Python 的开发环境（IDE）—— IDLE

IDE 是 Intergreated Development Environment 的缩写，中文称为集成开发环境，用来表示辅助程序员开发的应用软件，是它们的一个总称。程序员可选择的 IDE 类别是很多的，既可以选用 Python 自带的 IDLE，也可以选择使用 PyCharm、Microsoft Visual Studio、Eclipse+PyDev 和 Notepad++ 作为 IDE（**特别注意，不能使用 Windows 的记事本**）。

推荐初学者使用 Python 自带的 IDLE。因为 IDLE 的使用方法很简单，非常适合初学者入门。在安装 Python 后，会自动安装一个 IDLE，它是一个 Python Shell（可以在打开的 IDLE 窗口的标题栏上看到），程序开发人员可以利用 Python Shell 与 Python 交互，也可以创建一个文件保存代码，在全部编写完成后一起执行。

IDLE 提供的常用快捷键（可通过“Options -> Configure IDLE”菜单项，“Keys”选项卡中查看）

快提键	说 明	适用范围
F1	打开 Python 帮助文档	Shell 窗口和文件窗口
Alt+P	浏览历史命令（上一条）	Shell 窗口
Alt+N	浏览历史命令（下一条）	Shell 窗口
Alt+/	自动补全曾经出现过的单词，多个单词时可以连续循环选择	Shell 窗口和文件窗口
Alt+3	注释代码块	文件窗口
Alt+4	取消代码块注释	文件窗口
Alt+g	转到某一行	文件窗口
Ctrl+Z	撤销一步操作	Shell 窗口和文件窗口
Ctrl+Shift+Z	恢复上一次的撤销操作	Shell 窗口和文件窗口
Ctrl+S	保存文件	Shell 窗口和文件窗口
Ctrl+]	缩进代码块	文件窗口
Ctrl+[	取消代码块缩进	文件窗口
Ctrl+F6	重新启动 Python Shell	Shell 窗口

Python 标识符命名规范

1. **标识符**由字符（A~Z 和 a~z）、下划线和数字组成，但第一个字符不能是数字。
2. **标识符**不能和 以下 Python 中的保留字相同。（查看保留字：>>>import keyword　>>>keyword.kwlist）

and	as	assert	break	class	continue	def	del	elif	else	except
finally	for	from	False	global	if	import	in	is	lambda	nonlocal
not	None	or	pass	raise	return	try	True	while	with	yield

3. **标识符**不能包含空格、@、% 以及 $ 等特殊字符。
4. **标识符**的字母是严格区分大小写的。
5. **标识符**避免使用以下划线开头（下划线开头的标识符有特殊含义）
   - 以单下划线开头的标识符（如 _width）表示不能直接访问的类属性，无法通过 from...import* 导入；
   - 以双下划线开头的标识符（如 __add）表示类的私有成员；
   - 以双下划线作为开头和结尾的标识符（如 __init__），是专用标识符。
6. **模块名**应尽量短小，全部用小写字母，可使用下划线分割多个字母。如 game_mian、game_register
7. **包名称**应尽量短小，全部使用小写字母，不推荐使用下划线。如 com.mr、com.mr.book
8. **类名**采用单词首字母大写的形式。如，定义一个图书类，可以命名为 Book
9. **模块内部的类名**采用 " 下划线 + 首字母大写 " 的形式。如 _Book
10. **函数名**、**类中的属性名**和**方法名**，全部使用小写字母，多个单词之间可以用下划线分割；
11. **常量名**全部使用大写字母，单词之间可以用下划线分割；

Python 变量类型

- Python **整形**用来表示整数（即没有小数部分的数），包括正整数、0 和负整数。有 4 种表示形式：

1. **十进制形式**：最普通的整数就是十进制形式的整数，不能以 0（零）作为十进制数的开头（数值是 0 除外）。
2. **二进制形式**：由 0 和 1 组成，以 0b 或 0B 开头。例如，101 对应十进制数是 5；
3. **八进制形式**：八进制整数由 0~7 组成，以 0o 或 0O 开头（第一个是数字零，第二个是字母 O）。
4. **十六进制形式**：由 0~9 以及 A~F（或 a~f）组成，以 0x 或 0X 开头，

- Python **浮点型**用来表示带小数点的数值，有 2 种表示形式：

1. **十进制形式**：就是平常简单的浮点数，例如 5.12、512.0、0.512。必须包含一个小数点，否则会被当成整数类型处理。
2. **科学计数形式**：例如 5.12e2（即 5.12×10^2）、5.12E2（也是 5.12×10^2）。

- Python **字符串**必须用引号（单引号或者双引号）括起来。

字符串中的内容几乎可以包含任何字符（英文字符和中文字符）。当字符串中包含引号时，可使用不同的引号或转义符（\）。

当程序中有大段文本内容（可能是多行）要定义成字符串时，优先推荐使用长字符串形式（三引号），这种形式可以让字符串中包含任何内容，既可包含单引号，也可包含双引号。

Python 原始字符串以“r”开头，它不会把反斜线当成特殊字符。例如：r'G:\publish\codes\02\2.4'

◎ 获得**字符串长度**：len(string)

◎ 获得**字符串字节数**：len(str1.encode(**编码方式**))　　编码方式默认为 UTF-8

◎ 获得用户**输入的字符串**：msg = input(" **请输入你的值**：")

◎ **输出字符串**：print (value,...,sep='',end='\n',file=sys.stdout,flush=False)

» value 为要输出的内容，可以是一个变量，也可以是多个变量（以逗号分隔）；

» sep 指定 print() 函数输出多个变量时的分隔符（默认为空格）；

» end 指定 print() 函数输出的结束符（默认为换行符）；

» file 指定 print() 函数的输出目标（默认为 sys.stdout，即系统标准输出屏幕）；

» flush 用于控制输出缓存（默认为 false）

◎ **格式化字符串**（格式化输出）

Python 提供了“%”对各种类型的数据进行格式化输出，例如：

```
user = "Charli"；age = 8
print("%s is a boy" % user)                    # 格式化字符串有 1 个占位符
print("%s is a %s years old boy" % (user , age))    # 格式化字符串有 2 个占位符
```

Python 转换说明符

转换符	说明	转换符	说明
%d，%i	转换为带符号的十进制形式的整数	%g	智能选择使用 %f 或 %e 格式
%o	转换为带符号的八进制形式的整数	%G	智能选择使用 %F 或 %E 格式
%x，%X	转换为带符号的十六进制形式的整数	%c	格式化字符及其 ASCII 码
%e	转化为科学计数法表示的浮点数（e 小写）	%r	使用 repr() 将变量或表达式转换为字符串
%E	转化为科学计数法表示的浮点数（E 大写）	%s	使用 str() 将变量或表达式转换为字符串
%f，%F	转化为十进制形式的浮点数		

输出宽度控制（-、+、0）：

```
num = 30
print("num is: %6d" % num)        # 最小宽度 6，默认右对齐
print("num is: %-6d" % num)       # 最小宽度 6，左对齐
print("num is: %+6d" % num)       # 最小宽度 6，默认右对齐，带符号（正数带 +，负数带 -）
print("num is: %06d" % num)       # 最小宽度 6，默认右对齐，左边补 0
```

输出精度控制（.）：

```
my_value = 3.001415926535
print("my_value is: %8.3f" % my_value)      # 最小宽度 8，小数点后保留 3 位
the_name = "Charlie"
print("the name is: %.3s" % the_name)       # 只保留 3 个字符，输出 Cha
print("the name is: %10.2s" % the_name)     # 只保留 2 个字符，最小宽度 10
```

◎ **转义字符**

字符	说明	字符	说明
\	在行尾的续行符，即一行未完，下一行继续写	\t	水平制表符，用于横向跳到下一制表位
\'	单引号	\a	响铃
\"	双引号	\b	退格（Backspace）
\0	空	\\	反斜线
\n	换行符	\0dd	八进制数，dd 代表字符，如 \012 代表换行
\r	回车符	\xhh	十六进制数，hh 代表字符，如 \x0a 代表换行

● **bytes 类型及用法**。

bytes 对象只负责以字节（二进制格式）序列来记录数据，至于这些数据到底表示什么内容，完全由程序决定。如果采用合适的字符集，字符串可以转换成字节串；反过来，字节串也可以恢复成对应的字符串。

由于 bytes 保存的就是原始的字节（二进制格式）数据，因此 bytes 对象可用于在网络上传输数据，也可用于存储各种二进制格式的文件，比如图片、音乐等文件。

将一个字符串转换成 bytes 对象，有如下三种方式：

1. 如果字符串内容都是 ASCII 字符，则可以通过直接在字符串之前添加 b 来构建字节串值。例如：b=b'hello'。
2. 调用 bytes() 函数（其实是 bytes 的构造方法）将字符串按指定字符集转换成字节串，如果不指定字符集，默认使用 UTF-8 字符集。例如：b=bytes(' 我爱 Python 编程 ',encoding='utf-8')。
3. 调用字符串本身的 encode() 方法将字符串按指定字符集转换成字节串，如果不指定字符集，默认使用 UTF-8 字符集。例如：b=" 学习 Python 很有趣 ".encode('utf-8')。

将一个 bytes 对象转换成字符串，可调用 bytes 对象的 decode() 方法将其解码成字符串。

例如：st = b.decode('utf-8')。

● Python bool **布尔型**表示真（对）或假（错）

布尔类型可以当做整数来对待，即 True 相当于整数值 1，False 相当于整数值 0。

● Python **复数**，复数的虚部用 j 或 J 来表示。例如：3 + 0.2j、4 – 0.1J。

如果需要在程序中对复数进行计算，可导入 cmath 模块，该模块下包含了各种支持复数运算的函数。

● **数据类型转换函数**

函数	作用	函数	作用
int(x)	将 x 转换成整数类型	lloat(x)	将 x 转换成浮点数类型
complex(real，[,imag])	创建一个复数	eval(str)	计算在字符串中的有效 Python 表达式，并返回一个对象
str(x)	将 x 转换为字符串	repr(x)	将 x 转换为表达式字符串
chr(x)	将整数 x 转换为一个字符	hex(x)	将一个整数 x 转换为一个十六进制字符串
ord(x)	将一个字符 x 转换为它对应的整数值	oct(x)	将一个整数 x 转换为一个八进制的字符串

Python 运算符

- **算术运算符**

运算符	说明	实例	结果
+	加法运算，还可以作为字符串（包括后续要介绍的序列）的连接运算符	12.45 + 15	27.45
–	减法运算，还可以作为求负的运算符	4.56 – 0.26	4.3
*	乘法运算，还可以作为字符串（包括后续要介绍的序列）的连接运算符，表示将 N 个字符串连接起来。	5 * 3.6	18.0
/	除法运算	7 / 2	3.5
%	取余，即返回除法的余数	7 % 2	1
//	整除，返回商的整数部分	7 // 2	3
**	幂，即返回 x 的 y 次方	2 ** 4	16，即 24

- **赋值运算符**

运算符	说明	实例	展开形式
=	基本的赋值运算	x = y	x = y
+=	加赋值	x += y	x = x + y
–=	减赋值	x –= y	x = x – y
*=	乘赋值	x *= y	x = x * y
/=	除赋值	x /= y	x = x / y
%=	取余数赋值	x %= y	x = x % y
**=	幂赋值	x **= y	x = x ** y
//=	取整数赋值	x //= y	x = x // y
|=	按位或赋值	x |= y	x = x | y
^=	按位与赋值	x ^= y	x = x ^ y
<<=	左移赋值	x <<= y	x = x << y，这里的 y 指的是左移的位数
>>=	右移赋值	x >>= y	x = x >> y，这里的 y 指的是右移的位数

- **位运算符**

运算符	说明	实例	结果
&	按位与	a & b	4 & 5
|	按位或	a | b	4 | 5
^	按位异或	a ^ b	4 ^ 5
~	按位取反	~a	~4
<<	按位左移	a << b	4 << 2，表示数字 4 按位左移 2 位
>>	按位右移	a >> b	4 >> 2，表示数字 4 按位右移 2 位

- **比较运算符**

运算符	说明
>	大于，如果运算符前面的值大于后面的值，则返回 True；否则返回 False
>=	大于或等于，如果运算符前面的值大于或等于后面的值，则返回 True；否则返回 False
<	小于，如果运算符前面的值小于后面的值，则返回 True；否则返回 False
<=	小于或等于，如果运算符前面的值小于或等于后面的值，则返回 True；否则返回 False
==	等于，如果运算符前面的值等于后面的值，则返回 True；否则返回 False
!=	不等于，如果运算符前面的值不等于后面的值，则返回 True；否则返回 False
is	判断两个变量所引用的对象是否相同，如果相同则返回 True
is not	判断两个变量所引用的对象是否不相同，如果不相同则返回 True

