第一章 基础知识

1.1 第一个程序

1.创建并进入该文件夹

```
fray_hao$ mkdir -p basic-tutor/basic
fray_hao$ cd basic-tutor/basic
```

2.创建python源文件：

```
fray_hao$ touch 01-HelloPython.py
```

3.用vim打开该文件：

```
fray_hao$ vim 01-HelloPython.py
```

输入以下内容：

```
print("Hello Python")
```

最后保存文件。

4.在shell中通过python解释器执行源代码

```
fray_hao$ python 01-HelloPython.py
Hello Python
```

1.2 Python执行的三种方式

1.2.1 解释器执行

用python解释器执行源代码：

```
# 使用 python 2.x 解释器
python xxx.py

# 使用 python 3.x 解释器
python3 xxx.py
```

Python 的解释器 如今有多个语言的实现，包括：

- CPython —— 官方版本的 C 语言实现
- Jython —— 可以运行在 Java 平台
- IronPython —— 可以运行在 .NET 和 Mono 平台
- PyPy —— Python 实现的，支持 JIT 即时编译

1.2.2 交互式运行

(1). python shell

直接在终端中运行解释器进入交互shell，在shell中直接输入 Python 的代码，会立即看到程序执行结果

```
fray_hao$ python

>>> print("Hello World")
Hello World
```

缺点： 代码无法保存，不适合运行太大的程序

退出python shell：

```
exit()
```

(2). ipython

IPython 中 的 “I” 代表 交互 interactive,是一个 python 的 交互式 shell，比默认的 python shell 好用得多

- 支持自动补全 : 按tab键
- 自动缩进
- 支持 bash shell 命令
- 内置了许多很有用的功能和函数

```
fray_hao$ ipython


In [1]: print("hello World")
hello World
```

1.2..3 集成开发环境（IDE）

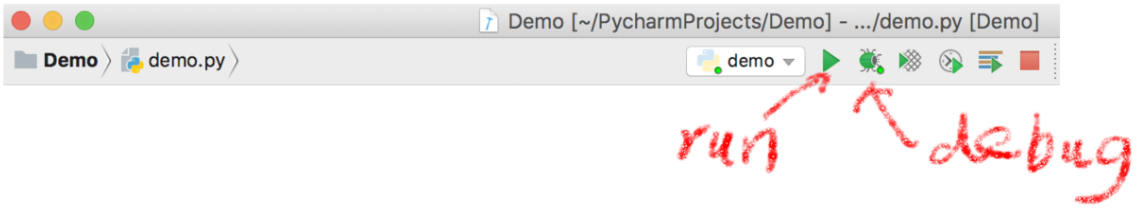
集成开发环境（IDE，Integrated Development Environment）—— 集成了开发软件需要的所有工具，一般包括以下工具：

- 图形用户界面
- 代码编辑器（支持 代码补全 / 自动缩进）

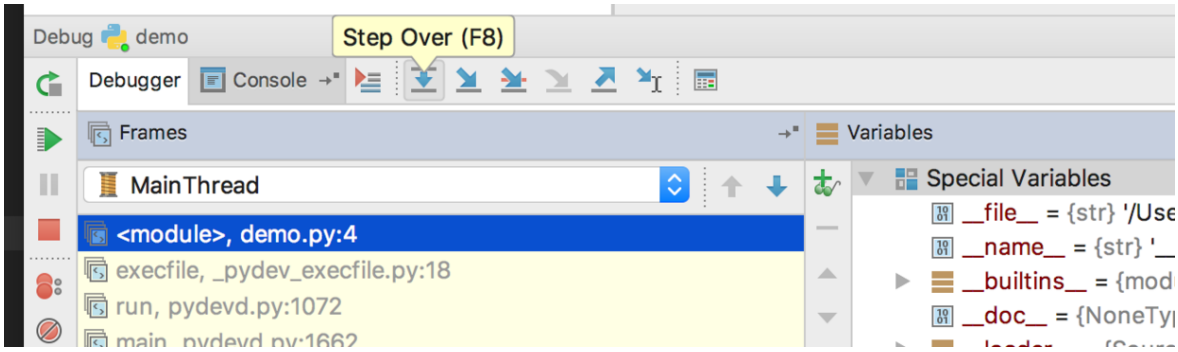
- 编译器 / 解释器
- 调试器（断点 / 单步执行）

PyCharm 是 Python 的一款非常优秀的集成开发环境.PyCharm 除了具有一般 IDE 所必备功能外，还可以在 Windows、Linux、macOS 下使用,适合开发大型项目。

1. 在文件编辑区域编写代码
2. 右上角的 工具栏 能够 执行(SHIFT + F10) / 调试(SHIFT + F9) 代码



3. 通过控制台上方的单步执行按钮(F8)，可以单步执行代码



1.3 注释

注释可以增强代码的可读性

1.3.1 单行注释

以 # 开头，# 右边的所有东西都被当做说明文字，而不是真正要执行的程序，只起到辅助说明作用

```
# 这是第一个单行注释
print("hello python")
```

单行注释可以写到代码后面:

```
print("Hello python")  # 这是一个单行注释
```

> 为了代码的规范性，根据PEP8，代码和注释之间需要至少两个空格

1.3.2 多行注释（块注释）

如果编写的注释信息很多，一行无法显示，就可以使用多行注释。

要在 Python 程序中使用多行注释，可以用一对连续的三个引号(单引号和双引号都可以)

```
'''
这是多行注释
可以写很多的信息
'''
print("Hello python")
```

1.3.3 什么时候需要使用注释?

1. 注释不是越多越好，对于一目了然的代码，不需要添加注释
2. 对于 复杂的操作，应该在操作开始前写上若干行注释
3. 对于 不是一目了然的代码，应在其行尾添加注释（为了提高可读性，注释应该至少离开代码 2 个空格）
4. 绝不要描述代码，假设阅读代码的人比你更懂 Python，他只是不知道你的代码要做什么

> 在一些正规的开发团队，通常会有 代码审核 的惯例，就是一个团队中彼此阅读对方的代码

1.4 关于代码规范

Python 官方提供有一系列 PEP（Python Enhancement Proposals） 文档。- 其中第 8 篇文档专门针对 Python 的代码格式 给出了建议，也就是俗称的 PEP 8。

- 文档地址：https://www.python.org/dev/peps/pep-0008/
- 谷歌有对应的中文文档：http://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/

第二章 面向过程编程

2.1 变量和常量

程序就是用来处理数据的，而变量就是用来存储数据的，对应于计算机内存中的一块区域。变量通过唯一的标识符（即变量名）来表示，并且可以通过各种运算符对变量的值进行操作。

2.1.1 变量的定义

Python是一种动态类型的编程语言，不需要显示的声明变量的数据类型，可以直接对变量进行赋值，变量赋值以后该变量才会被创建。

赋值语句的一般语法格式如下：

```
变量名 = 表达式
```

等号(=)为赋值运算符，用来给变量赋值：

- 左边为变量名
- 右边为表达式

2.1.2 常量

常量是指首次赋值后保持固定不变的值。在Python中没有常量机制，通常用一个不改变值的变量来表示常量，比如用下面的方式表示一个常量π：

```
PI = 3.14
```

2.1.3 变量命名规则

(1). 标识符

标示符就是程序员定义的 变量名、函数名。标识符的约束规则：

- 标示符可以由 字母、下划线和数字组成
- 不能以数字开头
- 不能与关键字重名

思考：下面的标示符哪些是正确的，哪些不正确，为什么？

```
fromNo12
from#12
my_Boolean
my-Boolean
Obj2
```

```
2ndObj
myInt
My_tExt
_test
test!32
haha(da)tt
jack_rose
jack&rose
GUI
G.U.I
```

(2). 关键字

关键字就是在Python内部已经使用的标识符。关键字具有特殊的功能和含义，开发者 不允许定义和关键字相同的名字的标示符 通过以下命令可以查看Python中的关键字：

```
In [29]: import keyword

In [30]: print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', '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']
```

(3). 变量命名规则

命名规则可以被视为一种惯例，并无绝对与强制，目的是为了增加代码的识别和可读性。常见的规则如下：

- 标识符区分大小写。Hero 与hero是不一样的
- 在定义变量时，为了保证代码格式，=号的左右应该各留一个空格
- 如果变量名需要由二个或多个单词组成时，可以按照以下方式命名
    - 每个单词都使用小写字母。例如，firstname
    - 单词与单词之间使用下划线连接。例如，first_name
    - 小驼峰式命名法，即驼峰命名。如，firstName
    - 大驼峰命名法，即帕斯卡（Pascal）命名如，FirstName

(4). 命名规范建议

- **模块**： 模块尽量使用小写命名，首字母保持小写，尽量不要用下划线(除非多个单词，且数量不多的情况)，并且将相关的类和顶级函数放在同一个模块里. 不像Java, 没必要限制一个类一个模块.

```
# 正确的模块名
import decoder
import html_parser

# 不推荐的模块名
import Decoder
```

- **类名**：类名使用帕斯卡命名风格，首字母大写，私有类可用一个下划线开头

```
class Farm():
    pass

class AnimalFarm(Farm):
    pass

class _PrivateFarm(Farm):
    pass
```

- **函数**：函数名一律小写，如有多个单词，用下划线隔开

```
class Person():
    def run():
    pass

    def run_with_env():
    pass
    def _private_func(): # 私有函数在函数前加一个或两个下划线
        pass
```

- **变量名**：变量名尽量小写, 如有多个单词，用下划线隔开

```
if __name__ == '__main__':
    count = 0
    school_name = '
```

- **常量**：常量采用全大写，如有多个单词，使用下划线隔开

```
MAX_CLIENT = 100
MAX_CONNECTION = 1000
CONNECTION_TIMEOUT = 600
```

- **引号**:

  简单说，自然语言使用双引号，机器标示使用单引号，因此 **代码里** 多数应该使用 **单引号**

- **自然语言 使用双引号** `"..."`
  例如错误信息；很多情况还是 unicode，使用`u"你好世界"`
- **机器标识 使用单引号** `'...'`
  例如 dict 里的 key
- **正则表达式 使用原生的双引号** `r"..."`
- **文档字符串 (docstring) 使用三个双引号** `"""......"""`

2.2 基本数据类型

数据类型在数据结构中的定义是一个值的集合以及定义在这个值集上的一组操作。在python中可以通过type()函数查看数据类型：

```
a =3
print(type(a)) # <class 'int'>
```

2.2.1 数值类型

数值类型用于表示数据，数值类型的数据可以参与算术运算。数值类型包括了整型、浮点型和复数型

(1) 整型(int)

整型数据即整数，没有小数部分，但可以有正负号。 在Python内部对整数的处理分为普通整数和长整数，普通整数长度为机器位长，通常都是32位，超过这个范围的整数就自动当长整数处理，而长整数在计算机内的表示不是固定长度的，在内存容量允许的情况下，长整数的取值范围几乎没有限制，这为大数据处理带来了便利。

```
 a = 3
 b = -3
```

在python中，整数常量可以用十进制、二进制、八进制和十六进制来表示：

```
In [25]: d2  = 0b1010 # 0b或0B做为前缀，表示后面的值为二进制

In [26]: print(d2)
10

In [27]: d8 = 0o12 # 0o或0O做为前缀,表示后面的值为八进制

In [28]: print(d8)
10

In [29]: d10 = 10 # 数据没有前缀，默认为十进制

In [30]: print(d10)
10
```

```
In [34]: d16 = 0xA # 0x或0X做为前缀，表示后面的数据为十六进制

In [35]: print(d16)
10
```

(2) 浮点型（float）

浮点数表示一个实数。对于浮点数，Python3.x默认提供17位有效数字的精度。 浮点型数据有两种表示方式:

- 十进制小数形式
- 指数形式

```
In [36]: f1 = 3.14159265

In [37]: f2 = 3.14e8  # 3.14 × 10^8
```

(3) 复数型(complex)

