ...的方式来返回一组变量。注意，这时候最好使用等同数量的变量来接收返回值。

```
def test():
    a = 1
    b = 2
    c = 3
    return a

def testMult():
    a = 1
    b = 2
    c = 3
    return a, b, c

single = test()
a, b, c = testMult()
mult = testMult()

print(single)
print(a)
print(b)
print(c)
print(mult)
print(type(mult))
```

运行结果为

```
1
1
2
3
(1, 2, 3)
<class tuple>
```

看到元组的时候是不是就明白这个多返回值是如何实现的了（滑稽）

4.4 迭代器

在C中并没有迭代器的概念，但是在面向对象的编程语言中。迭代器是非常重要的一个部分。

与Java相比，Python的迭代器在使用起来是非常简单的。迭代器是访问集合元素的一种方式。之前我们使用`for .. in ..`方式实现列表、元组、字典的遍历就是通过迭代器实现的。

迭代器相关方法`iter()`和`next()`

使用`iter()`来生成一个迭代器，可以传入列表、元组、字典。
使用`next()`来获取迭代器的下一个元素。

这里我用一张图和下面这段代码来解释迭代器到底是啥。

```python
a = [1, 2, 3]
t = iter(a)
print(next(t))
print(next(t))

# 运行结果
# 1
# 2
```

!(iter)[images\iter.jpg]

迭代器在遍历到末尾后，如果在调用`next()`进行迭代，会抛出`StopIteration`。可以使用异常捕获来实现全遍历。（有关异常捕获的详细内容后续会有详解）

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

while True:
    try:
        print(next(t))
    except StopIteration:
        break
```

Python已经为我们提供来for .. in ..来使用迭代器进行遍历了。前面讲for .. in ..的时候就已经强调在Python中for将依赖迭代器进行循环操作。

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

# 使用迭代器
for i in t:
    print(i)

# 不使用迭代器
for i in a:
    print(i)

# 上面两种方式进行序列元素遍历结果是相同的。可以向for提供有序序列让它生成迭代器进行遍历，也可以是我们自己生成迭代器提供给for进行遍历。
```

5 模块与包

5.1 模块

至此为止，我们一直将代码写在一个.py文件中。

在Python中，我们将一个.py文件称为一个模块。可以使用import来导入这个模块中的类、函数、变量，同时其主程序代码将会运行。。

如我们有个Test.py文件，其内容如下

```
def test() :
    print("导入成功")

# 主程序代码块
a = 1
b = 2
c = b - a
# 主程序代码块
```

我们用如下代码进行测试

```
import Test

Test.test()
print(Test.a)
print(Test.b)
print(Test.c)

# 运行结果:
# 导入成功
# 1
# 2
# 1
```

5.2 包

当我们有一组可以完成特定任务的模块时，可以将其集中起来构成一个包。这样可以极大简化调用和管理。

比如我们有一组可以用于进行网络通信的模块。其中Connection相关的模块在Connection文件夹下， DataStream相关的模块在DataStream文件夹下。可以构建一个包名为network的包。

下面是文件结构树

```
network                 顶层包
    __init__.py         初始化包
    Connection          Connection子包
        __init__.py
        ...
    DataStream          DataStream子包
        __init__.py
        ...
    ...
```

__init__.py文件帮助解释器确定当前目录为一个包。如果一个包内没有子包，那么__init__.py留空即可，如果包内存在子包，需要在__init__.py中创建一个名为__all__的列表，保存所有子包的名称。

比如上面示例中，network下的__init__.py文件内容为

```
__all__ = ["Connection", "DataStream"]
```

其余的留空即可。

> 这样可以帮助解释器精确区分顶层包内的模块与子包，确保在from .. import *时候可以正确导入子包。