- **逻辑运算符**

运算符	含义	基本格式	功能
and	逻辑与（简称“与”）	a and b	a 和 b 都是 True 时，才返回 True，否则返回 False。
or	逻辑或（简称“或”）	a or b	a 和 b 都是 False 时，才返回 False，否则返回 True。
not	逻辑非（简称“非”）	not a	a 的值为 True，返回 False； a 的值为 False，返回 True。

- **三目运算符**

语法格式：*True_statements* if *expression* else *False_statements*

运算规则：先对逻辑表达式 *expression* 求值，

如果逻辑表达式返回 True，则执行并返回 *True_statements* 的值；

如果逻辑表达式返回 False，则执行并返回 False_statements 的值。

```
a = 5
b = 3
print("a 大于 b") if a > b else print("a 不大于 b")      # 输出 "a 大于 b"
```

Python 允许在三目运算符的 *True_statements* 或 *False_statements* 中放置多条语句。支持两种放置方式：

1. 多条语句以英文**逗号**隔开：每条语句都会执行，程序返回多条语句的返回值组成的元组。
2. 多条语句以英文**分号**隔开：每条语句都会执行，程序只返回第一条语句的返回值。

- **运算符优先级**

运算符说明	Python 运算符	优先级
索引运算符	x[index] 或 x[index:index2[:index3]]	18、19
属性访问	x.attrbute	17
乘方	**	16
按位取反	~	15
符号运算符	+（正号）或 –（负号）	14
乘、除	*、/、//、%	13
加、减	+、–	12
位移	>>、<<	11
按位与	&	10
按位异或	^	9
按位或	|	8
比较运算符	==、!=、>、>=、<、<=	7
is 运算符	is、is not	6
in 运算符	in、not in	5
逻辑非	not	4
逻辑与	and	3
逻辑或	or	2

如果使用“()”就可以改变程序的执行顺序。

特别提示：虽然 Python 运算符存在优先级的关系，但并不推荐过度依赖运算符的优先级，因为这会导致程序的可读性降低。因此，在这里要提醒读者：

- 不要把一个表达式写得过于复杂，如果一个表达式过于复杂，则把它分成几步来完成。
- 不要过多地依赖运算符的优先级来控制表达式的执行顺序，这样可读性太差，应尽量使用“()”来控制表达式的执行顺序。

Python 内置的四种常用数据结构

Python 内置了四种常用数据结构：**列表**（list）、**元组**（tuple）、**字典**（dict）和**集合**（set）。

这四种数据结构都可用于保存多个数据项，这对编程而言非常重要，因为程序不仅需要使用单个变量来保存数据，还需要使用多种数据结构来保存大量数据，而列表、元组、字典和集合就可满足保存大量数据的需求。

列表（list）和**元组**（tuple）相似，都是按顺序保存元素，每个元素都有自己的索引，因此都可通过索引访问元素。二者的区别在于**元组是不可修改的**，但**列表是可修改的**。

字典（dict）和**集合**（set）类似，它们存储的数据都是无序的，其中**字典**是用 key–value 的形式保存数据。

序列指的是一块可存放多个值的连续内存空间，这些值按一定顺序排列，可通过每个值所在位置的编号（称为索引）访问它们。在 Python 中，序列类型包括字符串、列表、元组、集合和字典，这些序列支持以下几种通用的操作，但比较特殊的是，集合和字典不支持索引、切片、相加和相乘操作。

- **序列索引**

序列中每个元素都有属于自己的编号（索引）。从起始元素开始，索引值从 0 开始递增。Python 还支持索引值是负数，此类索引是从右向左计数，换句话说，从最后一个元素开始计数，从索引值 –1 开始，

- **序列切片**

切片操作是访问序列中元素的另一种方法，它可以访问一定范围内的元素，通过切片操作，可以生成一个新的序列。序列实现切片操作的语法格式如下：

sname[start : end : step]

  - sname：表示序列的名称；
  - start：表示切片的开始索引位置（包括该位置，默认为 0，即从开头进行切片）；
  - end：表示切片的结束索引位置（不包括该位置，默认为序列的长度）；
  - step：表示在切片过程中，隔几个存储位置（包含当前位置）取一次元素，也就是说，如果 step 的值大于 1，则在进行切片去序列元素时，会“跳跃式”的取元素。如果省略设置 step 的值，则最后一个冒号就可以省略。

- **序列相加**

Python 中，支持两种类型相同（“+”运算符的两侧序列要么都是序列类型，要么都是元组类型，要么都是字符串）的序列使用“+”运算符做相加操作，将两个序列进行连接，但不会去除重复的元素。

- **序列相乘**

Python 中，使用数字 n 乘以一个序列会生成新的序列，其内容为原来序列被重复 n 次的结果。比较特殊的是，列表类型在进行乘法运算时，还可以实现初始化指定长度列表的功能。

- **检查元素是否包含在序列中**

Python 中，可以使用 in 关键字检查某元素是否为序列的成员，其语法格式为：

value in sequence

  - value ：要检查的元素；
  - sequence：指定的序列。

- **和序列相关的内置函数**

函数	功能
len()	计算序列的长度，即返回序列中包含多少个元素。
max()	找出序列中的最大元素。
min()	找出序列中的最小元素。
list()	将序列转换为列表。
str()	将序列转换为字符串。
sum()	计算元素和。做加和操作的必须都是数字。
sorted()	对元素进行排序。
reversed()	反向序列中的元素。
enumerate()	将序列组合为一个索引序列，多用在 for 循环中。

Python tuple 元组

元组是 Python 中另一个重要的序列结构，和列表类似，也是由一系列按特定顺序排序的元素组成。和列表不同的是，列表可以任意操作元素，是可变序列；而元组是不可变序列，即元组中的元素不可以单独修改。

从形式上看，元组的所有元素都放在一对小括号“()”中，相邻元素之间用逗号“,”分隔。**从存储内容上看**，元组可以存储整数、实数、字符串、列表、元组等任何类型的数据，并且在同一个元组中，元素的类型可以不同。

- **创建元组** 创建列表的方法可分为 2 种：
  - ◎ **使用 = 运算符直接创建元组**（只有一个元素时，此元组后面必须要加一个逗号“,”）

```
num = (7,14,21,28,35)
a_tuple = ("C 语言中文网 ","http://c.biancheng.net")
python = ("Python",19,[1,2],('c',2.0))
```

  - ◎ **使用 tuple() 函数创建列表**。可将列表、区间（range）等对象转换为元组

```
a_list = ['crazyit', 20, –1.2]
a_tuple = tuple(a_list)              # 将列表转换成元组
a_range = range(1, 5)
b_tuple = tuple(a_range)             # 将区间转换成元组
```

- **访问元组元素**
  - ◎ 通过元组的索引获取指定的元素

```
a_tuple = ('crazyit', 20, –1.2)
print(a_tuple[1])
```

  - ◎ 通过切片操作获得获取指定范围内的元素

```
a_tuple = ('crazyit', 20, –1.2)
print(a_tuple[:2])          # 访问 a_tuple 元组中前 2 个元组
```

- **修改元组元素**
  - ◎ 重新赋值

```
a_tuple = ('crazyit', 20, –1.2)
a_tuple = ('c.biancheng.net',"C 语言中文网 ")
```

  - ◎ 通过连接多个元组的方式向元组中添加新元素

```
a_tuple = ('crazyit', 20, –1.2)
a_tuple = a_tuple + ('c.biancheng.net',)
```

- **删除元组** 使用 del 语句删除元组

元组没有列表那么多功能，但是元组依旧是很重要的序列类型之一，**元组的不可替代性体现在：**

1. 元组作为很多内置函数和序列类型方法的返回值存在，也就是说，在使用某些函数或者方法时，它的返回值会元组类型，因此你必须对元组进行处理。
2. 元组比列表的访问和处理速度更快，因此，当需要对指定元素进行访问，且不涉及修改元素的操作时，建议使用元组。
3. 元组可以在映射（和集合的成员）中当做“键”使用，而列表不行。

Python list 列表

Python 中没有数组，但是加入了更加强大的列表。**从形式上看**，列表会将所有元素都放在一对中括号 [] 中，相邻元素之间用逗号分隔。**从内容上看**，列表可以存储整数、实数、字符串、列表、元组等任何类型的数据，并且和数组不同的是，在同一个列表中元素的类型也可以不同。（注意，在使用列表时，通常情况下不这么做，同一列表中只放入同一类型的数据，这样可以提高程序的可读性。）

- **创建列表** 创建列表的方法可分为 2 种：
  - ◎ **使用 = 运算符直接创建列表**

```
num = [1,2,3,4,5,6,7]
name = ["C 语言中文网 ","http://c.biancheng.net"]
program = ["C 语言 ","Python","Java"]
```

  - ◎ **使用 list() 函数创建列表**。可将元组、区间（range）等对象转换为列表

```
a_tuple = ('crazyit', 20, –1.2)
a_list = list(a_tuple)                  # 将元组转换成列表
b_range = range(1, 5)
b_list = list(b_range)                  # 将范围转换成列表
```

- **访问列表元素**
  - ◎ 直接使用 print() 函数输出列表

```
name = ["C 语言中文网 ","http://c.biancheng.net"]
print(name)
```

  - ◎ 通过列表的索引获取指定的元素

```
name = ["C 语言中文网 ","http://c.biancheng.net"]
print(name[1])
```

  - ◎ 通过切片操作获得新的列表

```
num = [1,2,3,4,5,6,7]
print(num[2:4])
```

- **删除列表** 使用 del 语句删除列表

- **列表添加元素** 列表添加元素的 3 种方法：
  - ◎ **append() 方法在列表的末尾追加元素**

```
a_list = ['crazyit', 20, –2]
a_list.append('fkit')       # 追加元素
```

  - ◎ **extend() 方法在列表的末尾追加元素**

```
b_list = ['a', 30]
b_list.extend((–2, 3.1))            # 追加元组中的所有元素
b_list.extend(['C', 'R', 'A'])      # 追加列表中的所有元素
b_list.extend(range(97, 100))       # 追加区间中的所有元素
```

  - ◎ **insert() 方法插入元素**

```
c_list = list(range(1, 6))
c_list.insert(3, 'CRAZY' )   # 在索引 3 处插入字符串
```

- **列表删除元素** 列表删除元素的 3 种方法：
  - ◎ del() 方法：根据索引值删除元素

```
a_list=[10,20,30,40,(51,52)]
del a_list[-1]      # 删除最后一个元素
del a_list[1: 3]    # 删除第 2 个到第 4 个（不包含）元素
```

  - ◎ remove() 方法：根据元素值进行删除

```
c_list = [20, 'crazyit', 30, -4, 'crazyit', 3.4]
c_list.remove(30)                    # 删除第一次找到的 30
```

  - ◎ clear() 方法：删除列表所有元素

```
c_list = [20, 'crazyit', 30, -4, 'crazyit', 3.4]
c_list.clear()
```

- **列表修改元素**
  - ◎ 通过赋值语句修改元素

```
a_list = [2, 4, -3.4, 'crazyit', 23]
a_list[2] = 'fkit'              # 对第 3 个元素赋值
a_list[-2] = 9527               # 对倒数第 2 个元素赋值
```

  - ◎ 通过 slice 语法对列表其中一部分赋值（不能使用单个值）

```
b_list = list(range(1, 10))
b_list[1: 3] = ['a', 'b']                    # 将第 2 个到第 4 个（不包含）元素赋值为新列表的元素
c_list[2: 9: 2] = ['a', 'b', 'c', 'd']       # 指定 step 为 2，被赋值元素 4 个，用于赋值的列表也必须有 4 个
```

- list 常用方法
  - ◎ count() 方法：统计列表中某个元素出现的次数

```
a_list = [2, 30, 'a', [5, 30], 30]
print(a_list.count(30))              # 计算列表中 30 的出现次数
print(a_list.count([5, 30]))         # 计算列表中 [5, 30] 的出现次数
```

  - ◎ index() 方法：定位某个元素在列表中出现的位置（也就是索引）

```
a_list = [2, 30, 'a', 'b', 'crazyit', 30]
print(a_list.index(30))              # 定位元素 30 的出现位置
print(a_list.index(30, 2))           # 从索引 2 处开始、定位元素 30 的出现位置
print(a_list.index(30, 2, 4))        # 从索引 2 处到索引 4 处之间定位元素 30 的出现位置
```

  - ◎ pop() 方法：移除列表中指定索引处的元素（默认移除列表中最后一个元素）

```
a_list=[1,2,3]
a_list.pop()                 # 移除列表的元素 3
a_list.pop(0)                # 移除列表中索引为 0 的元素 1
```