复数的表示形式为a+bj, 其中a为复数的实部，b为复数的虚部，j为虚数单位（表示-1的平方根），字母j也可以写成大写形式J。

```
In [40]: complex =3+4j

In [42]: print(complex.real) # real属性获取实数部分
3.0

In [43]: print(complex.imag) # imags属性获取虚数部分
4.0
```

2.2.2 字符串类型

字符串是使用单引号、双引号和三引号（三个单引号或三个双引号）扩起来的任意文本

```
name1 = '我是python' # 单引号

name2 = "我是python" # 双引号

name3 = """我是python""" # 三引号

name4 = '''我是python''' # 三引号
```

单引号和双引号括起来的字符只能是单行的，如果代码中显示为多行需要添加上反斜杠():

```
name = '我' \
        "是" \
        'python'
print(name) # 我是python
```

三引号常用于定义文档字符串。类似于html中的pre标签的效果，可以将多行文字以及空格原样输出。

```
code = '''
    int main(){
        #代码
    }
'''

'''
输出结果:
    int main(){
        #代码
    }
'''
```

2.2.3 布尔类型

布尔类型数据用于描述逻辑判断的结果。布尔类型的数据只有两个值：

- 逻辑真：True
- 逻辑假：False

```
In [1]:  a = 3>2

In [2]: print(a)
True
```

2.2.4 空值

在python中，空值用None来表示

```
In [3]: a = None

In [4]: print(type(a))
<class 'NoneType'>
```

2.3 运算符

2.3.1 算术运算符

算法运算符可以用于对操作数进行算术运算，其运算结果是数值类型

运算符	描述	实例
+	加或单目运算中的正号	10 + 20 = 30
-	减或单目运算中的负号	10 - 20 = -10
*	乘	10 * 20 = 200
/	除	10 / 20 = 0.5
//	取整除	返回除法的整数部分（商） 9 // 2 输出结果 4
%	取余数	返回除法的余数 9 % 2 = 1
**	幂	又称次方、乘方，2 ** 3 = 8

2.3.2 关系运算符

关系运算符也叫比较运算符，用来比较两个操作数的大小，其运算结果是一个布尔值。关系运算符的操作数可以是数字或字符串。若操作数是字符串，系统会从左到右逐个比较每个字符的Unicode码，直到出现不同的字符或字符串为止。 | 运算符| 描述 | 实例 | | :---: | :---: | --- | |等于|23返回False；“abc”="ABC"返回False <|小于|2<5 返回True;"this"<"This"返回False

> |大于|2>5 返回False;"this">"This"返回True <=|小于等于| 2<=5返回True;"this"<="This"返回False =|大于等于|5>=5 返回True;"this">="This"返回True !=|不等于 |3!=5返回True;"this"!="This"返回True

2.3.3 赋值运算符

|运算符| 描述| 实例 | :---: | :---: | --- | |= |简单的赋值运算符| c = a + b 将 a + b 的运算结果赋值为 c| |+= |加法赋值运算符| c += a 等效于 c = c + a |-= |减法赋值运算符| c -= a 等效于 c = c - a |*= |乘法赋值运算符| c *= a 等效于 c = c * a |/= |除法赋值运算符| c /= a 等效于 c = c / a |%= |取模赋值运算符| c %= a 等效于 c = c % a |**=|幂赋值运算符| c** = a 等效于 c = c ** a |//=| 取整除赋值运算符| c //= a 等效于 c = c // a

2.3.4 逻辑运算符

逻辑运算符用于布尔值的运算，包括逻辑与、逻辑或、逻辑非。其中逻辑与、逻辑或是双目运算符、逻辑非是单目运算符

逻辑与的真值表：

	T	**F**
T	T	F
F	F	F

逻辑或的真值表：

	T	**F**
T	T	T
F	T	F

2.3.5 位运算符

位运算用于对数字的二进制位进行运算。

| 运算符| 描述 | 实例 | | :---: | --- | --- | << | 左移运算符（将左操作数的二进制数位全部左移若干（右操作数）位，高位丢弃球，低位补零）|2<<3 返回16。相当于`$2^3$`

> | 右移运算符（将左操作数的二进制数位全部右移若干（右操作数）位，高位补0，低位丢弃）|16>>3返回2.相当于`$2^{1/3}$` &|按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0|22&3返回2 ||按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。|32|3返回35 |按位异或运算符：当两对应的二进位相异时，结果为1|^{18}6返回20 ~|按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1 。~x 类似于 -x-1|~32返回-33

2.3.6 成员运算符

成员运算符用于判定对象是否存在于字符串等序列中。 |运算符| 描述 |实例 | :---: | --- | --- | |in |如果在指定的序列中找到值返回 True，否则返回 False。|"y" in "python" 返回True |not in| 如果在指定的序列中没有找到值返回 True，否则返回 False。|"y" not in "python" 返回False

2.3.7 身份运算符

身份运算符用于比较两个对象的内存地址是否相同。

|运算符| 描述| 说明 | | :---: | --- | --- |---| |is| is 是判断两个标识符是不是引用自一个对象 |x is y, 类似 id(x) == id(y) , 如果引用的是同一个对象则返回 True，否则返回 False| x=1;y=x; |is not| is not 是判断两个标识符是不是引用自不同对象 |x is not y ， 类似 id(a) != id(b)。如果引用的不是同一个对象则返回结果 True，否则返回 False。

例如：

```
a = 20
b = 20
print(a is b) # True
b = 30
print(a is b) # False
print(a is not b) # True
```

2.3.8 表达式与优先级

表达式是运算符和操作数组成的有意义的组合，它可以是常量、变量，也可以是函数的返回值。通过运算符对表达式中的值进行若干次运算，最终得到表达式的返回值。

按照运算符的种类，可以将表达式分为算术表达式、关系表达式、逻辑表达式等。多种运算符混合运算可形成复合表达式，此时系统会按照运算符的优先级和结合性依次进行运算。如果需要，用户可以使用圆括号来改变运算顺序。

常见的运算符按照优先级从高到低的顺序排列如下：

优先级（从高到底）	运算符
1	索引([])
2	幂(**)
3	单目(+、-)
4	*，/， //，%
5	加减（+，-）
6	关系（>，>=，<，<=，!=）is，身份运算符(is not)
7	赋值（=，+=，-=，/=，//=，%=,**=）
8	逻辑非(not)
9	逻辑与(and)
10	逻辑或（or）

2.4.数值类型的转换

2.4.1 其他数据类型转换为整数

```
In [1]: integer = int(3.14) # 浮点型转换为整型
```

```
In [2]: integer2 = int("3",10) # 字符串转换为整型.字符串中有数字(0-9)和正负号(+/-)以外的字符，就会报错。

In [3]: integer3 = int(True) # 布尔型转换为字符串

In [4]: print(integer)
3

In [5]: print(integer2)
3

In [6]: print(integer3)
1
```

> 注意：空值是无法转换为整数类型的。

2.4.2 其他数据类型转换为浮点型

```
In [9]: floatl = float(3) # int 转换为 float 时，会自动给添加一位小数。

In [11]: float2= float('-3.14') # 如果字符串含有正负号(+/-)、数字(0-9)和小数点(.)以外的字符，则不支持转换。

In [12]: float3 = float(True)

In [17]: print(floatl)
3.0

In [18]: print(float2)
-3.14

In [19]: print(float3)
1.0
```

2.4.3 任意对象转换为字符串

1. 基本数据类转换为字符串

```
In [20]: str1 = str(3) # int 转换 str 会直接完全转换

In [21]: str2 = str(-12.00) #loat 转换 str 会去除末位为 0 的小数部分。

In [22]: str3 = str(True) # 转换为字符串
```

```
In [24]: str4 = str(complex(12+9j)) #先将值转化为标准的 complex 表达式，然后再转换为字符串。

In [25]: print(str1)
3

In [26]: print(str2)
-12.0

In [27]: print(str3)
True

In [28]: print(str4)
(12+9j)
```

2.5 流程控制

计算机程序主要由数据结构和算法两个要素组成。数据结构即数据的存储形式，算法则是对操作步骤的描述。任何简单或复杂的算法都可以由顺序结构、选择结构和循环结构组合而成。通过这三种结构就可以实现程序的流程控制。

2.5.1 顺序结构

程序的工作流程一般分为输入数据、处理数据、输出结果。顺序结构是一种最简单的流程控制结构，其特点是程序中的各个操作是按照它们在源代码中的排列顺序执行的。

(1). 数据输入

为了让用户通过程序与计算机进行交互，程序通常应具有数据的输入\输出功能。在Python程序中，可以使用键盘输入数据，也可以从文件中或数据库中读取数据，然后由程序对输入的数据进行处理，处理结果可以输出到屏幕上，也可以保存到文件或数据库中。

实际应用中，最常见的情形是从键盘输入数据并通过屏幕输出数据，这是标准的控制台输入\输出模式。

在Python中，标准输入通过内置函数input实现，其格式如下：

```
variable = input("提示字符串")
```

例如，输入姓名，并用变量捕获：

```
In [31]: name = input('请输入姓名:')
请输入姓名:hao

In [32]: print(name)
hao
```

(2). 数据的处理

为了对输入的数据进行处理，首先需要将数据保存到变量所引用的内存中，这个要通过赋值语句来实现。

在Python中，赋值语句分为：

- 简单赋值语句
- 复合赋值语句
- 多变量赋值语句

1) 简单赋值语句

简单赋值语句用于对单个变量的赋值，其一般格式为：

```
变量 = 表达式
```

例如：

```
name = input('请输入姓名:')
```

当键盘中输入字符，回车时，会将输入的字符以字符串的形式保存内name变量中。

2) 复合赋值语句

复合赋值语句是利用复合赋值运算符对变量当前值进行某种运算后执行赋值操作的。变量既是运算对象又是赋值对象。

例如，求输入值的平方：

```
In [34]: num *= float(input('输入值: '))
输入值: 12

In [35]: num  # 在交互模式下，直接使用变量即可以输出值
Out[35]: 144.0
```

3) 多变量赋值语句

在python中，可以使用赋值语句的变化形式对多个变量进行赋值。赋值语句有两种变化形式：

- 链式赋值语句。用于对多个变量赋予同一个值，语法格式：
    - 变量1=变量2=...=变量n = 表达式
- 同步赋值语句。使用不同表达式的值分别对不同变量赋值,语法格式：
    - 变量1,变量2,...,变量n = 表达式1，表达式2,...,表达式n

链式赋值案例：

```
In [36]: x = y = z = 123

In [37]: x
Out[37]: 123

In [38]: y
Out[38]: 123

In [39]: z
Out[39]: 123
```

同步赋值语句案例：

```
In [41]: x,y,z = 3,4,5

In [42]: x
Out[42]: 3

In [43]: y
Out[43]: 4

In [44]: z
Out[44]: 5
```

4) 数据的处理

获取数据后，就可以进行下一步的处理了。我们可以对数据进行运算、排序、查找等各种操作，这里不再赘述。

(3). 数据的输出

1) 标准输出

标准输出可以通过两种方式实现：

- 在交互模式下，使用表达式语句来输出表达式的值。
- 在脚本模式下，使用print函数输出

print函数的语法结构：

```
print(output1[,output2,...][,sep=分隔符][,end=结束符])
```

例如：

```
In [51]: name,age = 'zs',18

In [52]: print(name,age,sep='-',end=';')
zs-18;
```

2) 格式化输出

使用字符串格式化运算符%可以将输出格式化，然后调用print函数，按照一定格式输出数据，具体调用格式如下：

```
print(格式化字符串%(输出项))
```

其中格式化字符串由普通字符串和格式说明符组成。普通字符按原样输出，格式说明符则用于指定对应输出项的输出格式。

常用的格式说明符：

格式说明符	含义
%%	输出百分号
%d	输出十进制整数
%c	输出字符
%s	输出字符串
%O	输出八进制整数
%x或%X	输出十六进制整数
%e或%E	以科学计数法输出浮点数
%[w][.p]f	以小数形式输出浮点数，数据长度为w，默认为0；小数部分有p为，默认有6位。