  - ◎ reverse() 方法：将列表中所有元素反向存放
  - ◎ sort() 方法：对列表元素进行排序，排序后原列表中的元素顺序会方发生改变

```
b_list = ['Python', 'Swift', 'Ruby', 'Go', 'Kotlin', 'Erlang']
b_list.sort()                        # 对列表元素排序
b_list.sort(key=len)                 # 对列表元素按字符串的长度大小顺序排序
b_list.sort(key=len, reverse=True)   # 对列表元素按字符串的长度大小逆序排序
```

Python dict 字典

字典类型是 Python 中唯一的映射类型，它是无序的可变序列，其保存的内容是以“键值对”的形式存放的。字典具备以下特征：

- ○ **通过键而不是通过索引来读取元素** 字典类型有时也称为关联数组或者散列表（hash）。它是通过键将一系列的值联系起来的，这样就可以通过键从字典中获取指定项，但不能通过索引来获取。
- ○ **字典是任意数据类型的无序集合** 和列表、元组不同，通常会将索引值 0 对应的元素称为第一个元素。而字典中的元素是无序的。
- ○ **字典是可变的，并且可以任意嵌套** 字典可以在原处增长或者缩短（无需生成一个副本），并且它支持任意深度的嵌套，即字典存储的值也可以是列表或其它的字典。
- ○ **字典中的键必须唯一** 字典中，不支持同一个键出现多次，否则，只会保留最后一个键值对。
- ○ **字典中的键必须不可变** 字典中的值是不可变的，只能使用数字、字符串或者元组，不能使用列表。
- ● **创建字典** 创建字典的方法如下：
  - ◎ **花括号语法创建字典**

```
scores = {' 语文 ': 89, ' 数学 ': 92, ' 英语 ': 93}
empty_dict = {}
dict2 = {(20, 30):'good', 30:[1,2,3]}
```

  - ◎ **通过 fromkeys() 方法创建字典**

```
knowledge = {' 语文 ', ' 数学 ', ' 英语 '}
scores = dict.fromkeys(knowledge)
```

  - ◎ **通过 dict() 映射函数创建字典**

创建格式	注意事项
a = dict(one=1,two=2,three=3)	其中的 one、two、three 都是字符串，但使用此方式创建字典时，字符串不能带引号。

demo = [('two',2),('one',1),('three',3)] # 方式 1
demo = [['two',2],['one',1],['three',3]] # 方式 2
demo = (('two',2),('one',1),('three',3)) # 方式 3
demo = (['two',2],['one',1],['three',3]) # 方式 4
a = dict(demo)
	向 dict() 函数传入列表或元组，而它们中的元素又各自是包含 2 个元素的列表或元组，其中第一个元素作为键，第二个元素作为值。

demokeys = ['one','two','three'] demovalues = [1,2,3] a = dict(zip(demokeys,demovalues))	通过应用 dict() 函数和 zip() 函数，可将前两个列表转换为对应的字典。

- ● **访问字典** 字典是通过键来访问对应的元素值

```
a = dict(one=1,two=2,three=3)
a['one']  或  a.get('two')  或  a.get('four',' 字典中无此键 ')        # 当键不存在时，返回 default 值
```

- ● **删除字典** 使用 del 语句删除字典
- ● **添加键值对** 添加键值对，只需为不存在的 key 赋值即可

```
a = {' 数学 ':95}
a[' 语文 '] = 89
a[' 英语 '] = 90
```

- ● **修改键值对** 不是同时修改某一键值对的键和值，而只是修改某一键值对中的值

```
a = {' 数学 ': 95, ' 语文 ': 89, ' 英语 ': 90}
a[' 语文 ']=100
```

- **删除键值对** 使用 del 语句

```
a = {' 数学 ': 95, ' 语文 ': 89, ' 英语 ': 90}
del a[' 语文 ']
del a[' 数学 ']
```

- **判断是否存在指定键值对** 判断字典是否包含指定键值对的键，可以使用 in 或 not in 运算符。

```
a = {' 数学 ': 95, ' 语文 ': 89, ' 英语 ': 90}
print(' 数学 ' in a)          # True
print(' 物理 ' in a)          # False
```

- **获取字典中的特定数据**
  - ◎ keys() 方法　　用于返回字典中的所有键；
  - ◎ values() 方法　　用于返回字典中所有键对应的值；
  - ◎ items() 方法　　用于返回字典中所有的键值对。

```
a = {' 数学 ': 95, ' 语文 ': 89, ' 英语 ': 90}
print(a.keys())    #dict_keys([' 数学 ', ' 语文 ', ' 英语 '])
print(a.values())  #dict_values([95, 89, 90])
print(a.items())   #dict_items([(' 数学 ', 95), (' 语文 ', 89), (' 英语 ', 90)])
```

如果想使用返回的数据，有以下 2 种方法：

1. 使用 list() 函数，将它们返回的数据转换成列表

```
a = {' 数学 ': 95, ' 语文 ': 89, ' 英语 ': 90}
b = list(a.keys())
```

2. 利用循环结构将键或值分别赋给不同的变量

```
a = {' 数学 ': 95, ' 语文 ': 89, ' 英语 ': 90}
for k in a.keys():
    print(k,end=' ')
for v in a.values():
    print(v,end=' ')
for k,v in a.items():
    print("key:",k," value:",v)
```

- **copy() 方法** 用于返回一个具有相同键值对的新字典

```
a = {'one': 1, 'two': 2, 'three': [1,2,3]}
b = a.copy()       # b 为 {'one': 1, 'two': 2, 'three': [1, 2, 3]}
```

注意，copy() 方法所遵循的拷贝原理，既有深拷贝，也有浅拷贝。copy() 方法只会对最表层的键值对进行深拷贝，也就是说，它会再申请一块内存用来存放 {'one': 1, 'two': 2, 'three': []}；而对于某些列表类型的值来说，此方法对其做的是浅拷贝，也就是说，b 中的 [1,2,3] 的值不是自己独有，而是和 a 共有。

```
a = {'one': 1, 'two': 2, 'three': [1,2,3]}
b = a.copy()
a['four']=100
print(a)           # {'one': 1, 'two': 2, 'three': [1, 2, 3], 'four': 100}
print(b)           # {'one': 1, 'two': 2, 'three': [1, 2, 3]}
a['three'].remove(1)
print(a)           # {'one': 1, 'two': 2, 'three': [2, 3], 'four': 100}
print(b)           # {'one': 1, 'two': 2, 'three': [2, 3]}
```

- **update() 方法** 用一个字典所包含的键值对来更新已有的字典。如果被更新的字典中已包含对应的键值对，那么原 value 会被覆盖；如果被更新的字典中不包含对应的键值对，则该键值对被添加进去。

```
a = {'one': 1, 'two': 2, 'three': 3}
a.update({'one':4.5, 'four': 9.3})      # a 为 {'one': 4.5, 'two': 2, 'three': 3, 'four': 9.3}
```

- **popitem() 方法** 用于随机弹出字典中的一个键值对。此处的随机其实是假的，它和 list.pop() 方法一样，也是弹出字典中最后一个键值对。但由于字典存储键值对的顺序是不可知的，因此 popitem() 方法总是弹出底层存储的最后一个键值对。

```
a = {'one': 1, 'two': 2, 'three': 3}
print(a.popitem())          # ('three', 3)
print(a)                    # {'one': 1, 'two': 2}
```

由于 popitem 弹出的是一个元组，可以通过序列解包的方式，用两个变量分别接收 key 和 value。

```
a = {'one': 1, 'two': 2, 'three': 3}
k, v = a.popitem()
print(k, v)                             # three 3
```

- **setdefault() 方法** 用于根据 key 来获取对应 value 的值。但该方法有一个额外的功能，即当程序要获取的 key 在字典中不存在时，该方法会先为这个不存在的 key 设置一个默认的 value，然后再返回该 key 对应的 value。

```
a = {'one': 1, 'two': 2, 'three': 3}
print(a.setdefault('four', 9.2))        # 该 key 在 dict 中不存在，新增键值对，输出：9.2
print(a.setdefault('one', 3.4))         # 该 key 在 dict 中存在，不会修改 dict 内容，输出：1
print(a)                                # {'one': 1, 'two': 2, 'three': 3, 'four': 9.2}
```

- **使用字典格式化字符串** 可以使用字典对字符串进行格式化输出。

```
# 字符串模板中使用 key
temp = ' 教程是 :%(name)s, 价格是 :%(price)010.2f, 出版社是 :%(publish)s'
# 使用字典为字符串模板中的 key 传入值
book = {'name':'Python 基础教程 ', 'price': 99, 'publish': 'C 语言中文网 '}
print(temp % book)          # 教程是 :Python 基础教程 , 价格是 :0000099.00, 出版社是 :C 语言中文网
book = {'name':'C 语言小白变怪兽 ', 'price':159, 'publish': 'C 语言中文网 '}
print(temp % book)          # 教程是 :C 语言小白变怪兽 , 价格是 :0000159.00, 出版社是 :C 语言中文网
```

Python set 集合

Python 中的集合和数学中的集合概念一样，用来保存不重复的元素，即集合中的元素都是唯一的，互不相同。

从形式上看，和字典类似，集合会将所有元素放在一对大括号 {} 中，相邻元素之间用“,”分隔。**从内容上看，**同一集合中，**只能存储不可变的数据类型**（整形、浮点型、字符串、元组），**无法存储可变的数据类型**（列表、字典、集合），并且数据必须保证是唯一的，因为集合对于每种数据元素，只会保留一份。由于 set 集合是无序的，所以每次输出时元素的排序顺序可能都不相同。

Python 中的集合有两种类型：**set 类型**（可以添加、删除元素），**frozenset 类型**（不能添加、删除元素）。

- **创建 set 集合** 2 种创建 set 集合的方法：

  ◎ **使用 {} 创建**

```
a = {1,'c',1,(1,2,3),'c'}          # a: {1, 'c', (1, 2, 3)}
```

  ◎ **set() 函数创建集合** 创建空集合，只能用 set() 函数。因为使用 {}，会视为一个空字典。

```
set1 = set("c.biancheng.net")      # set1: {'a', 'g', 'b', 'c', 'n', 'h', '.', 't', 'i', 'e'}
set2 = set([1,2,3,4,5])            # set2: {1, 2, 3, 4, 5}
set3 = set((1,2,3,4,5))            # set3: {1, 2, 3, 4, 5}
```

- **访问 set 集合** 访问集合元素最常用的方法是使用循环结构，将集合中的数据逐一读取出来：

```
a = {1,'c',1,(1,2,3),'c'}          # a: {1, 'c', (1, 2, 3)}
for ele in a:
    print(ele,end=' ')             # 1 c (1, 2, 3)
```

- **删除 set 集合** 使用 del 语句删除集合
- **添加 set 集合元素** 可使用 add() 方法

```
a = {1,2,3}                        # a: {1,2,3}
a.add((1,2))                       # a: {(1, 2), 1,2,3}
a.add([1,2])                       # 报错，不能添加列表
```

- **删除 set 集合元素** 可使用 remove() 方法或 discard() 方法

```
a = {1,2,3}                        # a: {1,2,3}
a.remove(1)                        # a: {2,3}
a.remove(1)                        # 报错，不能删除不存在的元素
a.discard(1)                       # 改用 discard() 方法，删除不存在的元素时不会报错
```

- **set 集合的交集、并集、差集运算**

运算操作	运算符	含义	例子
交集	&	取两集合公共的元素	set1 & set2
并集	|	取两集合全部的元素	set1 | set2
差集	–	取一个集合中另一集合没有的元素	set1 – set2 或 set2 – set1
对称差集	^	取集合 A 和 B 中不属于 A&B 的元素	set1 ^ set2

```
set1={1,2,3}                       # set1: {1,2,3}
set2={3,4,5}                       # set1: {3,4,5}
print(set1 & set2)                 # {3}
print(set1 | set2)                 # {1, 2, 3, 4, 5}
print(set1 – set2)                 # {1, 2}
print(set2 – set1)                 # {4, 5}
print(set1 ^ set2)                 # {1, 2, 4, 5}
```

- **set 集合方法**

可使用 dir(set) 方法查看 set 类型提供的方法

```
>>> dir(set)
['add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint',
'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

方法及语法格式	功能
set1.**add()**	向 set1 集合中添加数字、字符串、元组或者布尔类型
set1.**clear()**	清空 set1 集合中所有元素
set2 = set1.**copy()**	拷贝 set1 集合给 set2
set3 = set1.**difference**(set2)	将 set1 中有而 set2 没有的元素给 set3
set1.**difference_update**(set2)	从 set1 中删除与 set2 相同的元素
set1.**discard**(elem)	删除 set1 中的 elem 元素
set3 = set1.**intersection**(set2)	取 set1 和 set2 的交集给 set3
set1.**intersection_update**(set2)	取 set1 和 set2 的交集，并更新给 set1
set1.**isdisjoint**(set2)	判断 set1 和 set2 是否没有交集
set1.**issubset**(set2)	判断 set1 是否是 set2 的子集
set1.**issuperset**(set2)	判断 set2 是否是 set1 的子集
a = set1.**pop()**	取 set1 中一个元素，并赋值给 a
set1.**remove**(elem)	移除 set1 中的 elem 元素
set3 = set1.**symmetric_difference**(set2)	取 set1 和 set2 中互不相同的元素，给 set3
set1.**symmetric_difference_update**(set2)	取 set1 和 set2 中互不相同的元素，并更新给 set1
set3 = set1.**union**(set2)	取 set1 和 set2 的并集，赋给 set3
set1.**update**(elem)	添加列表或集合中的元素到 set1

- **frozenset 集合（set 集合的不可变版本）**

frozenset 是 set 的不可变版本，因此 set 集合中所有能改变集合本身的方法（如 add、remove、discard、xxx_update 等），frozenset 都不支持；set 集合中不改变集合本身的方法，fronzenset 都支持。

查看 set 类型提供的方法：

```
>>> dir(frozenset)
['copy', 'difference', 'intersection', 'isdisjoint', 'issubset', 'issuperset', 'symmetric_difference', 'union']
```

frozenset 的作用主要有两点：

1. 当集合元素不需要改变时，使用 frozenset 代替 set 更安全。
2. 当某些 API 需要不可变对象时，必须用 frozenset 代替 set。
   比如 dict 的 key 必须是不可变对象，因此只能用 frozenset；
   再比如 set 本身的集合元素必须是不可变的，因此 set 不能包含 set，set 只能包含 frozenset。

Python 字符串常用方法

- **字符串拼接（包含字符串拼接数字）**

```
s1 = "Hello," "Charlie"      # 直接书写拼接：Hello,Charlie
s2 = "Python "               # s2: "Python "
s3 = "iS Funny"              # s3: "iS Funny"
s4 = s2 + s3                 # “+”号拼接：s4: "Python iS Funny"
```

Python 不允许直接拼接数字和字符串，程序必须先将数字转换成字符串（可用 str() 或 repr() 函数）。

```
s1 = " 这是数字 : "
p = 99.8
print(s1 + p)                # 字符串直接拼接数值，程序报错
print(s1 + str(p))           # 使用 str() 将数值转换成字符串
print(s1 + repr(p))          # 使用 repr() 将数值转换成字符串
```

str 是 Python 内置的类型（和 int、float 一样），而 repr() 则只是一个函数，它的另一个功能是：可以 Python 表达式的形式来表示值。

```
st = "I will play my fife"
print (st)          # I will play my fife
print(repr(st))     # 'I will play my fife'（带上了引号）
```

- **字符串截取（字符串切片）**

字符串本质上就是由多个字符组成的，Python 允许通过索引来操作字符，比如获取指定索引处的字符，获取指定字符在字符串中的位置等。

```
s = 'crazyit.org is very good'
print(s[2])                  # 获取 s 中索引 2 处的字符：a
print(s[-4])                 # 获取 s 中从右边开始，索引 4 处的字符：g
print(s[3: 5])               # 获取 s 中从索引 3 处到索引 5 处（不包含）的子串：zy
print(s[3: -5])              # 获取 s 中从索引 3 处到倒数第 5 个字符的子串：zyit.org is very
print(s[-6: -3])             # 获取 s 中从倒数第 6 个字符到倒数第 3 个字符的子串：y g
print(s[::2])                # 每隔 1 个，取一个字符：caytogi eygo
print(s[5: ])                # 获取 s 中从索引 5 处到结束的子串：it.org is very good
print(s[-6: ])               # 获取 s 中从倒数第 6 个字符到结束的子串：y good
print(s[: 5])                # 获取 s 中从开始到索引 5 处的子串：crazy
print(s[: -6])               # 获取 s 中从开始到倒数第 6 个字符的子串：crazyit.org is ver
print('very' in s)           # 使用 in 运算符判断 s 是否包含 'very' 子串：True
print('fkit' in s)           # 使用 in 运算符判断 s 是否包含 'fkit' 子串：False
print(max(s))                # 使用 max() 函数获得 s 字符串中最大的字符：z
print(min(s))                # 使用 min() 函数获得 s 字符串中最小的字符：空格
```

- **统计子字符串出现次数**

count() 方法用于检索指定字符串在另一字符串中出现的次数，如果检索的字符串不存在，则返回 0，否则返回出现的次数。该方法的基本语法格式如下：

str.count(sub[,start[,end]])

- str：表示原字符串；
- sub：表示要检索的字符串。
- start：指定检索的起始位置，也就是从什么位置开始检测。如果不指定，默认从头开始检索；
- end：指定检索的终止位置，如果不指定，则表示一直检索到结尾。

- **字符串分割和合并**
  - ◎ **split() 方法** 可以实现将一个字符串按照指定的分隔符切分成多个子串，这些子串会被保存到列表中（不包含分隔符），作为方法的返回值反馈回来。该方法的基本语法格式如下：

    str.split(sep,maxsplit)

    - ○ str：表示要进行分割的字符串；
    - ○ sep：指定分隔符，可以包含多个字符。默认为 None，表示所有空字符，包括空格、换行符“\n”、制表符“\t”等。
    - ○ maxsplit：可选参数，指定分割的次数，最后列表中子串的个数最多为 maxsplit+1。如果不指定或者指定为 -1，则表示分割次数没有限制。
  - ◎ **join() 方法**是 split() 方法的逆方法，用来将列表（或元组）中包含的多个字符串连接成一个字符串。该方法的基本语法格式如下：

    newstr = str.join(iterable)

    - ○ newstr：合并后生成的新字符串；
    - ○ str：指定合并时的分隔符。
    - ○ iterable：做合并操作的源字符串数据，允许以列表、元组等形式提供。

- **检索子字符串**
  - ◎ **find() 方法** 用于检索字符串中是否包含目标字符串，如果包含，则返回第一次出现该字符串的索引；反之，则返回 -1。该方法的基本语法格式如下：

    str.find(sub[,start[,end]])

    - ○ str：表示原字符串；
    - ○ sub：表示要检索的字符串。
    - ○ start：指定检索的起始位置，也就是从什么位置开始检测。如果不指定，默认从头开始检索；
    - ○ end：指定检索的终止位置，如果不指定，则表示一直检索到结尾。
  - ◎ **index() 方法** 同 find() 方法类似，也可以用于检索是否包含指定的字符串，不同之处在于，当指定的字符串不存在时，index() 方法会抛出异常。该方法的基本语法格式如下：

    str.index(sub[,start[,end]])

    - ○ str：表示原字符串；
    - ○ sub：表示要检索的字符串。
    - ○ start：指定检索的起始位置，也就是从什么位置开始检测。如果不指定，默认从头开始检索；
    - ○ end：指定检索的终止位置，如果不指定，则表示一直检索到结尾。

- **检测字符串的开头和结尾 startswith() 和 endswith() 方法**
  - ◎ **startswith() 方法** 用于检索字符串是否以指定字符串开头，如果是返回 True；反之返回 False。该方法的基本语法格式如下：

    str.startswith(sub[,start[,end]])

    - ○ str：表示原字符串；
    - ○ sub：表示要检索的字符串。
    - ○ start：指定检索的起始位置，也就是从什么位置开始检测。如果不指定，默认从头开始检索；
    - ○ end：指定检索的终止位置，如果不指定，则表示一直检索到结尾。
  - ◎ **endswith() 方法** 用于检索字符串是否以指定字符串结尾，如果是则返回 True；反之则返回 False。该方法的基本语法格式如下：

    str.endswith(sub[,start[,end]])

    - ○ str：表示原字符串；
    - ○ sub：表示要检索的字符串。
    - ○ start：指定检索的起始位置，也就是从什么位置开始检测。如果不指定，默认从头开始检索；
    - ○ end：指定检索的终止位置，如果不指定，则表示一直检索到结尾。

- **字符串大小写转换**
  - ◎ **title() 方法** 用于将字符串中每个单词的首字母转为大写，其他字母全部转为小写，转换完成后，此方法会返回转换得到的字符串。如果字符串中没有需要被转换的字符，此方法会将字符串原封不动地返回。该方法的基本语法格式如下：

    str.title()

    - ○ str：表示要进行转换的字符串；
  - ◎ **lower() 方法** 用于将字符串中的所有大写字母转换为小写字母，转换完成后，该方法会返回新得到的字符串。如果字符串中原本就都是小写字母，则该方法会返回原字符串。该方法的基本语法格式如下：

    str.lower()

    - ○ str：表示要进行转换的字符串；
  - ◎ **upper() 方法** 用于将字符串中的所有小写字母转换为大写字母，和以上两种方法的返回方式相同，即如果转换成功，则返回新字符串；反之，则返回原字符串。该方法的基本语法格式如下：

    str.upper()

    - ○ str：表示要进行转换的字符串；

- **去除字符串中空格（删除指定字符）**
  - ◎ **strip() 方法** 用于删除字符串左右两边的空格和特殊字符。该方法的基本语法格式如下：

    str.strip([chars])

    - ○ str：表示原字符串；
    - ○ chars：用来指定要删除的字符，可以同时指定多个，如果不手动指定，则默认会删除空格以及制表符、回车符、换行符等特殊字符。
  - ◎ **lstrip() 方法** 用于去掉字符串左边的空格和特殊字符。该方法的基本语法格式如下：

    str.lstrip([chars])

    - ○ str：表示原字符串；
    - ○ chars：用来指定要删除的字符，可以同时指定多个，如果不手动指定，则默认会删除空格以及制表符、回车符、换行符等特殊字符。
  - ◎ **rstrip() 方法** 用于删除字符串右边的空格和特殊字符。该方法的基本语法格式如下：

    str.rstrip([chars])

    - ○ str：表示原字符串；
    - ○ chars：用来指定要删除的字符，可以同时指定多个，如果不手动指定，则默认会删除空格以及制表符、回车符、换行符等特殊字符。

- **字符串格式化输出方法**

  使用 % 操作符对各种类型的数据进行格式化输出是早期 Python 提供的方法。自 Python 2.6 版本开始，字符串类型（str）提供了 format() 方法对字符串进行格式化。该方法的基本语法格式如下：

  str.format(args)

  - ○ str：用于指定字符串的显示样式；
  - ○ args：用于指定要进行格式转换的项，如果有多项，之间有逗号进行分割。

  学习 format() 方法的难点，在于搞清楚 str 显示样式的书写格式。在创建显示样式模板时，需要使用 {} 和: 来指定占位符，其完整的语法格式为：

  { [index][: [[fill] align] [sign] [#] [width] [.precision] [type]] }

  - » index：指定：后边设置的格式要作用到 args 中第几个数据，数据的索引值从 0 开始。如果省略此选项，则会根据 args 中数据的先后顺序自动分配。
  - » fill：指定空白处填充的字符。注意，当填充字符为逗号 (,) 且作用于整数或浮点数时，该整数（或浮点数）会以逗号分隔的形式输出。
  - » align：指定数据的对齐方式；

    <（数据左对齐）；>（数据右对齐）；=（数据右对齐，同时将符号放置在填充内容的最左侧，该选项只对数字类型有效）；^（数据居中，此选项需和 width 参数一起使用）。

» sign：指定有无符号数；
+（正数前加正号，负数前加负号）；-（正数前不加正号，负数前加负号）；**空格**（正数前加空格，负数前加负号）；#（对于二进制数、八进制数和十六进制数，使用此参数，各进制数前会分别显示 0b、0o、0x 前缀；反之则不显示前缀）。
» width：指定输出数据时所占的宽度；
» .precision：指定保留的小数位数；
» type：指定输出数据的具体类型；

type 类型值	含义
s	对字符串类型格式化。
d	十进制整数。
c	将十进制整数自动转换成对应的 Unicode 字符。
e 或者 E	转换成科学计数法后，再格式化输出。
g 或 G	自动在 e 和 f（或 E 和 F）中切换。
b	将十进制数自动转换成二进制表示，再格式化输出。
o	将十进制数自动转换成八进制表示，再格式化输出。
x 或者 X	将十进制数自动转换成十六进制表示，再格式化输出。
f 或者 F	转换为浮点数（默认小数点后保留 6 位），再格式化输出。
%	显示百分比（默认显示小数点后 6 位）。

- **字符串编码转换**
  - ◎ encode() **方法** 用于将 str 类型转换成 bytes 类型，这个过程也称为“编码”。基本语法格式如下：

    str.encode([encoding="utf-8"][,errors="strict"])

    - ○ str：表示要进行转换的字符串；
    - ○ encoding = "utf-8"：指定进行编码时采用的字符编码（默认为 utf-8，可省略“encoding=”）。
    - ○ errors = "strict"：指定错误处理方式：strict（默认，遇到非法字符就抛出异常）；ignore（忽略非法字符）；replace（用“？”替换非法字符）；xmlcharrefreplace（使用 xml 的字符引用）。
  - ◎ decode() **方法** 用于将 bytes 类型转换为 str 类型，这个过程也称为“解码”。基本语法格式如下：

    bytes.decode([encoding="utf-8"][,errors="strict"])

    - ○ bytes：表示要进行转换的二进制数据；
    - ○ encoding = "utf-8"：指定解码时采用的字符编码（默认为 utf-8，可省略“encoding=”）。
    - ○ errors = "strict"：指定错误处理方式：strict（默认，遇到非法字符就抛出异常）；ignore（忽略非法字符）；replace（用“？”替换非法字符）；xmlcharrefreplace（使用 xml 的字符引用）。

- **帮助函数**
  - ◎ dir() 列出指定类或模块包含的全部内容（包括函数、方法、类、变量等）。
  - ◎ help() 查看某个函数或方法的帮助文档。

Python 流程控制

- **if else 条件语句**

Python 中，选择（条件）语句可细分为 3 种形式，分别是 **if 语句**、**if else 语句**和 **if elif else 语句**。if、if else 和 if elif else 之间可以相互嵌套。

if 语句	if else 语句	if elif else 语句
if 表达式： 　代码块	if 表达式： 　代码块 1 else： 　代码块 2	if 表达式 1： 　代码块 1 elif 表达式 2： 　代码块 2 elif 表达式 3： 　代码块 3 ...// 可以有零条或多条 elif 语句 else： 　代码块 n

if 表达式真假值的判断不止可以通过 **bool 类型**来判断，对于 False、None、0、""、()、[]、{}，解释器当作 False 处理。

if else 语句用法规范（注意事项）：

1. **代码块不要忘记缩进**（建议 4 个空格）
   再次强调，Python 的代码块是通过缩进来标记的，具有相同缩进的多行代码属于同一个代码块。
2. **if 代码块不要随意缩进**
   同一个代码块内的代码必须保持相同的缩进
3. **if 表达式不要遗忘冒号**

- **pass 语句**

有时候程序需要占一个位、放一条语句，但又不希望这条语句做任何事情，此时就可通过 pass 语句来实现。通过使用 **pass 语句**，可以让程序更完整。

- **assert 断言函数**

assert 断言语句和 if 分支有点类似，它用于对一个 bool 表达式进行断言，如果该 bool 表达式为 True，该程序可以继续向下执行；否则程序会引发 AssertionError 错误。assert 语句用在检查函数参数的属性（是参数是否是按照设想的要求传入），或者作为初期测试和调试过程中的辅助工具。

- **while 循环语句**

while 循环的语法格式如下：