例如：

```
In [53]: print("姓名：%s,年龄：%d"%('zs',18))
姓名：zs,年龄：18
```

还可以用格式化辅助指令：

符号	功能
m.n	m是显示的最小总宽度，n是小数位数
-	用于左对齐
+	在正数前面显示加号
0	显示的数字前面填充'0'取代空格
#	在输出的八进制数前添加0o，在输出的十六进制前添加0x

例如：

```
In [54]: print('%10.4f'%12.34) #要求最终返回的数字的宽度是10位，小数后4位小数。默认右对齐
   12.3400

In [55]: print('%-10.4f'%12.34) # 左对齐
12.3400

In [57]: print('%-+10.4f'%12.34) # 显示正号
+12.3400

In [61]: print('%010.4f'%12.34) # 右对齐，左边以0填充
00012.3400
In [62]: print("%#x"%11) # 在十六进制数前加0x
0xb
In [63]: print("%#o"%11) # 在八进制数前加0o
0o13
```

除了print提供的格式化输出外，还可以使用str.format函数实现字符串的格式化。语法结构如下：

```
str.format(输出项)
```

其中，str（格式化字符串）由普通字符与格式说明符组成。普通字符按原样输出，格式说明符则用于指定对应输出项的输出格式。格式说明符用花括号括起来，其一般形式如下：

```
{[序号或键名]:格式控制符}
```

> 序号：可选性。用于指定要格式化的输出项的位置，0表示第一项。若序号全部省略，则按自然顺序输出。\ 键名：可选性。一个标识符，对应于输出项的名字或字典的键值\

常见的格式控制符：

符号	说明
d	输出十进制整数
b	输出二进制整数
o	输出八进制整数
x	输出十六进制整数
c	输出整数对应的ascii字符
f	输出浮点数
e	输出科学计数法形式的浮点数
%	输出百分号

```
In [64]: print("{0:c}".format(97)) # 输出数字对应的ascii字符
a

In [65]: print("{0:d},{1:b},{2:o},{3:x}".format(10,10,10,10)) # 以不同进制输出
10
10,1010,12,a
In [67]: print("{:d},{:b},{:o},{:x}".format(10,10,10,10)) # 省略序号
10,1010,12,a
In [68]: print("{0:#d},{1:#b},{2:#o},{3:#x}".format(10,10,10,10)) # 可以使用
格式化辅助指令
10,0b1010,0o12,0xa

In [74]: print("{:08.4f}".format(3.14)) # 输出浮点数
003.1400
```

综合案例：编写一个简单的成绩录入程序

```
In [97]: name,gender,classes = eval(input("请输入姓名，性别与班级"))
请输入姓名，性别与班级"zs","male",18

In [98]: chinese,math,eng = eval(input("请输入语文，数学与英文成绩"))
请输入语文，数学与英文成绩89,80,76

In [100]: print("姓名：{:},性别：{:},班级：{:}".format(name,gender,classes))
姓名：zs,性别：male,班级：18

In [101]: print("语文：{:},数学：{:},英文：{:}".format(chinese,math,eng))
语文：8,数学：80,英文：76
```

2.5.2 选择结构

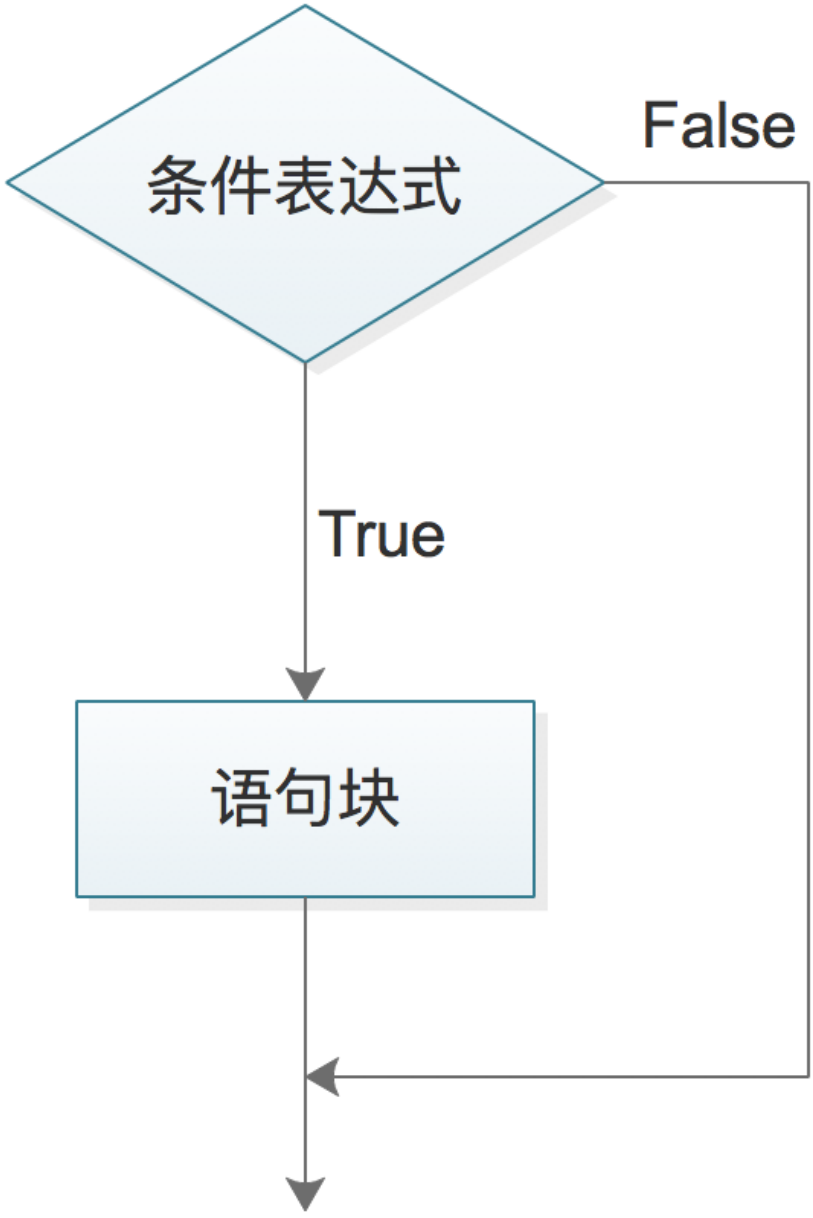
选择结构是指程序在运行时根据某个特定条件选择一个分支执行。根据分支的多少，分为选择结构分为单分支选择结构、双分支选择结构、多分支选择结构

(1). 单分支选择结构

用于处理单个条件、单个分支的情况。语法结构如下：

```
if 条件表达式 ：
    语句块
```

if语句的执行流程如下图：



例题：判断是否闰年

> 闰年的必要条件：1. 能被4整除2。2.不能被100整除。世纪闰年中能被400整除

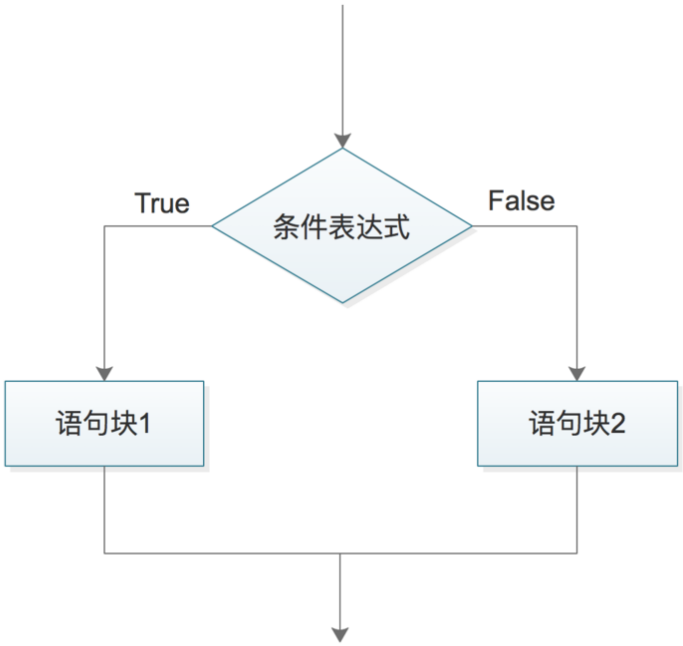
```
leapYear = '不是'

year = int(input('输入一个年份：'))

if year%4 == 0 and year%100 != 0 or year%400 == 0 :
    leapYear ="是"

print("{0}年{1}闰年。".format(year,leapYear))
```

(2). 双分支选择结构

用于处理单个条件、多个分支的情况。语法结构如下：

```
if 表达式：
    语句块1
else：
    语句块2
```

if-else语句的执行流程：



例题：从键盘输入三角形的三条边，计算三角形的面积

> 构成三角形的充要条件：任意两边之和大于第三边。

```
a,b,c = eval(input("输入三条边"))

if a+b>c and a+c>b and b+c>a:
    p=(a+b+c)/2 # 半周长
    area = (p*(p-a)*(p-b)*3,4,(p-c))**0.5 # 海伦公式
    print("当三角形的三条边为a={},b={},c={}时：".format(a,b,c))
    print("三角形的面积：area={}".format(area))
else:
    print("a={},b={},c={}不构成三角形".format(a,b,c))
```

例题：输入一个整数，判断它是不是水仙花数

> 水仙花数是自幂数的一种，严格来说3位数的3次幂数才称为水仙花数。它的每个位上的数字的3次幂之和等于它本身。

if... else 还能作为条件运算。条件运算是一个三目运算，它有三个运算对象，其一般格式如下：

```
表达式1 if 逻辑表达式 else 表达式2
```

> 首先计算逻辑表达式的值，如果为True，则计算表达式1的值，否则计算表达式2的值。

例如： 比较两个数的大小

```
a,b = eval(input("输入两个数。两个数用逗号分隔："))

result = "{}>{}".format(a,b) if a>b else "{}<{}".format(a,b) # 三目运算的结果赋值给result

print(result)
```

例题：判断输入的数字是否是水仙花数

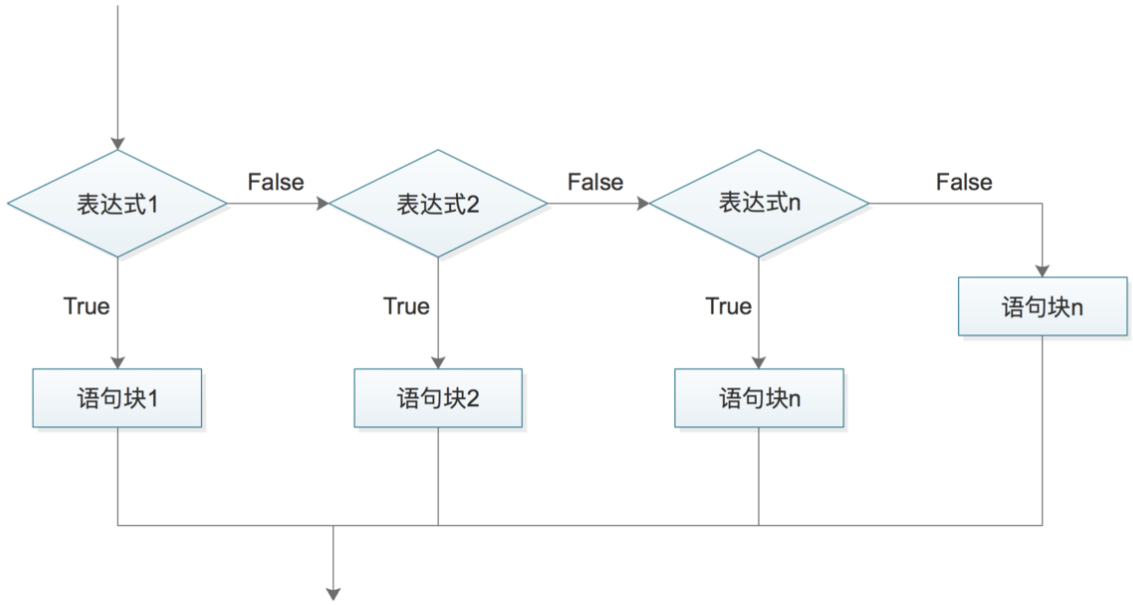
```


num = int(input("输入一个三位数："))

unit = num % 10  # 个位数

ten = num // 10 % 10  # 十位数

hundred = num // 100  # 百位数

result = "是" if unit ** 3 + ten ** 3 + hundred ** 3 == num else "不是"

print("{}{}水仙花数".format(num, result))
```