```
while 条件表达式：      # 首先判断条件表达式的值，其值为真（True）时，则执行代码块中的语句
    代码块              # 执行完代码块后，再重新判断条件表达式的值，若仍为真，则重新执行代码块
# 直到条件表达式的值为假（False），才终止循环
```

```
a_tuple = ('fkit', 'crazyit', 'Charli')    # 使用 while 循环遍历列表和元组
i = 0
while i < len(a_tuple):                    # 只有 i 小于 len(a_list)，继续执行循环体
    print(a_tuple[i])                      # 根据 i 来访问元组的元素
    i += 1
```

- **for 循环语句**

for 循环的语法格式如下：

```
for 迭代变量 in 字符串 | 列表 | 元组 | 字典 | 集合：# 迭代变量用于存放从序列类型变量中读取出来的元素
    代码块          # 所以一般不会在循环中对迭代变量手动赋值
```

for 进行数值循环（计算 1+2+...+100 的结果）

```
result = 0                  # 保存累加结果的变量
for i in range(101):        # 从 0 开始，到 101 结束（不包括 101），步长为 1
    result += i
print(result)
```

range() 函数是 Python 内置的函数，用于生成一系列连续的整数，多用于 for 循环中。其语法格式为：

range(start,end,step)　　start：用于指定计数的起始值（默认起始值为 0）。

end：用于指定计数的结束值（不包括结束值），**此参数不能省略**。

step：用于指定步长，即两个数之间的间隔（默认步长为 1）。

for 循环遍历字符串

```
name = ' 张三 '
for ch in name:             # 使用 for 循环遍历 “张三” 字符串，迭代变量 ch 先后被赋值为‘张’和‘三’
    print(ch)               # 在循环体中，逐个打印‘张’和‘三’
```

for 循环遍历元组

```
a_tuple = ('crazyit', 'fkit', 'Charlie')
for ele in a_tuple:
    print(' 当前元素是 :', ele)
```

for 循环遍历列表（计算列表中所有数值元素的总和、平均值）

```
src_list = [12, 45, 3.4, 13, 'a', 4, 56, 'crazyit', 109.5]
my_sum = 0
my_count = 0
for ele in src_list:
    if isinstance(ele, int) or isinstance(ele, float): # isinstance() 函数用于判断某个变量是否为指定类型的实例
        print(ele)
        my_sum += ele       # 累加该元素
        my_count += 1       # 数值元素的个数加 1
print(' 总和 :', my_sum)
print(' 平均数 :', my_sum / my_count)
```

for 循环遍历字典

```
my_dict = {' 语文 ': 89, ' 数学 ': 92, ' 英语 ': 80}
for key, value in my_dict.items():          # 通过 items() 方法遍历所有 key-value 对
    print('key:', key)
    print('value:', value)
for key in my_dict.keys():                  # 通过 keys() 方法遍历所有 key
    print('key:', key)
    print('value:', my_dict[key])
for value in my_dict.values():              # 通过 values() 方法遍历所有 value
    print('value:', value)
```

统计列表中各元素出现的次数

```
src_list = [12, 45, 3.4, 12, 'fkit', 45, 3.4, 'fkit', 45, 3.4]
statistics = {}     # 定义一个字典，以列表的元素为 key，该元素出现的次数为 value
for ele in src_list:
    if ele in statistics:       # 如果字典中包含 ele 代表的 key
        statistics[ele] += 1    # 将 ele 元素代表出现次数加 1
    else:           # 如果字典中不包含 ele 代表的 key，说明该元素还未出现过
        statistics[ele] = 1     # 将 ele 元素代表出现次数设为 1
for ele, count in statistics.items():   # 遍历 dict，打印出各元素的出现次数
    print("%s 的出现次数为 :%d" % (ele, count))
```

- **列表推导式** 可利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的列表。
  列表推导式的语法格式：[表达式 for 迭代变量 in 可迭代对象 [if 条件表达式]]

```
a_list = [x * x for x in range(10)]                    # [0 , 1 , 4 , 9 , 16 , 25 , 36 , 49 , 64, 81]
b_list = [x * x for x in range(10) if x % 2 == 0]      # [0 ,4 , 16, 36, 64]
d_list = [(x, y) for x in range(5) for y in range(4)]                    # 两重循环
e_list = [[x, y, z] for x in range(5) for y in range(4) for z in range(6)]    # 三重循环
src_a = [30, 12, 66, 34, 39, 78, 36, 57, 121]
src_b = [3, 5, 7, 11]
result = [(x, y) for x in src_b for y in src_a if y % x == 0]   # src_b 可以整除 src_a 的数据对
```

- **元组推导式** 可利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的元组。
  元组推导式的语法格式：(表达式 for 迭代变量 in 可迭代对象 [if 条件表达式])
  使用元组推导式生成的结果并不是一个元组，而是一个生成器对象，这一点和列表推导式是不同的。

```
# 使用 tuple() 函数，可以直接将生成器对象转换成元组
a = (x for x in range(1,10))
print(tuple(a))                    # (1, 2, 3, 4, 5, 6, 7, 8, 9)
# 使用 for 循环遍历生成器对象，可以获得各个元素
a = (x for x in range(1,10))
for i in a:
    print(i,end=' ')               # 1 2 3 4 5 6 7 8 9 ()
# 使用 __next__() 方法遍历生成器对象，也可以获得各个元素
a = (x for x in range(3))
print(a.__next__())
print(a.__next__())
print(a.__next__())
# 注意，无论是使用 for 循环遍历生成器对象，还是使用 __next__() 方法遍历生成器对象，遍历后原生成器对象将不复存在，这就是遍历后转换原生成器对象却得到空元组的原因。
```

- **字典推导式** 可借助列表、元组、字典、集合以及 range 区间，快速生成符合需求的字典。
  字典推导式的语法格式：{ 表达式 for 迭代变量 in 可迭代对象 [if 条件表达式]}

```
# 将列表中各字符串值为键，各字符串的长度为值，组成键值对
listdemo = ['C 语言中文网 ','c.biancheng.net']
newdict0 = {key:len(key) for key in listdemo}      # {'C 语言中文网 ': 6, 'c.biancheng.net': 15}
# 交换现有字典中各键值对的键和值
newdict1 = {v: k for k, v in newdict0.items()}          # {6: 'C 语言中文网 ', 15: 'c.biancheng.net'}
# 使用 if 表达式筛选符合条件的键值对
newdict2= {v: k for k, v in newdict0.items() if v>10}      # {15: 'c.biancheng.net'}
```

- **集合推导式** 可借助列表、元组、字典、集合以及 range 区间，快速生成符合需求的集合。
  集合推导式的语法格式：{ 表达式 for 迭代变量 in 可迭代对象 [if 条件表达式] }

```
setnew = {i**2 for i in range(3)}                  # {0, 1, 4}
# 既然生成的是集合，那么其保存的元素必须是唯一的。
tupledemo = (1,1,2,3,4,5,6,6)
setnew = {x**2 for x in tupledemo if x%2==0}       # {16, 4, 36}
# 获得字典的 key 值集合
dictdemo = {'1':1,'2':2,'3':3}
setnew = {x for x in dictdemo.keys()}              # {'2', '1', '3'}
```

- **zip() 函数** 可以把两个列表“压缩”成一个 zip 对象（可迭代对象），这个对象所包含的元素是由原列表元素组成的元组。当两个列表长度不相等时，以长度更短的列表为准。

  如果使用 zip() 函数压缩 N 个列表，那么 zip() 函数返回的可迭代对象的元素就是长度为 N 的元组。

```
books = [' 疯狂 Kotlin 讲义 ', ' 疯狂 Swift 讲义 ', ' 疯狂 Python 讲义 ']
prices = [79, 69, 89]
# 使用 zip() 函数压缩两个列表，从而实现并行遍历
for book, price in zip(books, prices):
    print("%s 的价格是 : %5.2f" % (book, price))
# 疯狂 Kotlin 讲义的价格是 : 79.00
# 疯狂 Swift 讲义的价格是 : 69.00
# 疯狂 Python 讲义的价格是 : 89.00
```

- **reversed() 函数** 可接收各种序列（元组、列表、区间等）参数，然后返回一个“反序排列”的法代器，该函数对参数本身不会产生任何影响。

```
# 反序区间
[x for x in reversed(range(10))]   # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
# 反序列表、元组
b = ['a', 'fkit', 20, 3.4, 50]
[x for x in reversed(b)]           # [50, 3.4, 20, 'fkit', 'a']
# 反序字符串
c = 'Hello,Python'
[x for x in reversed(c)]           # ['n', 'o', 'h', 't', 'y', 'P', ',', 'o', 'l', 'l', 'e', 'H']
```

- **sorted() 函数** 可接收一个可迭代对象作为参数，返回一个对元素排序的列表。

```
a = [20, 30, -1.2, 3.5, 90, 3.6]
sorted(a)                          # 升序排序：[-1.2, 3.5, 3.6, 20, 30, 90]
sorted(a, reverse = True)          # 逆序排序： [90, 30, 20, 3.6, 3.5, -1.2]
b = ['fkit', 'crazyit', 'charlie', 'fox', 'Emily']
sorted(b, key = len)               # 按字符串长度：['fox', 'fkit', 'Emily', 'crazyit', 'charlie']
# 通过 sorted() 函数的帮助，程序可对可迭代对象按照由小到大的顺序进行遍历。
for s in sorted(b, key=len):
    print(s)
```

- **continue 语句和 break 语句**

  Python 提供了 2 种强制离开当前循环体的办法：

  ◎ **continue 语句**，可以跳过执行本次循环体中剩余的代码，转而执行下一次的循环。

  ◎ **break 语句**，可以完全终止当前循环。

  需要注意的是，对于带 else 块的 for 循环，如果使用 break 强行中止循环，程序将不会执行 else 块。

Python 函数

函数是执行特定任务的一段代码，程序通过将一段代码定义成函数，并为该函数指定一个函数名，这样即可在需要的时候多次调用这段代码。因此，函数是代码复用的重要手段。

- **函数的定义**