(3). 多分支选择结构

多分支结构用于处理多个条件，多个分支的情况。其一般格式如下：

```
    if 表达式1:
        语句块1
    elif 表达式2:
        语句块2
    elif 表达式n:
        语句块n
    else:
        缺省语句块
```

if-elif-else语句的执行流程：



例题：输入成绩，判断等级

> 85-100为优秀；70~84为良好；60-69为及格；60以下不及格

```
score = float(input("请输入学生的成绩："))

if score >= 85:
    grade = "优秀"
elif score >= 70:
    grade = "良好"
elif score >= 60:
    grade = "及格"
else:
    grade = "不及格"
print("百分制成绩：{},成绩等级：{}".format(score, grade))
```

> 从上面的例题可知。if语句中定义的变量在当前的作用域中都可用。

(4). 嵌套选择

例题：编写一个登陆程序

> 首先对输入的用户名进行验证。如果存在，再对输入的密码进行验证。

```
# 模拟用户名与密码
USER = "admin"
PWD = "888"

# 从键盘输入用户名
userName = input("请输入用户名：")

# 验证用户名
if userName == USER:
    pwd = input("输入密码：")
    # 验证密码
    if pwd == PWD:
        print("登陆成功！")
        print("欢迎{}".format(userName))
    else:
        print("密码不正确")
else:
    print("用户名不存在")
```

2.5.3 循环结构

循环结构是控制一个语句块重复执行的程序结构。它由两部分构成：

- 循环体：重复执行的语句块
- 循环条件：控制是否继续执行循环体的表达式

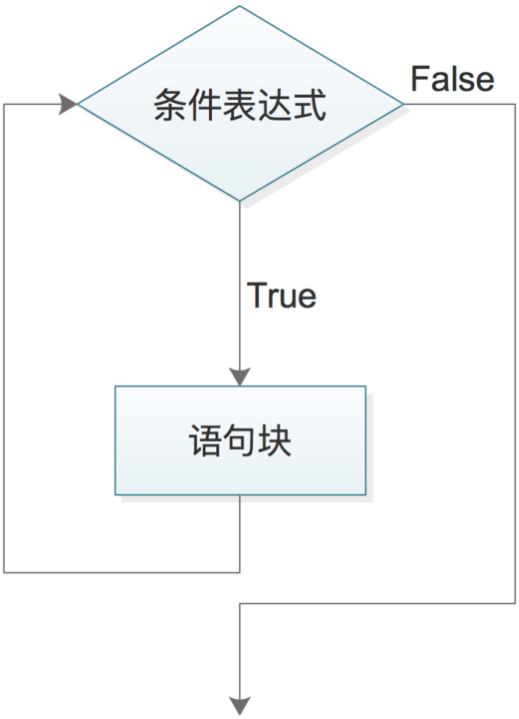
循环结构的特点是在一定条件下重复执行，直至重复执行到一定次数或该条件不再成立为止。

(1). while语句

while语句的功能是在满足指定条件时执行一个语句块。其语法结构如下：

```
    while 循环条件表达式：
        语句块
```

while语句的执行流程如下：



例题：1加到100

```
i, num = 0,0

while i<=100:
    num+=i
    i+=1
print(num)
```

(2). for语句

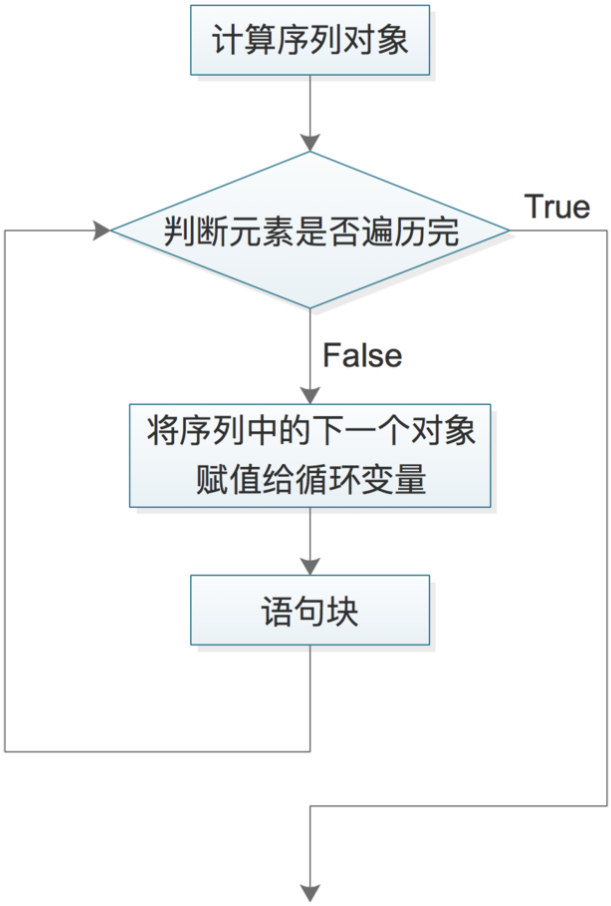
在Python中，for语句是一个通用的序列迭代器，可以用于遍历（是指沿着某条搜索路线，依次对序列中每个对象做且仅做一次访问）任何有序序列对象中的所有元素。语法格式：

```
for 循环遍历 in 序列对象：
    语句块
```

序列对象有：

- 字符串
- 列表
- 元祖
- 集合
- 字典

for语句的执行流程：



例题1：获取字符串中的所有字符

```
str = input("输入字符串")

for char in str:
    print(char)
```

例题2： 遍历0-99的整数序列

> range函数可以创建一个整数序列

```
# 打印0到99
for i in range(100): # range创建的是一个半开区间。包左不包右
    print(i)
```

例题3：找到1000以内的素数

> 素数是大于1的自然数。特点：除了1和自身外不能被其他自然数整除。要判断一个数是否素数，可以用2，4,...,`$\sqrt{n}$`去整除它。如果不能被整除，则n为素数。

2.5.4 循环控制语句

如果想要改变循环的执行流程，可以使用Python提供的循环控制语句

(1). break语句

break语句用于终止当前循环的执行操作。

例题： 前100个自然数之和

```
i, sum = 1, 0

while 1:
    sum += i
    i += 1
    if i == 101: break
print("1+2+...+1000={}".format(sum))
```

(2). continue语句

continue语句用于跳出本次循环。

```
for str in "Python":
    if str == "t": continue
    print(str) # p,y,h,o,n
```

(3). pass语句

为了保持程序结构的完整性，Python提供了一个空语句pass。pass语句一般仅作为占位语句，不做任何事情。

```
for str in "Python":
    if str == "t":
        pass
        print("---")
    print(str) # p,y,---,t,h,o,n
```

2.6 函数

函数是从英文function翻译过来的。function在英文中的意思是”功能”。从本质意义上来说，函数就是用来完成一定功能的。在编程中，函数就是一组用于完成特定功能的语句块。

2.6.1 函数的定义与调用

函数定义语法格式如下：

```
def 函数名(形式参数):
    函数体
```

- 函数体第一行可以使用文档字符串，用于存放函数说明；
- 函数体可以使用return来结束函数，使系统有选择性地返回一个值给调用代码；如果未使用return语句，或者使用了不带表达式的return语句，系统会返回None。
- 函数体中使用一个pass语句，将定义一个空函数。

【例题】定义一个计算矩形面积的函数，并查看其帮助

```
def area(width, height):
    '''
    function: 计算矩形面积
    :param width: 宽度
    :param height: 高度
    :return: 矩形的面积
    '''
    return width * height

help(area) # 查看函数
```

函数调用语法：

```
函数名(实际参数)
```

- 实际参数应当与形式参数按顺序一一对应，而且参数的数据类型需要保持一致
- 函数定义和函数调用可以在同一个程序文件中，此时函数定义必须位于函数调用之前。
- 函数定义和函数调用可以放在不同的程序文件中，此时需要先导入函数定义所在的模块，然后才能调用函数

【例题】调用area函数，并计算值

```
a = area(3, 4)
print(a) # 12
```

2.6.2 参数的传递

函数的参数传递主要有两种类型：

- 值传递： 传递的是实参变量的值。不会影响到实参变量
- 引用传递：传递的是实参变量的地址。会影响到实参变量

在Python中，函数参数传递机制采用的是对象引用传递方式。这种方式是值传递与引用传递的结合。在参数内部对形参变量所指向对象的修改是否会影响到函数外部。这要取决于对象本身的性质。 在python中对象分为可变对象和不可变对象：

- 可变对象：包括列表、集合和字典。如果参数属于可变对象，则相当于按引用传递
- 不可变对象：包括了数字、字符串、元祖、不可变集合，相当于按值传递

```
def fun(num, str, st, tup):
    num += 10
```

```
    str = str.upper()
    st.add("set")
    tup = (1, 2, 3)


num = 10
str = "hello"
st = {1, 2, 3}
tup = (1, 2, 3)
fun(num, str, st, tup)
print(num)  # 10
print(str)  # hello
print(st)  # {1, 2, 3, 'set'}  # 只有可变对象值受到函数的影响
print(tup)  # (1, 2, 3)
```

2.6.3 参数的类型

(1). 位置参数

调用参数时，通常是按照位置匹配方式传递参数，即按照从左到右的顺序将各个实参依次传递给相应的形参，此时要求实参的数目和形参的数目相等。

```
def add(x, y, z):
    return x + y + z


print(add(3, 4, 5))  # 12
```

(2). 关键字参数

调用函数时，如果不想按照位置匹配的方式传递参数，则可以使用传递关键字参数，即通过形参的名称来指定将实参值传递给哪个形参。 语法格式：

```
def add(x, y, z):
    return x + y + z


result = add(x=3, z=4, y=2)
print(result) # 9
```

(3). 默认值参数

定义函数时可以为形参指定默认值。 语法格式如下：

```
形参名称=默认值
```

默认值参数必须位于形参列表的最右端。如果对于一个形参设置了默认值，则必须对其右边的所有形参设置默认值。 调用带有默认值参数的函数时，如果未提供参数值，则形参会取默认值。

```
def student(name, gender="M"):
    print("姓名：{}\t性别：{}".format(name, gender))
student("张三")  # 姓名：张三  性别：M
```

(4). 元祖类型的变长参数

定义参数时，如果参数数目不固定，则可以定义元祖类型变长参数。方法是在形参前面加“*”号。这样的形参可以接受任意多个实参并将其封装成一个元祖。

这种元祖类型的变长参数可以看成是可选项，调用函数时可以向其传递任意多个实参值，各个实参值用逗号分割。当不提供任何实参时，相当于提供了一个空元祖作为参数。

```
def add(*args):
    x = 0
    if len(args) > 0:
        for i in args:
            x += i
    return x


print(add(3, 4, 4))  # 11
```

如果函数还有其他形参，一般放在这类变长参数之前。如果放在变长参数之后，调用时需要使用关键字参数的形式：

```
def add(*args, power):
    x = 0
    if len(args) > 0:
        for i in args:
            x += i ** power
    return x

# 在元祖变长参数之后的参数，在函数调用时需要用关键字参数形式
print(add(3, 4, power=2)) # 25
```

(5). 字典类型变长参数

定义函数时，可以定义字典类型的变长参数，方法是在形参名称前面加两个星号“**”。如果函数还有其他形参，则必须放在这类变长参数之前。 调用函数时，定义的字典类型变长参数可以接受任意多个实参，各个实参之间以逗号分隔。语法：

键=实参值

```
def student(sid, **args):
    print(sid, end="\t")
    if len(args):
        for stu in args.items():
            print("{}={}".format(stu[0], stu[1]), end="\t")


student("20190101", name="zs", age=18) # 20190101    name=zs age=18
```

第三章 高级特性

3.1 对象池

pythobn中，所有数据类型都是对象，即使是值类型，变量中存放的也不是数据值，而是值类型对象的内存地址。这种通过内存地址间接访问数据对象的方式称为引用。

通过变量之间的赋值可以使两个变量引用相同的对象，而使用身份运算符is则可以判定两个变量是否引用同一个对象

```
In [1]: a = 3

In [2]: b = a  # 将a中的内存地址赋值给b

In [3]: print(a is b) # 判断是否引用了同一个对象
True
```

实际上，Python为了优化速度，使用了**小整数对象池**， 避免为整数频繁申请和销毁内存空间。[-5, 256] 这些整数对象是提前建立好的，不会被垃圾回收。所有位于这个范围内的整数使用的都是同一个对象

```
In [1]: a = 3

In [2]: b = 3

In [3]: print(a is b)
True
```

整数类型的池中的值是不可变的，也就是说当修改数据的值时，会将修改后的值所在的内存地址赋值给变量，而不会影响原来的值：

```
In [16]: a = 3

In [17]: print(id(a)) # id函数可以获取变量引用的内存地址
4492330400

In [18]: a += 1

In [19]: print(id(a))
4492330432
```

而对于其他的值类型，即使值相同，也不是引用的同一个对象：

```
In [6]: d = 3.1

In [7]: e = 3.1

In [8]: print(d is e)
False
```

对于其他的基本数据类型，只要值相同，它们引用的也是同一个对象：

```
In [1]: str_a = "abc"

In [2]: str_b = "abc"

In [3]: print(str_a is str_b)
True

In [4]: a = True

In [5]: b = True

In [6]: print(a is b)
True

In [7]: c = None

In [8]: d = None

In [9]: print(c is d)
True
```

3.2 高阶函数

在python中，调用函数时也可以将其它函数名称作为实参来使用，这种能够接受函数名称作为参数的函数称为高阶函数。

3.2.1 函数式编程

在python中，可以将函数名称赋值给变量，赋值后变量指向函数对象；也允许将函数名称作为参数传入另一个函数，还允许从函数中返回另一个函数。

```
def add(x, y):  # 定义加法函数
    return x + y


def substract(x, y):  # 定义减法函数
    return x - y
```

```
def multiply(x, y):  # 定义乘法函数
    return x * y


def divide(x, y):  # 定义乘法函数
    return x / y


def artithmetic(x, y, operate):  # 定义算术运算函数。形参operate接受函数名称
    return operate(x, y)


a, b = eval(input("请输入两个数（以，号分割：）"))
c = input("请输入运算符")
if c == "+":
    op = add  # 函数名称赋值给变量
elif c == "-":
    op = substract
elif c == "*":
    op = multiply
elif c == "/":
    op = divide
else:
    op = -1
if op != -1:
    print(artithmetic(a, b, op))  # 将实参op指向的函数作为参数赋值给函数
```

3.2.2 map函数

内置的map()函数是一个可用于序列对象的高阶函数，其调用格式如下：

```
    map(func,iterable)
```

- func 用于指定映射函数
- iterable：用于指定可迭代对象
- return: map()函数将映射函数func()作用到可迭代对象iterable的每一个元素并组成新的map对象返回

【例题】将输入的所有单词转换为大写

```
upper = str.upper  # 为了防止str变量覆盖内置函数


def get_strs():
    strs = []
    print("请输入英文单词（q=退出）")
```

```
    while 1:
        str = input("请输入：")
        if str.lower() == "q": break
        strs.append(str)
    return strs


strs = get_strs()

print("=" * 30)

print("输入的单词：")
for str in strs:
    end_str = "," if strs.index(str) < len(strs) - 1 else "\n"
    print(str, end=end_str)

print("=" * 30)
print("格式化后的单词")
# 调用map函数，以 upper作为参数，也就是将其执行的upper函数传入map函数。第二个参数为可迭代的对象
strs = map(upper, strs)
list = list(strs)  # map对象转换为列表。目的是使用列表中的index方法
i = 0
for str in list:
    i += 1
    end_str = "," if list.index(str) < len(list) - 1 else "\r"
    print(str, end=end_str)
```

3.2.3 filter函数

内置函数filter()也是一个高阶函数，可用于对序列进行筛选。格式如下：

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

- func：指定筛选函数
- iterable：指定可迭代对象
- func()函数将作用于迭代对象中的每一个元素，并根据func的返回值是True还是False类判断是保留还是丢弃该元素
- return： 返回值是一个filter对象。

【例题】 奇数筛选

```
# 奇数筛选函数
def is_odd(num):
    return num % 2 == 1
```

```
nums = [x for x in range(1, 51)]
# 筛选奇数
nums_fliter = filter(is_odd, nums)

filter_list = list(nums_fliter)

i = 0
for lst in filter_list:
    i += 1
    print(lst, end="\t")
    if i % 5 == 0: print("\r")
```

3.3 匿名函数

匿名函数是指没有名称的函数，它只能包含一个表达式，而不能包含其他语句，该表达式的值就是函数的返回值。

使用def语句创建用户自定义函数时必须指定一个函数名称，以后需要时便可以通过该名称来调用函数，这样有助于提高代码的复用性。如果某项计算功能只需要临时使用一次而不需要在其他地方重复使用，则可以考虑通过定义匿名函数来实现这项功能。

3.3.1 匿名函数的定义

语法格式：

```
lambda 参数列表:表达式
```

- 关键字lamba 表示匿名函数，因此匿名函数也称为lambda函数
- 参数列表中的各个参数以逗号分隔
- 表达式的结果就是函数的返回值

3.3.2 匿名函数的调用

函数式编程是匿名函数的主要应用场景。虽然没有名称，但匿名函数仍然是函数对象，在程序中可以将匿名函数赋值给一个变量，然后通过该变量来调用匿名函数。

```
f = lambda x, y: x * y

res = f(3, 4)
print(res)  # 12
```

3.3.3 作为高阶函数的参数

【例题】 匿名函数作为过滤函数的参数，用来筛选出1-10之间的所有奇数

```
odd = filter(lambda x: x % 2 == 1, range(1, 11))
print(list(odd))  # [1, 3, 5, 7, 9]
```

3.3.4 作为序列或字典的元素

```
op = {"add": lambda x, y: x + y, "sub": lambda x, y: x - y, "multi": lambda
x, y: x * y, "divide": lambda x, y: x / y}

res = op["add"](3, 4)
print(res) # 7
```

3.3.5 作为函数的返回值

【例题】 算术四则运算

```
def arithmetric(c):
    if c == "+":
        return lambda x, y: x + y
    elif c == "-":
        return lambda x, y: x - y
    elif c == "*":
        return lambda x, y: x * y
    elif c == "/":
        return lambda x, y: x / y


if __name__ == '__main__':
    x, y = eval(input("请输入两个数字"))
    choice = input("请选择一种运算方式:")
    if choice in ["+", "-", "*", "/"]:
        print("{0}{1}{2}={3}".format(x, choice, y, arithmetric(choice)(x,
y)))
    else:
        print("无效的运算方式")
```

3.4 递归函数

递归函数是指自我调用的函数，即在一个函数内部直接或间接调用该函数自身。 递归函数具有以下的特性：

- 必须有一个明确的递归结束条件
- 每当进入更深一层递归时，问题规模相比上次递归都应该有所减少
- 相邻两次递归之间有紧密的联系。通常前一次的输出会作为后一次的输入。

【例题】 从键盘输入一个正整数，然后计算其阶乘 阶乘运算可以表示为：

$$\begin{cases} 1 & n \leq 1 \\ n(n-1)! & n > 1 \end{cases}$$

```
def fact(n):
    if n>1:
        return n*fact(n-1)
    else:
        return 1
print(fact(4))
```

程序中能够对变量进行存取操作的范围称为变量的作用域。变量按照作用域的不同一般分为局部变量和全局变量。

3.5 变量作用域

3.5.1 局部变量

在一个函数体或语句块内部定义的变量称为局部变量。局部变量的作用域就是定义它的函数体或语句块。

定义一个函数时，可以在函数体内部定义另一个函数，此时两个函数形成嵌套关系，内层函数只能在外层函数中调用，而不能在模块级别被调用

在具有嵌套关系的函数中，在外层函数中定义的局部变量可以直接在内层函数中使用。默认情况下，不属于当前局部作用域的变量具有只读性质，可以直接对其进行读取，但如果对其赋值，则python会在当前作用域中定义个新的同名局部变量。

如果外层函数和内层函数中定义了同名变量，则在内层函数中将优先使用自身所定义的局部变量；在存在同名变量的情况下，如果要在内层函数中使用外层函数中定义的局部变量，则应使用关键字nonlocal对变量进行声明。

```
def outer():
    x = 3

    def inner():
        nonlocal x  # 声明使用外层变量
        x = 5  # 对外层变量进行赋值
        print("内层函数中，x={}".format(x))

    inner()
    print("外层函数中，x={}".format(x))


if __name__ == "__main__":
    outer()
```

运行结果：

```
内层函数中，x=5
外层函数中，x=5
```

如果不用nonlocal进行声明：

```
def outer():
    x = 3

    def inner():

        x = 5  # 对x赋值，会创建一个与同名变量
        print("内层函数中，x={}".format(x))

    inner()
    print("外层函数中，x={}".format(x))


if __name__ == "__main__":
    outer()
```

运行结果：

```
内层函数中，x=5
外层函数中，x=3
```

3.5.2 全局变量

在python中，一个python程序文件就是一个模块。模块级别上，在所有函数外部定义的变量称为全局变量。它可以在当前模块范围内被引用。

默认情况下，在python程序中引用变量的优先顺序如下：

1. 当前作用域局部变量
2. 外层作用域变量
3. 当前模块中的局部变量
4. python内置变量

如果在某个函数内部定义的局部变量与全局变量同名，则优先使用局部变量。在这种情况下，如果要在函数内部使用全局变量，则应使用global关键字对变量进行声明，否则会创建同名的局部变量。

```
x = "global"
y = "other global"

def func():
    x = "local"
    global y  # 声明使用全局变量
    y = "local"
    print("函数内部x={},y={}".format(x,y))

func()
print("模块中，x={},y={}".format(x,y))
```

运行结果：

```
函数内部x=local,y=local
模块中，x=global,y=local
```

通过定义全局变量可以在函数之间提供直接传递数据的通道。

- 将一些参数的值存放在全局变量中，可以减少调用函数时传递的数据量
- 将函数的执行结果存放在全局变量中，则可以使函数返回多个值

3.6 闭包

如果在函数内部定义了另一个函数，在内层函数中对外层函数中定义的局部变量进行了存取操作，并且外层函数返回对内层函数的引用，则这个内层函数称为闭包（closure）。 闭包是函数式编程的重要语法结构，是将函数的语句和执行环境打包在一起的对象。当执行嵌套函数时，闭包将获取内层嵌入函数所需要的整个环境。闭包是由函数和与其相关的环境组合成的实体。在程序运行时可以生成闭包的多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

```
def outer(name):
    msg = "学生信息"
    # inner就是一个闭包
    def inner(age, gender):
        print("{}:{},{},{}".format(msg, name, age, gender))  # 使用外层函数中
的信息

    return inner  # 返回内层函数


if __name__ == "__main__":
    stu1 = outer("张三")  # 返回的是内层函数
    stu1(18, "男")  # 调用内层函数
    stu2 = outer("李四")
    stu2(19, "女")
```