```
def 函数名 ( 形参列表 ):
    // 由零条到多条可执行语句组成的代码块
    [return [ 返回值 ]]
```

  - **函数名**：函数的名称，调用时使用。
  - **形参列表**：用于定义该函数可以接收的参数。形参列表由多个形参名组成，多个形参名之间以英文逗号 (,) 隔开。一旦在定义函数时指定了形参列表，调用该函数时就必须传入相应的参数值，也就是说，谁调用函数谁负责为形参赋值。在创建函数时，即使函数不需要参数，也必须保留一对空的“()”。

```
def my_max(x, y) :      # 定义一个函数 my_max，该函数接收两个参数：x、y
    z = x if x > y else y   # 定义一个变量 z，该变量等于 x、y 中较大的值
    return z            # 返回变量 z 的值
def say_hi(name) :      # 定义一个函数 say_hi，该函数接收一个参数：name
    print("=== 正在执行 say_hi() 函数 ===")
    return name + "，您好！ "
```

- **函数的调用**

```
函数名 ([ 形参值 ])
```

  - **函数名**：要调用的函数的名称。
  - **形参值**：当初创建函数时要求传入的各个形参的值。需要注意的是，创建函数有多少个形参，那么调用时就需要传入多少个值，且顺序必须和创建函数时一致。即便该函数没有参数，函数名后的小括号也不能省略。

```
a = 6
b = 9
result = my_max(a , b)          # 调用 my_max() 函数，将函数返回值赋值给 result 变量
print("result:", result)        # result: 9
print(say_hi(" 孙悟空 "))       # 调用 say_hi() 函数，直接输出函数的返回值
                                # === 正在执行 say_hi() 函数 ===
                                # 孙悟空，您好！
```

- **为函数提供说明文档**　函数的说明文档放在函数声明之后、函数体之前，作为函数的部分。通过 help() 函数查看函数的说明文档，也可通过函数的 __doc__ 属性访问函数的说明文档。

```
def my_max(x, y) :
    '''
    获取两个数值之间较大数的函数。
    my_max(x, y) 返回 x、y 两个参数之间较大的那个
    '''
    z = x if x > y else y
    return z
```

- **函数的参数**

函数参数的作用是传递数据给函数，令其对接收的数据做具体的操作处理。

◎ **形参和实参** 根据参数的位置不同，函数参数有 2 种形式：

1. **形式参数**（简称**形参**）：在定义函数时，函数名后面括号中的参数
2. **实际参数**（简称**实参**）：在调用函数时，函数名后面括号中的参数。

调用函数时传入实际参数的数量和位置必须和定义函数时形式参数的数量和位置保持一致。

◎ **值传递和引用传递** 根据实参的类型不同，函数参数有 2 种传递方式：

1. **值传递**：适用于实参类型为不可变类型（**字符串、数字、元组**）；
   值传递时，将实参值的副本（复制品）传入函数的形参，而实参本身不会受到任何影响。
2. **引用（地址）传递**：适用于实参类型为可变类型（**列表，字典**）；
   引用传递时，将实参的地址传递给形参，当形参的值改变时，实参的值也发生改变。

◎ **关键字参数** 是指使用形式参数的名字来确定输入的参数值。通过此方式指定函数实参时，不再需要与形参的位置完全一致，只要将参数名写正确即可。

```
def girth(width , height):                  # 定义一个函数及形参
    print("width: ", width)
    print("height: ", height)
    return 2 * (width + height)
print(girth(3.5, 4.8))                      # 传统调用函数的方式，根据位置传入参数
print(girth(width = 3.5, height = 4.8))     # 根据关键字参数来传入参数
print(girth(height = 4.8, width = 3.5))     # 使用关键字参数时可交换位置
print(girth(3.5, height = 4.8))             # 部分使用关键字参数，部分使用位置参数
```

需要注意的是，如果希望在调用函数时混合使用关键字参数和位置参数，则关键字参数必须位于位置参数之后。换句话说，在关键字参数之后的只能是关键字参数。

◎ **默认参数** 允许为参数设置默认值，即在定义函数时，直接给形式参数指定一个默认值，这样，即便调用函数时没有给拥有默认值的形参传递参数，该参数可以直接使用定义函数时设置的默认值。

```
def say_hi(name = " 孙悟空 ", message = " 欢迎来到 C 语言中文网 "):   # 为两个参数指定默认值
    print(name, ", 您好 ")
    print(" 消息是：", message)
say_hi()                                    # 全部使用默认参数
say_hi(" 白骨精 ")                          # 只有 message 参数使用默认值
say_hi(" 白骨精 ", " 欢迎学习 Python")      # 两个参数都不使用默认值
say_hi(message = " 欢迎学习 Python")        # 只有 name 参数使用默认值
```

需要注意的是，调用函数时关键字参数必须位于位置参数的后面，因此在定义函数时指定了默认值的参数（关键字参数）必须在没有默认值的参数之后。

◎ **可变参数** 又称不定长参数，即传入函数中的实际参数可以是任意多个。有 2 种形式定义可变参数：

1. **形参前添加一个 '*'** 可以接收任意多个实际参数，并将其放到一个元组中。

```
def test(a, *books) :
    print(books)
    for b in books :          # books 被当成元组处理
        print(b)
    print(a)                  # 输出整数变量 a 的值
test(5 , "C 语言中文网 " , "Python 教程 ")          # 调用 test() 函数
```

2. **形参前添加两个 '*'** 可以接收任意多个以关键字参数赋值的实际参数，并将其放到一个字典中。

```
def test(x, y, z=3, *books, **scores) :
    print(x, y, z)
    print(books)
    print(scores)
test(1, 2, 3, "C 语言中文网 " , "Python 教程 ", 语文 =89, 数学 =94)          # 调用 test() 函数
```

◎ **逆向参数收集** 指的是在程序已有列表、元组、字典等对象的前提下，把它们的元素“拆开”后传给函数的参数。逆向参数收集传入**列表**、**元组**参数之前添加一个星号。

```
def test(name, message):
    print(" 用户是 : ", name,"，欢迎消息 : ", message)
my_list = [' 孙悟空 ', ' 欢迎来 C 语言中文网 ']
test(*my_list)              # 用户是：孙悟空 ，欢迎消息：欢迎来 C 语言中文网
```

即使是可变参数，如果程序需要将一个元组传给该参数，那么同样需要使用逆向收集。

```
def foo(name, *nums):
    print("name 参数 : ", name,"，nums 参数 : ", nums)
my_tuple = (1, 2, 3)
foo('fkit', *my_tuple)      # name 参数： fkit ， nums 参数： (1, 2, 3)
```

逆向参数收集传入**字典**参数之前添加两个星号。

```
def bar(book, price, desc):
    print(book, " ( ",desc," ) VIP 价格 :", price," 元 ")
my_dict = {'price': 159, 'book': 'C 语言中文网 ', 'desc': ' 这是一个精美而实用的网站 '}
bar(**my_dict)              # C 语言中文网 （ 这是一个精美而实用的网站 ）VIP 价格 : 159 元
```

- **函数的返回值**

用 def 语句创建函数时，可以用 return [**返回值**] 指定返回的值，返回值可以是任意类型（省略返回值，将返回空值 None）。需要注意的是，return 语句在同一函数中可以出现多次，但只要有一个得到执行，就会直接结束函数的执行。

```
def add(a,b):
    c = a + b
    return c
c = add(3,4)                # 函数赋值给变量，也作为其他函数的实际参数，如：print(add(3,4))
```

如果程序需要有多个返回值，则既可将多个值包装成列表之后返回，也可直接返回多个值。如果 Python 函数直接返回多个值，Python 会自动将多个返回值封装成元组。

```
def sum_and_avg(list):
    sum = 0
    count = 0
    for e in list:
        if isinstance(e, int) or isinstance(e, float):     # 如果元素 e 是数值
            count += 1
            sum += e
    return sum, sum / count
my_list = [20, 15, 2.8, 'a', 35, 5.9, –1.8]
tp = sum_and_avg(my_list)       # 获取 sum_and_avg 函数返回的多个值，多个返回值被封装成元组
s, avg = sum_and_avg(my_list)  # 还使用序列解包来获取多个返回值
```

- **函数的递归**

在一个函数体内调用它自身，被称为函数递归。函数递归包含了一种隐式的循环，它会重复执行某段代码，但这种重复执行无须循环控制。例：已知数列：f(n + 2) = 2*f(n+ 1) +f(n)，求 f(10) 的值。

```
def fn(n) :
    if n == 0 :
        return 1
    elif n == 1 :
        return 4
    else :
        return 2 * fn(n – 1) + fn(n – 2)        # 函数中调用它自身，就是函数递归
print("fn(10) 的结果是 :", fn(10))              # 输出：fn(10) 的结果是 : 10497
```

- **变量的作用域（全局变量和局部变量）**
  - ◎ **变量的作用域**是指程序代码能够访问该变量的区域，如果超过该区域，将无法访问该变量。
  - ◎ **局部变量**是指在函数内部定义并使用的变量，只在函数内部有效。每个函数在执行时，系统都会为该函数分配一块“临时内存空间”，所有的局部变量都被保存在这块临时内存空间内。当函数执行完成后，这块内存空间就被释放了，这些局部变量也就失效了，因此离开函数之后就不能再访问局部变量了。
  - ◎ **全局变量**是指能作用于函数内外的变量，即既可以在各个函数的外部使用，也可以在各函数内部使用。定义全局变量的方式有以下 2 种：① 在函数体外定义的变量，一定是全局变量。② 在函数体内定义全局变量。即使用 global **关键字**对变量进行修饰后，该变量就会变为全局变量。
  - ◎ **获取指定作用域范围中的变量** Python 提供了三个工具函数来获取指定范围内的“变量字典”：
    1. globals()：返回全局范围内所有变量的“变量字典”。
    2. locals()：返回当前局部范围内所有变量的“变量字典”。
    3. vars(object)：获取指定对象范围内所有变量的“变量字典”，缺省 object 同 locals()。

    globals() 和 locals() 看似完全不同，但实际上也有联系，这两个函数的区别和联系有以下两点：
    1. locals() 获取当前局部范围内所有变量的“变量字典”，如果在全局范围内（在函数之外）调用 locals() 函数，会获取全局范围内所有变量的“变量字典”；而 globals() 无论在哪里执行，总是获取全局范围内所有变量的“变量字典”。
    2. 一般来说，使用 locals() 和 globals() 获取的“变量字典”只应该被访问，不应该被修改。但实际上，使用 globals() 或使用 locals() 获取的全局范围内的“变量字典”，可以真正修改全局变量：但通过 locals() 获取的局部范围内的“变量字典”，即使修改也不会影响局部变量。

```
def test ():
    age = 20
    print(age)              # 直接访问 age 局部变量， 输出 20
    print(locals())         # 访问函数局部范围的“变量数组”， {'age': 20}
    print(locals()['age'])  # 通过函数局部范围的“变量数组”访问 age 变量：20
    locals()['age'] = 12    # 通过 locals 函数局部范围的“变量数组”改变 age 变量的值
    print('xxx', age)       # 再次访问 age 变量的值，依然输出 20
    globals()['x'] = 19     # 通过 globals 函数修改 x 全局变量
x = 5
y = 20
print(globals())            # {..., 'x': 5, 'y': 20}
print(locals())             # 全局范围内使用 locals 函数，访问的是同上全局变量的“变量数组”
print(x)                    # 直接访问 x 全局变量： 5
print(globals()['x'])       # 通过全局变量的“变量数组”访问 x 全局变量 ：5
globals()['x'] = 39         # 通过全局变量的“变量数组”对 x 全局变量赋值
print(x)                    # 输出 39
locals()['x'] = 99          # 在全局范围内使用 locals 函数对 x 全局变量赋值
print(x)                    # 输出 99
```

  - ◎ **全局变量和局部变量的遮蔽现象** 全局变量默认可以在所有函数内被访问，但在函数中定义了与全局变量同名的变量，此时就会发生局部变量遮蔽（hide）全局变量的情形。

```
name = ' 全局变量 '
def test ():
    print(globals()['name'])        # 通过 globals()['name'] 访问全局变量 name：全局变量
    global name                     # 使用 global name 声明以后访问的 name 都是全局变量
    print(name)                     # 访问全局变量 name：全局变量
    name = 'name 新值 '             # 对全局变量 name 赋值：name 新值
test()
print(name)                         # name 新值
```

- **局部函数** 是指在函数体内定义的函数。在默认情况下，局部函数对外部是隐藏的，局部函数只能在其封闭（enclosing）函数内有效，其封闭函数也可以返回局部函数，以便程序在其他作用域中使用局部函数。