3.7 装饰器

装饰器(decorator)是在闭包的基础上发展起来的。装饰器在本质上也是一个嵌套函数，其外层函数的返回值是一个新的函数对象的引用，所不同的是，其外层函数可以接受一个现有函数对象引用作为参数。

装饰器可以用于包装现有函数，即在不修改任何代码的前提下为现有函数增加额外功能。

装饰器通常应用于有切面（面向切面编程是指运行时动态实现程序维护的一种技术）需求的场景，例如插入日志、性能测试、事务处理 缓存以及权限校验等。装饰器是解决这类问题的绝佳设计，有了装饰器，就可以抽离出大量与函数功能本身无关的雷同代码并继续重用它们。

3.7.1 无参数装饰器

引入装饰器是为了在不修改原函数定义和函数调用代码的情况下拓展程序的功能。装饰器是在闭包的基础上传递了一个函数，然后覆盖原来函数的执行入口，以后调用这个函数时即可额外添加一些功能。

(1) 定义装饰器

装饰器的实质是一个高阶函数，其参数是要装饰的函数名，其返回值是完成装饰的函数名，其作用是为已经存在的函数对象添加额外的功能，其特点是不需要对象有函数做任何代码上的变动。

定义装饰器通常会涉及以下3个函数：

- 装饰器函数：它在嵌套关系中作为外层函数出现，其函数体内容包括定义一个内层函数以完成装饰功能的函数，通过return语句向调用者返回内层函数对象引用。
- 目标函数：即需要进行装饰的函数，它作为装饰器函数的形参出现，该函数的定义则出现在调用装饰器的地方
- 完成装饰的函数：它在函数嵌套中作为内层函数出现，用于为待装饰的目标函数添加额外的功能。在这个内层函数中调用目标函数，并未目标函数添加一些新的功能

装饰器的语法模板：

```
def 装饰器名称(待装饰函数名称):
    def 装饰函数名称():
        #目标函数前添加功能
        #目标函数调用
        #目标函数执行后添加功能
    return 装饰函数名称
```

例如：

```
def decorator(func):     # 定义装饰器函数
    def wrapper(x, y):  # 定义完成装饰的函数
        print("参数x={},参数y={}".format(x, y)) # 添加打印参数的功能
        return func(x, y) # 调用待装饰函数

    return wrapper # 返回完成装饰的函数的对象引用
```

(2) 调用装饰器

调用装饰器时，需要在装饰器函数前面加上符号“@”，后面跟要装饰的目标函数定义。 语法如下：

```
@装饰器名称
def 目标函数名称():
    函数体

目标函数调用
```

例如：

```
@decorator  # 调用装饰器
def add(x, y):  # 定义目标函数
    return x + y

print("{}+{}={}".format(2, 4, add(2, 4)))  # 调用模板函数
```

调用装饰器后，调用目标函数时实际调用的是装饰函数：

```
@decorator  # 调用装饰器
def add(x, y):  # 定义目标函数
    return x + y


print(add)
```

结果：

```
<function decorator.<locals>.wrapper at 0x115403f28>
```

目标函数如果不添加装饰器，调用的则是函数自身：

```
def add(x, y):  # 定义目标函数
    return x + y


print(add) # <function add at 0x11a427268>
```

【例题】 通过装饰器为模拟游戏函数添加计时功能

```
import time


def decorator(func):
    def wrapper():
        print("游戏现在开始：")
        start = time.time()
        func()
        end = time.time() - start
        print("游戏历时{:2f}".format(end))
    return wrapper

@decorator
def play_game():
    for i in range(100000000):
        pass

if __name__=="__main__":
    play_game()
```

运行结果：

```
游戏现在开始：
游戏历时2.08
```

3.7.2 有参装饰器

有参装饰器的调用格式是”@z装饰器名称(参数)“。通过提供参数，可以为装饰器的定义和调用带来更大的灵活性。

无参数装饰器本质就是一个双层结构的高阶函数；有参数装饰器则是一个三层结构的高阶函数。有参数装饰器可以看做是在无参数装饰器外面又封装了一层函数。

所谓有参数是指最外层装饰器函数可以有一个或多个参数，这些参数可以在内部各层函数中使用，而且最外层装饰器的返回值就是内层的无参数装饰器。

【例题】 通过有参数装饰器为模拟游戏添加计时功能，并允许指定游戏名称。

```
import time


def decorator(name):  # 最外层函数接收参数
    def _decorator(func):  # 中层函数用于接受要装饰的目标函数
        def wrapper():  # 内层函数用于完成装饰功能
            print("《{0}》游戏现在开始:".format(name))
            start = time.time()
            func()
            end = time.time() - start
            print("《{}》游戏历时{:.2f}:".format(name, end))

        return wrapper

    return _decorator


@decorator("王者荣耀")
def play_game1():
    for i in range(10000000):
        pass


@decorator("英雄联盟")
def play_game2():
    for i in range(10000000):
        pass


if __name__ == "__main__":
    play_game1()
    play_game2()
```

调用模板函数时，实际调用的还是最内层的无参的完成装饰的函数：

```
 print(play_game2)  # <function decorator.<locals>._decorator.
<locals>.wrapper at 0x1123b0598>
```

3.7.3 多重装饰器

多重装饰器时指使用多个装饰器来修改同一个函数，此时要注意多重装饰器的执行顺序是后面的装饰器先执行，前面的装饰器后执行，即后来者居上

```
# 定义第一个装饰器
def first(func):
    print("函数{}传递到了第一个装饰器".format(func.__name__))
```

```
    def _first(*args, **kw):
        print("在_first()中调用函数{}".format(func.__name__))
        return func(*args, **kw)

    return _first


# 定义第二个装饰器
def second(func):
    print("函数{}传递到了第二个装饰器".format(func.__name__))

    def _second(*args, **kw):
        print("在_second()中调用函数{}".format(func.__name__))
        return func(*args, **kw)

    return _second


# 调用两个装饰器
@first
@second
def test():
    print("现在开始执行test()函数")


if __name__ == "__main__":
    test()
```

运行结果：

```
函数test传递到了第二个装饰器
函数_second传递到了第一个装饰器
在_first()中调用函数_second
在_second()中调用函数test
现在开始执行test()函数
```

3.8 模块

在python中，一个扩展名为".py"的程序文件就是一个模块(module)。为了方便组织和维护代码，通常可以将相关的代码存放到一个python模块中。

3.8.1 模块的定义与使用

(1) 导入整个模块

语法：

```
import 模块名1 [as 别名1] [,模块名n [as 别名n]]
```

- 模块名是去掉扩展名".py"后的文件名
- 导入多个模块时，各个模块名之间用逗号分隔

导入一个模块后，就可以调用该模块中的所有函数，调用格式如下：

```
 模块名或别名.函数名(参数)
```

也可以将"模块名.函数名"赋值给一个变量，然后通过该变量来调用模块中的函数。

(2) 从模块中导入指定项目

语法格式：

```
from 模块名 import 项目1 [,项目n]
```

通过这种方式导入模块中指定的项目，在这种情况下可以直接调用函数，而不需要添加模块名作为前缀。

如果要导入所有项目，可以使用下面的语句：

```
from 模块名 import *
```

【案例】 首先创建一个名为moudule.py文件，添加如下代码：

```
def test():
    print("我是moudule模块中的代码")
```

然后创建一个demo.py文件，添加如下代码：

```
import module  # 导入模块

module.test()  # 调用模块中的函数
```

3.8.2 设置模块的搜索路径

python解释器遇到import语句时，如果所指定的模块包含在搜索路径列表中，系统会导入该模块

python导入模块时的的搜索路径可以通过sys.path对象来查看：

```
import sys
print(sys.path)
```

该对象是一个列表：

- 第一个元素为当前程序文件所在的目录
- 还包括标准模块所在目录
- 通过PYTHONPATH环境变量配置的目录
- 在扩展名为".pth"的文件中设置的目录

(1) 动态添加模块搜索路径

通过调用sys.path.insert(0,path)或sys.path.append(path)函数可以动态地添加模块搜索路径，将指定的目录添加到搜索路径中。例如：

```
import sys
sys.path.insert(0,"/Users/fray.hao/")
print(sys.path)
```

使用这种方式添加的模块搜索目录是临时性的，只在程序运行期间有效。当退出python环境后，搜索路径将失效

(2) 通过环境变量配置搜索路径

要设置永久的python模块搜索路径，可以使用PYTHONPATH环境变量来配置搜素路径。所设置的路径会自动添加到sys.path列表中，而且可以在不同的pyahton版本中共享。

【例题】mac中添加环境变量

首先打开文件

```
vim ~/.bash_profile
```

添加环境变量：

```
# 配置python搜索路径
export  PYTHONPATH="/Users/free.hao":"/Users/fray.hao"
```

最后使文件生效：

```
source ~/.bash_profile
```

(3) 使用扩展名为".pth"文件设置搜索路径

要永久设置pytho的搜索路径，也可以在python安装路径的site-packages目录（sys.path列表中的第四个元素就是该目录的路径）中创建一个扩展名为".pth"的文件，并将模块的搜索路径写进去，其中每一个目录占一行。例如，创建search.pth，它的内容：

```
/Users/free.hao
/Users/
```

3.8.3 模块探微

(1) dir()

在python中，模块也是函数，加载一个模块之后，可以使用内置函数dir()列出该模块对象所包含的函数名称和全局变量名称的列表 例如有一个文件名为demo.py的文件，其源代码如下：

```
'''
    模块名：demo
    内容：全局变量x,y；函数func()
    功能：用于演示
'''
x=123
y="demo"
def func():
    '''
    功能：打印hello world
    功能：打印hello world
    :return：无
    '''
    print("hello world")
```

当其它程序文件中导入demo模块时，可以使用dir()函数来查看demo中包含的变量和函数。

```
import demo

print(dir(demo))
```

运行结果：

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'func', 'x', 'y']
```

dir()函数返回的是一个列表对象，其中包括在该模块中定义的所有变量名称和函数名称，还有一些内置全局变量的名称。 内置全局变量功能如下：

全局变量	功能
__builtins__	对内置模块的引用。该模块在python启动后首先被加载。该模块中的函数即内置函数，可直接被使用
__cached__	表示当前模块经过编译后生成的字节码文件的路径
__doc__	表示当前模块的文档字符串
__file__	表示当前模块的完整路径
__loader__	表示用于加载模块的加载器
__name__	表示当前模块执行过程中的名称。当直接运行模块时，则模块名为”__main__“。当模块被其它模块导入时，值为模块的名称
__package__	表示当前模块所在的包，也就是获取导入文件的路径
__spec__	表示当前模块的规范（名称，加载器和源文件）

(2) help()

将模块名传入help()，可以查看关于模块的名称、函数、变量、文件路径等信息

```
help(demo)
```

运行结果：

```
Help on module demo:
NAME
    demo
DESCRIPTION
    模块名：demo
    内容：全局变量x,y；函数func()
    功能：用于演示
FUNCTIONS
    func()
        功能：打印hello world
        功能：打印hello world
        :return: 无
DATA
    x = 123
    y = 'demo'
FILE
    /Users/free.hao/PycharmProjects/Demo/demo.py
```

也可以将模块的函数名传入，以查看关于该函数的帮助信息

```
help(demo.func)
```

运行结果：

```
Help on function func in module demo:
func()
    功能：打印hello world
    :return: 无
```

(3) __name__

Python模块中可以定义一些变量、函数和类，以便其他模块导入和调用；模块中也可以包含能够直接运行的代码，如函数的调用。当导入该模块时，不仅导入了一些变量和函数，还会直接调用函数。 在很多情况下只希望在直接运行模块时才去执行函数的调用，在被其他模块导入时，则不执行。这种情况下就可以使用__name__来加以区分。当直接运行模块时内置全局变量__name__的名称为__main__，因此可以通过下面的语句对函数的调用进行限制：

```
# 只有在直接运行模块时才会调用函数。在其他模块中导入时，不会调用
if __name__ == "__main__":
    func()
```

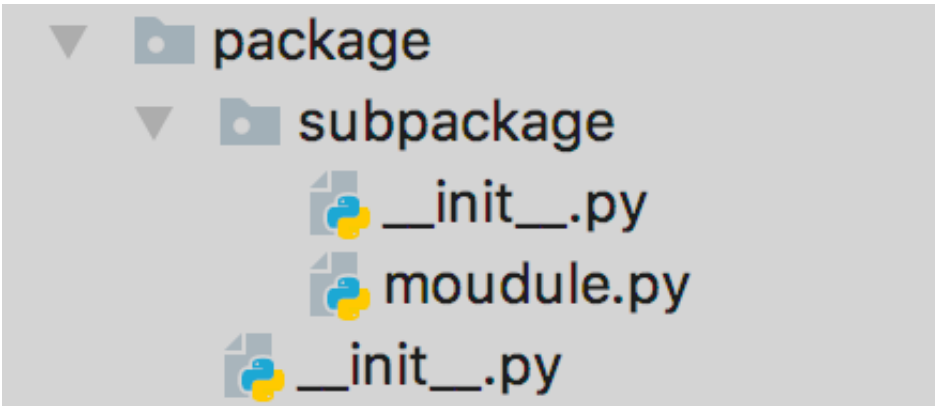
3.9 包

创建许多模块后，如果希望某些功能相近的模块组织在同一个文件夹下，就需要用到包。

在python中，包是一个有层次的文件目录结构，它定义了由若干模块或子包组成的应用程序执行环境。

包是python模块文件所在的目录。在包目录中必须有一个文件名为__init__.py的包定义文件，用于对包进行初始化操作。

如下面的包结构：



包的使用方法与模块类似。根据需要，可以从包中导入单独的模块，例如：

```
import package.subpackage.moudule # 最后一项必须是包或模块，不能是类、函数或变量
```

使用这种语法格式导入带包的模块时，必须通过完整路径形式来使用模块中的函数或变量。例如：

```
package.subpackage.moudule.test()
```

可以使用from ... import语句来导入包中的模块：

```
from  package.subpackage import moudule

moudule.test() # 此时，可以直接使用模块名而不用加上包前缀
```

还可以直接导入模块中的函数或变量：

```
from  package.subpackage.moudule import test

test() # 直接使用函数，不需要模块前缀
```

还可以在包文件__init__.py通过列表变量__all__设置要导入的模块的列表，则可以使用模糊导入。例如，在subpacakge包的包文件中添加：

```
__all__=["moudule"]
```

则可以使用模糊导入：

```
from package.subpackage import *
moudule.test()
```

第四章 面向对象

面向对象编程是目前比较流行的程序设计方法。在面向对象程序设计中，数据和对数据的操作可以封装在一个独立的数据结构中，这个数据结构就是对象。对象之间通过消息的传递来进行相互作用。

4.1 面向对象的基本概念

4.1.1 对象

对象是现实世界中的任何事物。对象不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

- 对象的状态或特征用数据值表示出来就是属性
- 对象的操作用于改变对象的状态，这些操作就是方法。

4.1.2 类

类是对象的模板，是对一组具有相同属性和相同操作的对象的抽象。类实际上是一种数据类型，一个类所包含的数据和方法用于描述一组对象的共同属性和行为。类的属性是对象状态的抽象，类的操作是对象行为的抽象。

类是对象的抽象化，而对象则是类的具体化，是类的实例。

4.1.3 消息

一个程序中通常包含多个消息，不同对象之间通过消息相互联系、相互作用。

消息由某个对象发出，用于请求另一个对象执行某项操作，或者回复某些消息。

在发送者将一个消息发送给某个对象时，消息包含接收对象去执行某种操作的信息。

发送一条消息至少需要：

- 接受消息的对象名（对象名）
- 接受对象要执行的操作的名称（方法名）
- 还可以包含参数。参数可以是认识该消息的对象所知道的变量名，或者是所有对象都知道的全局变量名

4.1.4 封装

封装是指将对象的数据和操作数据的过程结合起来所构成的单元，其内部消息对外部是隐藏的，外界不能直接访问对象的属性，而只能通过类对外提供的接口对类进行进行各种操作，这样可以保证程序中数据的安全性。

类是实施数据封装的工具，对象则是封装的具体实现，是封装的基本单位。

定义类时将其成员分为：

- 公有成员
- 私有成员

- 保护成员
  这样形成了类的访问机制，使得外界不能随意存取对象的内部数据。

4.1.5 继承

继承是指在一个类的基础上定义一个新的类。原有的类称为基类、超类或父类，新生成的类称为派生类或子类。

子类不仅可以通过继承从父类中得到所有的属性和方法，也可以对所得到的这些方法进行重写和覆盖，还可以添加一些新的属性和方法，从而扩展父类的功能。

父类体现出对象的共性和普遍性，子类则体现出对象的个性和特殊性。

继承反应了抽象程度不同的类之间的关系，即共性和个性的关系，普遍性和特殊性的关系。

4.1.6 多态

多态是指一个名称相同的方法产生了不同的动作行为，即不同对象收到相同的消息时产生了不同的行为方式。

多态可以通过覆盖和重载两种方式来实现：

- 覆盖：在子类中重新定义父类的成员方法
- 重载：允许存在多个同名函数，而这些函数的参数列表有所不同。

4.1.7 面向对象编程的基本步骤

1. 通过定义类来设置数据类型的的数据和行为
2. 基于类创建对象
3. 通过存取对象的属性或调用对象的方法来完成所需的操作

4.2 类与对象

类是一种自定义复合数据类型。

4.2.1 类的定义

类可以通过class语句来定义，其语法格式如下：

```
class 类名:
    类体
```

类体中定义类的变量成员和函数成员。

- 变量成员即类的属性，用于描述对象的状态和特征
- 函数成员即类的方法，用于实现对象的行为和操作

通过定义类可以实现数据和操作的封装。

在python中，一节皆对象。定义类时便创建了一个新的自定义类型对象，简称类对象。类名即指向类对象。此时可以通过类名和圆点运算符”.“来访问类的属性，其语法格式如下：

```
类名.属性名
```

【例题】 类定义示例

```
class Student:
    name="张三"
    gender ="男"

print(Student.name) # 访问类的属性
```

4.2.2 创建对象

类是对象的模板，对象是类的实例。定义类之后，可以通过赋值语句来创建类的实例对象，其语法如下：

```
对象名=类名(参数)
```

创建对象之后，该对象就拥有类中定义的所有属性和方法，此时可以通过对象名和圆点运算符来访问这些属性和方法，其语法格式如下：

```
对象名.属性名
对象名.方法名(参数)
```

【例题】利用类和对象计算圆的周长和面积

```
import math


class Circle:  # 定义类
    radius = 0  # 定义类的属性

    def getPerimeter(self):  # 定义类的方法。参数self代表类的实例对象
        return 2 * math.pi * self.radius

    def getArea(self):
        return math.pi * self.radius * self.radius


if __name__ == "__main__":
    c1 = Circle()  # 创建类的实例对象
    c1.radius = 10  # 设置对象的属性值
    print("圆的半径为：{}".format(c1.radius))
    print("周长={0},面积={1}".format(c1.getPerimeter(), c1.getArea()))  # 调
用类的方法
```

4.3 属性与方法

4.3.1 成员属性

在类中定义的变量成员就是属性。属性按所属的对象可以分为类属性和实例属性

- 类属性：类对象所拥有的属性，属于该类的所有实例对象
- 实例属性：类的实例对象所拥有的属性，属于该类的某个特定实例的属性

(1) 类属性

类属性按能否在外部访问可以分为公有属性和私有属性。它们都可以通过在类中定义成员变量来创建，创建类之后也可以在类定义的外部通过类名和圆点运算符来添加公有属性，定义属性时，如果属性名以双下画线"__"开头，则该属性就是私有属性，否则就是公有属性

在类的所有方法之外（即属性），无论是公有属性还是私有属性都可以通过变量名来访问，在类的成员方法内部则要通过"类名.属性名"形式来访问。 在类的外部，公有属性仍然可以通过"类名.属性名"形式来访问，私有属性则不能通过这种形式来访问，如果一定要类的外部访问类的私有属性，则必须使用一个新的属性名来访问该属性。这个新的属性名以一条下划线开头，后跟类名和私有属性名、 例如，在MyClass类的内部定义了一个名为**attr的私有成员属性。则在类成员方法之外可以直接通过**attr形式访问这个私有属性，在类成员方法中则通过以下相似来访问：

```
MyClass__attr
```

在类的外部，这个私有属性必须通过以下形式来访问：

```
MyClass_MyClass__attr
```

【类属性示例】

```
from inspect import isfunction


class DemoClass:
    pub_attr = 111  # 定义公有属性
    __priv_attr = 222  # 定义私有属性
    attr = pub_attr + __priv_attr  # 定义公有属性，引用了公有属性和私有属性

    def showAttrs(self):  # 定义类的实例方法，self表示当前实例
        for x in self.__class__.__dict__.items():  # 遍历类的属性和方法组成的字
典
            if isfunction(x[1]):  # 判断是否函数
                print("成员方法：{}".format(x[0]))
            elif type(x[1]) == int:
                if x[0].find("_DemoClass") == -1:  # 判断是否公有属性
                    print("公有属性：{}，值为{}".format(x[0], x[1]))
```

```
                else:
                    print("私有属性：{}，值为{}".format(x[0], x[1]))


if __name__ == "__main__":
    DemoClass().showAttrs()  # 调用实例方法
    print("-" * 30)
    DemoClass.pub_attr = 444  # 类外部对公有属性赋值
    DemoClass._DemoClass__priv_attr = 333  # 类外部对私有属性赋值
    DemoClass.attr3 = 555  # 添加新的公有属性
    DemoClass().showAttrs()
```

运行结果：

```
公有属性：pub_attr，值为111
私有属性：_DemoClass__priv_attr，值为222
公有属性：attr，值为333
成员方法：showAttrs
------------------------------
公有属性：pub_attr，值为444
私有属性：_DemoClass__priv_attr，值为333
公有属性：attr，值为333
成员方法：showAttrs
公有属性：attr3，值为555
```

(2) 实例属性

实例属性是某个类的实例所拥有的属性，属于该类的某个特定实例对象。实例属性可以在类的内部或外部通过赋值语句来创建

1. 在类的内部，定义类的构造方法**init**或其他实例方法时，通过在赋值语句中使用self关键字、圆点运算符和属性名类创建实例属性。语法：

```
self.属性名=值
```

- self是类的实例方法的第一个参数，代表类的当前实例。所谓实例方法就是类的实例能够使用的方法。定义实例方法时，必须设置一个用于接受类的实例的参数，而且这个参数必须是第一个参数。

2. 在类的外部，创建类的实例后，可以通过在赋值语句中使用实例对象名、圆点运算符和属性名来创建新的实例属性。语法：

```
对象名.属性名=值
```

```
class Student:
    def __init__(self, name, age):  # 定义构造函数
```

```
        self.name = name  # 定义实例属性
        self.age = age

    def showInfo(self):  # 定义实例方法
        for attr in self.__dict__.items():
            print("{}:{}".format(attr[0], attr[1]))


if __name__ == "__main__":
    stu = Student("zs", 18)  # 类的实例化
    stu.showInfo()  # 调用实例方法
    stu.gender = "male"  # 添加新的实例属性
    print("-" * 30)
    stu.showInfo()  # 调用实例方法
```

运行结果：