```
def get_math_func(type, nn) :
    def square(n) :          # 定义一个计算平方的局部函数 square
        return n * n
    def cube(n) :            # 定义一个计算立方的局部函数 cube
        return n * n * n
    def factorial(n) :       # 定义一个计算阶乘的局部函数 factorial
        result = 1
        for index in range(2, n + 1) :
            result *= index
        return result
    if type == "square" :
        return square(nn)            # 调用局部函数 square
    elif type == "cube":
        return cube(nn)              # 调用局部函数 cube
    else:
        return factorial(nn)         # 调用局部函数 factorial
print(get_math_func("square", 3))        # 输出 9
print(get_math_func("cube", 3))          # 输出 27
print(get_math_func("", 3))              # 输出 6
```

- **使用函数变量** 所有函数都是 function 对象，这意味着可以把函数本身赋值给变量，当把函数赋值给变量之后，接下来程序也可通过该变量来调用函数。

```
def pow(base, exponent) :        # 定义一个计算乘方的函数
    result = 1
    for i in range(1, exponent + 1) :
        result *= base
    return result
def area(width, height) :        # 定义一个计算面积的函数
    return width * height
my_fun = pow             # 将 pow 函数赋值给 my_fun，则 my_fun 可当成 pow 使用
print(my_fun(3 , 4))     # 输出 81
my_fun = area            # 将 area 函数赋值给 my_fun，则 my_fun 可当成 area 使用
print(my_fun(3, 4))      # 输出 12
```

- **使用函数作为函数形参** 如果希望调用函数时能动态传入代码，那么就需要在函数中定义函数形参，这样即可在调用该函数时传入不同的函数作为参数，从而动态改变这段代码。

```
def map(data, fn) :      # 定义函数类型的形参，其中 fn 是一个函数
    result = []
    for e in data :                  # 遍历 data 列表中每个元素
        result.append(fn(e))         # 并用 fn 函数对每个元素进行计算
    return result                    # 然后将计算结果作为新数组的元素
def square(n) :          # 定义一个计算平方的函数
    return n * n
def cube(n) :            # 定义一个计算立方的函数
    return n * n * n
def factorial(n) :       # 定义一个计算阶乘的函数
    result = 1
    for index in range(2, n + 1) :
        result *= index
    return result
data = [3 , 4 , 9 , 5, 8]
print(map(data , square))        # 调用 map() 函数，传入 square 函数，计算数组元素的平方
print(map(data , cube))          # 调用 map() 函数，传入 cube 函数，计算数组元素的立方
print(map(data , factorial))     # 调用 map() 函数，传入 factorial 函数，计算数组元素的阶乘
```

- **使用函数作为返回值** Python 还支持使用函数作为其他函数的返回值。

```
def get_math_func(type) :
    def square(n) :          # 定义一个计算平方的局部函数
        return n * n
    def cube(n) :            # 定义一个计算立方的局部函数
        return n * n * n
    def factorial(n) :       # 定义一个计算阶乘的局部函数
        result = 1
        for index in range(2 , n + 1):
            result *= index
        return result
    if type == "square" :
        return square        # 返回局部函数 square
    if type == "cube" :
        return cube          # 返回局部函数 cube
    else:
        return factorial     # 返回局部函数 factorial
math_func = get_math_func("cube")      # 调用 get_math_func()，得到 cube 函数
print(math_func(5))                    # 输出 125
math_func = get_math_func("square")    # 调用 get_math_func()，得到 square 函数
print(math_func(5))                    # 输出 25
math_func = get_math_func("other")     # 调用 get_math_func()， 得到 factorial 函数
print(math_func(5))                    # 输出 120
```

- **lambda 表达式** 是现代编程语言争相引入的一种语法，如果说函数是命名的、方便复用的代码块，那么 lambda 表达式则是功能更灵活的代码块，它可以在程序中被传递和调用。

  lambda 表达式只能是单行表达式。语法格式是：**lambda [parameter_list] : 表达式**

  - lambda 表达式必须使用 **lambda 关键字**定义。
  - lambda 关键字之后、冒号左边的是**参数列表**，可以没有参数，也可以有多个参数（逗号隔开）。冒号右边是该 lambda 表达式的**返回值**。

```
def get_math_func(type) :        # 该函数返回的是 Lambda 表达式
    result=1
    if type == 'square':
        return lambda n: n * n
    elif type == 'cube':
        return lambda n: n * n * n
    else:
        return lambda n: (1 + n) * n / 2
math_func = get_math_func("cube")              # 调用 get_math_func()，程序返回一个嵌套函数
print(math_func(5))                            # 输出 125
math_func = get_math_func("square")            # 调用 get_math_func()，程序返回一个嵌套函数
print(math_func(5))                            # 输出 25
math_func = get_math_func("other")             # 调用 get_math_func()，程序返回一个嵌套函数
print(math_func(5))                            # 输出 15.0
```

lambda 表达式有两个用途：

1. 对于单行函数，使用 lambda 表达式可以省去定义函数的过程，让代码更加简洁。
2. 对于不需要多次复用的函数，使用 lambda 表达式可以在用完之后立即释放，提高了性能。

Python 面向对象（一切皆对象）

- **类**：可以理解是一个模板，通过类可以创建出无数个具体实例，这一过程又称为类的实例化。
- **对象**：类并不能直接使用，通过类创建出的实例（又称对象）才能使用。
- **属性**：类中的所有变量称为属性。
- **方法**：类中的所有函数通常称为方法。不过，和函数所有不同的是，类方法至少要包含一个 self 参数。类方法无法单独使用，只能和类的对象一起使用。

- class：**定义类**　创建一个类使用 class **关键字**实现，其基本语法格式如下：

  class 类名：
  零个到多个类属性和类方法 ...

- **__init__() 类构造方法**　是一个特殊的类实例方法，称为**构造方法**（或**构造函数**）。用于创建对象时使用，每当创建一个类的实例对象时，都会自动调用构造方法。添加构造方法的语法格式如下：

  def __init__(self,...):
  代码块

  __init__() 方法可以包含多个参数，但必须包含一个名为 self 的参数，且必须作为第一个参数。

```
class Person :
    ''' 这是一个学习 Python 定义的一个 Person 类 '''
    def __init__(self,name,age):
        print(" 这个人的名字是：",name," 年龄为：",age)
zhangsan = Person(" 张三 ",20)          # 创建 zhangsan 对象，并传递参数给构造函数
# 虽然构造方法中有 self、name、age 3 个参数，但实际仅需传入 name 和 age，self u 无需手动传 u 入。
```

- **类对象的创建和使用**　使用 class 语句只能创建一个类，而无法创建类的对象，要想使用已创建好的类，需要手动创建类的对象，创建类对象的过程又称为**类的实例化**。其语法格式如下：

  类名 (参数)

```
class Person :
    ''' 这是一个学习 Python 定义的一个 Person 类 '''
    name = "zhangsan"              # 类变量 name
    age = "20"                     # 类变量 age
    def __init__(self,name,age):
        self.name = name           # 实例变量 name
        self.age = age             # 实例变量 name
        print(" 这个人的名字是：",name," 年龄为：",age)
    def say(self, content):        # 定义实例方法 say
        print(content)
p = Person(" 张三 ",20)                          # 实例化一个 Person 对象赋给变量 p
print(p.name, p.age)                            # 访问实例变量
p.say('Python 语言很简单，学习很容易！ ')          # 访问实例方法
p.skills = ['programming', 'swimming']          # 添加实例变量
del p.name                                      # 删除实例变量
def intro_func(self, content):
    print(" 我是一个人，信息为：%s" % content)
p.intro=intro_func
p.intro(p," 信息 ")          # 需要手动将调用者绑定为第一个参数 p
from types import MethodType              # 导入 MethodType
p.intro = MethodType(intro_func, p)       # 使用 MethodType 对 intro_func 进行包装 , 绑定第一个参数 p
p.intro(" 生活在别处 ")                    # 无需传入第一个参数 p
```

- **类变量（类属性）** 指的是定义在类中，在各个类方法外的变量。类变量可以在所有实例化对象中作为公用资源。**注意，类变量推荐直接用类名访问，但也可以使用对象名访问。**

```
class Address :
    detail = ' 广州 '
    post_code = '510660'
    def info (self):
        #print(detail)                  # 尝试直接访问类变量，报错
        print(Address.detail)          # 通过类来访问类变量，输出：广州
        print(Address.post_code)       # 输出 ：510660
addr1 = Address()                      # 创建类对象 addr1
addr1.info()                           # 广州，510660
addr2 = Address()                      # 创建类对象 addr2
addr2.info()                           # 广州，510660
Address.detail = ' 佛山 '              # 修改 Address 类的类变量
Address.post_code = '460110'
addr1.info()                           # 佛山，460110
addr2.info()                           # 佛山，460110
```

- **实例变量（实例属性）** 指的是定义在类的方法中的属性。实例变量只作用于调用方法的对象。**注意，实例变量只能通过对象名访问，无法通过类名直接访问。**可以通过对象访问类变量，但不能修改。

```
class Inventory:
    item = ' 鼠标 '                          # 定义类变量 item
    quantity = 2000                          # 定义类变量 quantity
    def change(self, item, quantity):        # 定义实例方法 change
        self.item = item                     # 给实例变量 item 赋值（非类变量）
        self.quantity = quantity             # 给实例变量 quantity 赋值（非类变量）
iv = Inventory()                       # 创建 Inventory 对象
iv.change(' 显示器 ', 500)             # 访问 iv 的实例方法 change
print(iv.item)                         # 访问 iv 的实例变量 item：显示器
print(iv.quantity)                     # 访问 iv 的实例变量 quantity：：500
print(Inventory.item)                  # 访问 Inventory 的类变量 item：鼠标
print(Inventory.quantity)              # 访问 Inventory 的类变量 quantity：2000
```

- **类实例方法** 通常情况下，在类中定义的方法默认都是实例方法。实例方法最少要包含一个 self 参数，用于绑定调用此方法的实例对象。实例方法通常会用类对象直接调用，当然也可以用类名调用。

```
class Person :
    def __init__(self, name = 'Charlie', age=8):    # 类构造方法，也属于实例方法
        self.name = name
        self.age = age
    def say(self, content):                         # 定义一个 say 实例方法
        print(content)
person = Person()                                   # 创建一个类对象
person.say(" 类对象调用实例方法 ")                  # 类对象调用实例方法
Person.say(person," 类名调用实例方法 ")             # 类名调用实例方法，需手动给 self 参数传值
Person.say("abc"," 类名调用实例方法 ")              # 并不要求必须绑定 Person 对象，只要给出第一个参数
```

用**实例对象**访问类成员的方法称为**绑定方法**；
用**类名**访问类成员的方法称为**非绑定方法**。

- **类方法** 类方法需要使用@ classmethod 进行修饰。类方法最少要包含一个 cls 参数，用于绑定类本身（而不是类对象），因此，在调用类方法时，无需显式为 cls 参数传参。类方法推荐使用类名直接调用，也可以使用实例对象来调用（不推荐）。

```
class Bird:
    @classmethod          # classmethod 修饰的方法是类方法
    def fly (cls):
        print(' 类方法 fly: ', cls)
Bird.fly()          # 调用类方法，Bird 类会自动绑定到第一个参数，类方法 fly: <class '__main__.Bird'>
b = Bird()
b.fly()             # 使用对象调用类方法，其实依然还是使用类调用，类方法 fly: <class '__main__.Bird'>
```

- **类静态方法** 静态方法需要使用@ staticmethod 修饰。静态方法没有类似 self、cls 这样的特殊参数，因此不会对包含的参数做任何类或对象的绑定，也正是因为如此，此方法中无法调用任何类和对象的属性和方法，静态方法其实和类的关系不大。静态方法的调用，既可以使用类名，也可以使用类对象。

```
class Bird:
    @staticmethod         # staticmethod 修饰的方法是静态方法
    def info (p):
        print(' 静态方法 info: ', p)
Bird.info(" 类名 ")         # 类名直接调用静态方法，静态方法 info: 类名
b = Bird()
b.info(" 类对象 ")          # 类对象调用静态方法，静态方法 info: 类对象
```

- **类命名空间** Python 中，编写的整个程序默认处于**全局命名空间**内，而类体则处于**类命名空间**内。Python 允许在全局范围内放置可执行代码，当执行该程序时，这些代码就会获得执行的机会。类似地，Python 同样允许在类范围内放置可执行代码，当执行该类定义时，这些代码同样会获得执行的机会。

  可执行代码被放在 Python 类命名空间与全局空间并没有太大的区别。但在全局空间和类命名空间内分别定义 lambda 表达式就有区别了：

  在全局空间内定义的 lambda 表达式，相当于一个普通函数，使用调用函数的方式来调用该 lambda 表达式，并显式地为第一个参数绑定参数值。

  在类命名空间内定义的 lambda 表达式，相当于在该类命名空间中定义了一个实例方法，必须使用调用方法的方式来调用该 lambda 表达式，Python 同样会为该方法的第二个参数（相当于 self 参数）绑定参数值。