```
name:zs
age:18
------------------------------
name:zs
age:18
gender:male
```

(3) 类属性与实例属性的比较

区别：

1. 所属的对象不同，类属性属于类对象本身，可以由类的所有实例共享；实例属性则属性类的某个特定实例。如果存在同名的类属性和实例属性，则两者相互独立，互不影响
2. 定义的方法不同。类属性是在类中所有方法之外定义的，实例属性则是在构造方法或其他实例方法中定义的
3. 引用的方法不同。类属性是通过”类名.属性名“形式引用的，实例属性则是通过”对象名.属性名“形式引用的。

共同点和联系：

1. 类对象和实例对象都是对象。它们所属的类都可以通过**class**属性来获取，类对象属于type类，实例对象则属于创建实例时所调用的类。
2. 类对象和实例属性包含的属性及其值都可以通过**dict**属性来获取，该属性的值是一个字典，每个字典元素中包含一个属性及其值
3. 如果要读取的某个实例属性还不存在，但在类中定义了一个与其同名的类属性。则Python就会以这个类属性的值作为实例属性的值，同时还会创建一个新的实例属性，此后修改实例属性的值时，将不会对同名的类属性产生影响。

4.3.2 成员方法

定义类时，除了定义成员属性之外，还需要在类中定义一些函数，以便对类的成员属性进行操作。 类的成员方法分为：

- 内置方法
- 类方法
- 实例方法
- 静态方法

(1) 内置方法

在python中，每当定义一个类时，系统都会自动地为它添加一些默认的内置方法，这些方法通常由特定的操作触发，不需要显式调用，它们的命名有特殊的约定。 下面介绍两个常用的内置方法：

- 构造方法
- 析构方法

1) 构造方法

构造方法**init**(self,...)是在创建新对象时被自动调用的，可以用来对类的实例对象进行一些初始化的操作。

构造方法支持重载，定义类时可以根据需要重新编写构造方法，如果类中没有定义构造方法，则系统将执行默认的构造方法。

定义构造方法时，第一个参数用于接受类的当前实例，每当创建类的实例时，python会自动将当前实例传入构造方法，因此不必在类名后面的圆括号中写入这个参数。

```
class Car:
    def __init__(self, brand, color, length):
        self.brand = brand
        self.color = color
        self.length = length

    def run(self):
        print("{}的{}在行驶中".format(self.color, self.brand))


if __name__ == "__main__":
    car1 = Car("BMW", "红色", 4909)
    car1.run()  # 红色的BMW在行驶中
```

2) 析构方法

析构方法**del**(self)是在对象被删除之前自动调用的，不需要在程序中显式调用。

析构方法支持重载，通常可以通过该方法执行一些释放资源的操作。

```
class Demo:
    counter = 0

    def __init__(self, name):
```

```
        self.name = name
        self.__class__.counter += 1  # 修改类属性
        print("创建实例{},当前一共有{}个实例".format(self.name, self.counter))

    def __del__(self):
        self.__class__.counter -= 1
        print("删除实例{}当前一共有{}个实例".format(self.name, self.counter))


def func(name):
    print("函数调用开始")
    Demo(name)  # 在函数中创建对象
    print("函数调用结束")


if __name__ == "__main__":
    a = Demo("a")
    func("func")
    print("程序运行结束")
```

运行结果：

```
创建实例a,当前一共有1个实例
函数调用开始
创建实例func,当前一共有2个实例
删除实例func当前一共有1个实例
函数调用结束
程序运行结束
删除实例a当前一共有0个实例
```

(2) 类方法

类方法是类对象本身拥有的成员方法，通常可以用于对类属性进行修改。要将一个成员函数定义为类方法，必须将函数作为装饰器classmethod的目标函数，而且以类本身作为其第一个参数。语法如下：

```
@classmethod
def 函数名(cls,...):
    函数体
```

定义类方法后，可以通过类对象或实例对象来访问它，其语法格式如下：

```
类名.方法名([参数])
对象名.方法名([参数])
```

【例题】

```
class Math:
    __PI = 3.14

    @classmethod
    def getPI(cls):
        return cls.__PI


if __name__ == "__main__":
    print(Math.getPI())
    print(Math._Math__PI)
```

(3) 实例方法

实例方法是类的实例对象所用用的成员方法。定义类的实例方法时，必须以类的实例对象作为第一个参数。语法格式如下：

```
def 函数名(self,...):
    函数体
```

定义实例方法后，只能通过对象名、圆点运算符和方法名来调用它，而且不需要将对象实例作为参数传入方法中。语法格式如下：

```
对象名.函数名([参数])
```

【例题】利用实例方法计算三角形的面积

```
class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def getArea(self):
        # ---海伦公式------
        p = (self.a + self.b + self.c) / 2  # 半周长
        s = (p * (p - self.a) * (p - self.b) * (p - self.c)) ** 0.5
        return s

    def isTrangle(self):
        return self.a + self.b > self.c and self.a + self.c > self.b and
self.b + self.c > self.a


if __name__ == "__main__":
    a, b, c = eval(input("输入三角形的三个边："))
```

```
    tri = Triangle(a, b, c)
    if tri.isTrangle():
        print("三角形的面积为：{:.2f}".format(tri.getArea()))
    else:
        print("不能构成三角形")
```

(4) 静态方法

类中的静态方法即不属于类对象，也不属于实例对象，它只是类中的一个普通的成员函数。静态方法主要是用来存放逻辑性的代码，但是和类本身没有交互，即在静态方法中，不会涉及到类中的方法和属性的操作。可以理解为将静态方法存在此类的名称空间中。

与类方法和实例方法不同，静态方法可以带任意数量的参数，也可以不带任何参数。此外，如果要将类中的一个成员函数定义为静态方法，还必须将其作为装饰器staticmethod的目标函数。语法格式如下：

```
@staticmethod
def 函数名([参数])
    函数体
```

当定义一个类时，可以在类的静态方法中通过类名来访问类属性，但是不能在静态方法中访问实例属性。 在类的外部，可以通过类对象成或实例对象来调用静态方法。语法格式如下：

```
类名.静态方法名([参数])
对象名.静态方法名([参数])
```

```
import time


class Utils:
    @staticmethod
    def getTime():
        return time.strftime("%H:%M:%S", time.localtime())


if __name__ == "__main__":
    print(Utils.getTime())
```

4.3.3 私有方法

默认情况下，在类中定义的各种方法都属于公有方法，可以在类的外部调用这些公有方法。根据需要，与也可在类中创建一些各种类型的私有方法。

在类中创建某种类型的私有方法的过程与创建相同类型的公有方法类似。所不同的是，在定义私有方法时，成员函数名必须以"__"开头。

私有方法只能在类的内部使用，其调用方法与公有方法类似。不允许也不提倡在类的外部使用私有方法，如果一定要在类的外部使用私有方法，则需要使用一个新的方法名，该方法名以"_"下划线开头，后跟类名和私有方法名。

```
class Demo:
    def __priv(self):
        print("this is private function")


if __name__ == "__main__":
    demo = Demo()
    demo._Demo__priv()  # 在类外部调用私有方法
```

4.4 继承

继承是指在一个父类的基础上定义一个新的子类。子类通过继承将从父类中得到所有的属性和方法，也可以对所得到的这些方法重写和覆盖，同时还可以添加一些新的属性和方法，从而扩展父类的功能。

继承关系按父类的多少分为：

- 单一继承：子类从单个父类中继承
- 多重继承：子类从多个父类中继承

4.4.1 单一继承

语法格式：

```
class 子类名(父类名):
    类体
```

基于父类创建新的子类之后，该子类将拥有父类中的所有公有属性和所有成员方法。

除了继承父类的素有公有成员外，还可以在子类中扩展父类的功能，这可以通过两种方式来实现：

- 在子类中增加新的成员属性和成员方法
- 对父类中已有的成员方法进行重定义，从而覆盖父类的同名方法

在某些情况下，可能希望在子类中继续保留父类的功能，此时就需要调用父类的方法。在子类中可以通过父类的名称或super()函数来调用父类的方法。语法:

```
super().要调用的父类的函数名([参数])

父类名称.要调用的父类的函数名([参数])
```

类对象拥有内置的属性：

- **name**：获取类对象的类名
- **bases**：获取类对象所属的若干个父类组成的元祖

实例对象拥有内置的属性：

- **ckass**：获取该对象所属的类

此外，还可以使用内置函数isinstance()函数来判断一个对象是否属于一个已知的类型，此函数类似于内置函数type()。区别在于：

- type()不考虑继承关系，不会认为子类是一种父类类型
- isinstance()会考虑继承关系，会认为子类是一种父类类型

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

    def showInfo(self):
        print("姓名:", self.name, sep="", end="; ")
        print("年龄:", self.age, sep="")

    @classmethod
    def showClass(cls):
        print("当前类名: ", cls.__name__, sep="")  # 获取类名
        print("所属父类: ", cls.__bases__[0].__name__, sep="")  # 获取父类的类
名


# 单一继承
class Student(Person):
    def __init__(self, sid, name, age):
        self.sid = sid
        super().__init__(name, age)  # 调用父类的构造方法

    # 子类中添加新的功能
    def setScore(self, grade):
        self.grade = grade

    # 覆盖父类的方法
    def showInfo(self):
        print("学号:", self.sid, sep="", end="; ")
        super().showInfo()


if __name__ == "__main__":
    person = Person("zs", 34)
    print("*" * 5, "个人信息", "*" * 5)
    person.showInfo()
    Person.showClass()
```

```
    print("=" * 30)
    stu = Student("20190101", "李明", 18)
    print("*" * 5, "个人信息", "*" * 5)
    stu.showInfo()
    stu.showClass()
```

结果：

```
***** 个人信息 *****
姓名:zs; 年龄:34
当前类名: Person
所属父类: object
==============================
***** 个人信息 *****
学号:20190101; 姓名:李明; 年龄:18
当前类名: Student
所属父类: Person
```

4.4.2 多重继承

语法格式：

```
class 子类名(父类名1,父类名2,...)
    类体
```

在多重继承中，子类将从指定的多个父类中继承所有公有成员。为了扩展父类的功能，通常需要在子类中使用super()函数来调用父类中的方法。如果多个父类拥有同名的成员方法，使用super()函数时将会调用哪个父类的方法呢?

要搞清楚这个问题，就需要对python中类的继承机制有所了解。对于继承链上定义的各个类，python将对所有父类进行排列并计算出一个方法解析顺序（Method Resolution Order,或MRO），通过类的**mro**属性可以返回一个元祖，其中包含方法解析顺序的各个类。当调用子类的某个方法时，python将从MRO最左边的子类开始，从左到右依次查找，直至找到所需要的方法为止。如果同一个方法在不同层次的类中都存在，则从前面的类中进行选择，以保证每个父类只继承一次，这样可以避免重复继承。

```
class A:
    def __init__(self):
        print("A__init__", self)

    def say(self):
        print("A say: Hello!", self)

    @classmethod
    def showMRO(cls):
```

```
        print(cls.__name__, cls.__mro__)


class B(A):
    def __init__(self):
        print("B__init__", self)

    def eat(self):
        print("B Eatting.", self)


class C(A):
    def __init__(self):
        print("C__init__", self)

    def eat(self):
        print("C Eatting.", self)


class D(B, C):
    def __init__(self):
        super().__init__()  # 父类B,C中均有构造方法，将调用MRO中B的构造方法
        print("D__init__", self)

    def say(self):
        super().say()  # B和C均无say()方法，从MRO中找到了A中的say()
        print("D say:Hello!", self)

    def dinner(self):
        self.say()  # 将调用自己类的say()
        super().say()  # 将调用A的say()
        self.eat()  # 从MRO中找到了B的say()
        super().eat()  # 从MRO中找到了B的say()
        C.eat(self)  # 忽略MRO 调用C的eat()


if __name__ == "__main__":
    A.showMRO()  # A (<class '__main__.A'>, <class 'object'>)
    B.showMRO()  # B (<class '__main__.B'>, <class '__main__.A'>, <class
'object'>)
    C.showMRO()  # C (<class '__main__.C'>, <class '__main__.A'>, <class
'object'>)
    D.showMRO()  # D (<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'object'>)
    print("*" * 30)
    d = D()
    print("*" * 30)
    d.say()
    print("*" * 30)
```

```
    d.dinner()
```