```
global_fn = lambda p: print(' 执行 lambda 表达式，p 参数 : ', p)
class Category:
    cate_fn = lambda p: print(' 执行 lambda 表达式，p 参数 : ', p)
global_fn('fkit')           # 调用全局范围内的 global_fn，为参数 p 传入参数值
c = Category()
c.cate_fn()                 # 调用类命名空间内的 cate_fn，自动绑定第一个参数
```

- **定义属性（property() 函数和 @property）**

  正常情况下的类，它包含的属性应该是隐藏的，只允许通过类提供的方法来间接实现对类属性的访问和操作。直接用“类对象 . 属性”的方式访问类中定义的属性，破坏了类的封装原则。因此，在不破坏类封装原则的基础上能够有效操作类中的属性，类中应包含读（或写）类属性的多个 getter（或 setter）方法，这样就可以通过“**类对象 . 方法（参数）**”的方式操作属性。

  为了既保护类的封装特性，又让开发者可以使用“**对象 . 属性**”的方式方便地操作类属性，Python 中提供了 property() **函数**和 @property **装饰器**两种方法。

◎ 使用 property() 函数定义属性，基本格式如下（可根据需要只传入部分参数）：

属性名 =property(fget=None, fset=None, fdel=None, doc=None)

- ○ fget 参数用于指定获取该属性值的类方法；
- ○ fset 参数用于指定设置该属性值的方法；
- ○ fdel 参数用于指定删除该属性值的方法；
- ○ doc 是一个文档字符串，用于提供说明此函数的作用。

```
class Rectangle:
    def __init__(self, width, height):          # 定义构造方法
        self.width = width
        self.height = height
    def setsize (self , size):                  # 定义 setsize() 函数
        self.width, self.height = size
    def getsize (self):                         # 定义 getsize() 函数
        return self.width, self.height
    def delsize (self):                         # 定义 delsize() 函数
        self.width, self.height = 0, 0
    size = property(getsize, setsize, delsize, ' 用于描述矩形大小的属性 ')   # 使用 property 定义属性
print(Rectangle.size.__doc__)    # 访问 size 属性的说明文档
help(Rectangle.size)             # 通过内置的 help() 函数查看 Rectangle.size 的说明文档
rect = Rectangle(4, 3)
print(rect.size)            # 访问 rect 的 size 属性 (4, 3)
rect.size = 9, 7            # 对 rect 的 size 属性赋值
print(rect.width)           # 访问 rect 的 width 实例变量：9
print(rect.height)          # 访问 rect 的 height 实例变量：7
del rect.size               # 删除 rect 的 size 属性
print(rect.width)           # 访问 rect 的 width 实例变量：0
print(rect.height)          # 访问 rect 的 height 实例变量：0
```

◎ 通过 @property 装饰器　可以直接通过方法名来访问方法，不需要在方法名后添加一对小括号。语法格式如下：

@property # getter 方法	@ 方法名 .setter # getter 方法	@ 方法名 .deleter # deleter 方法
def 方法名 (self)	def 方法名 (self)	def 方法名 (self)
代码块	代码块	代码块

```
class Rect:
    def __init__(self,area):
        self.__area = area
    @property                   # 用 property 装饰器，为 area 属性添加 getter 方法
    def area(self):
        return self.__area
    @area.setter                # 用 setter 装饰器，为 area 属性添加 setter 方法
    def area(self, value):
        self.__area = value
    @area.deleter               # 用 deleter 装饰器，为 area 属性添加 deleter 方法
        def area(self):
        self.__area = 0
rect = Rect(30)
print(" 矩形初始化的面积是：",rect.area)          # 直接通过方法名来访问 area 方法
rect.area = 90                                  # 修改属性
print(" 矩形修改后的面积是：",rect.area)
del rect.area                                   # 删除属性
print(" 矩形删除后的面积是：",rect.area)
```

- **@ 函数装饰器**

@ staticmethod、@ classmethod 和 @property 都是 Python 的内置函数。我们也可以开发自定义的函数装饰器。当程序使用“@函数”（如函数 A）装饰另一个函数（如函数 B）时，实际上完成如下两步：

1. 将被修饰的函数（函数 B）作为参数传给 @ 符号引用的函数（函数 A）。
2. 将函数 B 替换（装饰）成第 1 步的返回值。

```
def funA(fn):
    print('A')
    fn()            # 执行传入的 fn 参数
    return 'fkit'
@funA               # 相当于：funA(funB)，funB 将会替换（装饰）成 funA() 语句的返回值；
def funB():
    print('B')
print(funB)         # 由于 funA() 函数返回 fkit，因此 funB 就是 fkit
```

通过 @ 符号来修饰函数是 Python 的一个非常实用的功能，它既可以在被修饰函数的前面添加一些额外的处理逻辑（比如权限检查），也可以在被修饰函数的后面添加一些额外的处理逻辑（比如记录日志），还可以在目标方法抛出异常时进行一些修复操作……这种改变不需要修改被修饰函数的代码，只要增加一个修饰即可。这种在被修饰函数之前、之后、抛出异常后增加某种处理逻辑的方式，就是其他编程语言中的 AOP(Aspect Orient Progiuning，面向切面编程）。

```
def auth(fn):
    def auth_fn(*args):
        print("---- 模拟执行权限检查 ----")        # 用一条语句模拟执行权限检查
        fn(*args)                                  # 回调要装饰的目标函数
    return auth_fn
@auth
def test(a, b):
    print(" 执行 test 函数 , 参数 a: %s, 参数 b: %s" % (a, b))
test(20, 15)        # 调用 test() 函数 , 其实是调用装饰后返回的 auth_fn 函数，输出结果：
                    # ---- 模拟执行权限检查 ----
                    # 执行 test 函数 , 参数 a: 20, 参数 b: 15
```

- **动态创建类（type() 函数）**

用 type() 函数查看类的类型是 type。Python 允许使用 type() 函数（相当于 type 类的构造器函数）来创建 type 对象，又由于 type 类的实例就是类，因此 Python 可以使用 type() 函数来动态创建类。

type() 创建类的语法格式是：**type(类名 , (父类 ,), 类变量和方法字典)**

- **类名**：要创建类的名称；
- **父类**：该类继承的父类名称，多个父类以逗号间隔，最后必须多一个逗号；
- **类变量和方法字典**：**格式**：dict(key=value, ...)

  key 为变量或方法名；value 是普通值，代表类变量；value 是函数，代表方法。

```
def fn(self):
    print('fn 函数 ')
Dog = type('Dog', (object,), dict(walk=fn, age=6))          # 使用 type() 定义 Dog 类
d = Dog()                                                   # 创建 Dog 对象
print(type(d))          # 查看 d 的类型：<class '__main__.Dog'>
print(type(Dog))        # 查看 Dog 的类型：<class 'type'>
d.walk()                # fn 函数
print(Dog.age)          # 6
```

- MetaClass 元类

type() 函数更适合于动态地创建相对简单的类，要创建复杂的类，则需要通过 MetaClass（元类）的方式。

MetaClass（元类），简单的理解，就是创建类的类，即创建类之后，再由类来创建实例进行应用。使用元类可以在创建类时动态修改类定义。为了使用元类动态修改类定义，程序需要先定义元类。

定义元类时，需继承 type 类，且默认的命名习惯是：类名 + MetaClass 。并且元类中需要定义并实现 __new__() 方法（一定要有返回值）。其作用是：当程序使用 class 定义新类时，如果指定了元类，那么元类的 __new__ 方法就会被自动执行。

```
class ItemMetaClass(type):        # 定义 Item 元类，继承 type
    def __new__(cls, name, bases, attrs):
        # cls 代表动态修改的类
        # name 代表动态修改的类名
        # bases 代表被动态修改的类的所有父类
        # attr 代表被动态修改的类的所有属性、方法组成的字典
        attrs['cal_price'] = lambda self: self.price * self.discount        # 动态为该类添加一个 cal_price 方法
        return type.__new__(cls, name, bases, attrs)
class Book(metaclass=ItemMetaClass):              # 定义 Book 类
    __slots__ = ('name', 'price', '_discount')
    def __init__(self, name, price):
        self.name = name
        self.price = price
    @property
    def discount(self):
        return self._discount
    @discount.setter
    def discount(self, discount):
        self._discount = discount
class CellPhone(metaclass=ItemMetaClass):      # 定义 cellPhone 类
    __slots__ = ('price', '_discount' )
    def __init__(self, price):
        self.price = price
    @property
    def discount(self):
        return self._discount
    @discount.setter
    def discount(self, discount):
        self._discount = discount
b = Book("Python 基础教程 ", 89)          # 创建 Book 对象
b.discount = 0.76
print(b.cal_price())                      # 67.64
cp = CellPhone(2399)                      # 创建 CellPhone 对象
cp.discount = 0.85
print(cp.cal_price())                     # 2039.1499999999999
```

通过使用元类可以动态修改程序中的一批类，对它们集中进行某种修改。这个功能在开发一些基础性框架时非常有用，程序可以通过使用元类为某一批需要具有通用功能的类添加方法。

Python 封装

封装（Encapsulation）是面向对象的三大特征之一（另外两个是**继承**和**多态**），指的是将对象的状态信息隐藏在对象内部，不允许外部程序直接访问对象内部信息，而是通过该类所提供的方法来实现对内部信息的操作和访问。

封装机制保证了类内部数据结构的完整性，很好地避免了外部对内部数据的影响，提高了程序的可维护性。对一个类或对象实现良好的封装，可以达到以下目的：

- 隐藏类的实现细节。
- 让使用者只能通过事先预定的方法来访问数据，限制对属性的不合理访问。
- 可进行数据检查，从而有利于保证对象信息的完整性。
- 便于修改，提高代码的可维护性。

为了实现良好的封装，需要从以下两个方面来考虑：

1. 将对象的属性和实现细节隐藏起来，不允许外部直接访问。
2. 把方法暴露出来，让方法来控制对这些属性进行安全的访问和操作。

Python 并没有提供类似于其他语言的 private 等修饰符，因此 Python 并不能真正支持隐藏。为了隐藏类中的成员，只要将 Python 类的**成员命名为以双下画线开头**的，就会把它们隐藏起来。

```
class User :
    def __hide(self):
        print(' 示范隐藏的 hide 方法 ')
    def getname(self):                  # 定义 getname 方法（获得隐藏实例变量 __name）
        return self.__name
    def setname(self, name):            # 定义 setname 方法（设置隐藏实例变量 __name）
        if len(name) < 3 or len(name) > 8:
            raise ValueError(' 用户名长度必须在 3 ~ 8 之间 ')          # 用 raise 关键字抛出异常
        self.__name = name
    name = property(getname, setname)  # 通过 name 属性获得、设置实例变量 __name
    def getage(self):                   # 定义 getage 方法（获得隐藏实例变量 __age）
        return self.__age
    def setage(self, age):              # 定义 setage 方法（设置隐藏实例变量 __age）
        if age < 18 or age > 70:
            raise ValueError(' 用户名年龄必须在 18 在 70 之间 ')       # 用 raise 关键字抛出异常
        self.__age = age
    age = property(getage, setage)          # 通过 age 属性获得、设置实例变量 __age
u = User()          # 创建 User 对象
u.name = 'fk'       # 对 name 属性赋值，实际上调用 setname() 方法
                    # 引发 ValueError: 用户名长度必须在 3 ~ 8 之间
u.name = 'fkit'     # 对 name 属性赋值，实际上调用 setname() 方法
u.age = 25          # 对 age 属性赋值，实际上调用 setage() 方法
print(u.name)       # 获得 name 属性，实际上调用 getname() 方法： fkit
print(u.age)        # 获得 age 属性，实际上调用 getge() 方法： 25
u.__hide()          # 尝试调用隐藏的 __hide() 方法：AttributeError:'User' object has no attribute 'hide'
```

最后需要说明的是，Python 其实没有真正的隐藏机制，双下画线只是 Python 的一个小技巧，Python 会“偷偷”地改变以双下画线开头的方法名，会在这些方法名前添加单下画线和类名。

```
u._User__hide()          # 输出：示范隐藏的 hide 方法，说明 Python 并没有实现真正的隐藏
u._User__name = 'fk'     # 对隐藏变量进行赋值，并可“绕开”setname() 方法的检查逻辑
print(u.name)            # 访问 User 对象的 name 属性（实际上访问 __name 实例变量）
```

Python 继承

继承是面向对象的三大特征之一，也是实现代码复用的重要手段。继承常用于创建和现有类功能类似的新类，又或是新类只需要在现有类基础上添加一些成员（属性和方法），又不想直接将现有类代码复制给新类。

Python 中，实现继承的类称为子类，被继承的类称为父类（也可称为基类、超类）。子类继承父类的语法是：在定义子类时，将多个父类放在子类之后的圆括号里。语法格式如下：

```
class 类名 ( 父类 1, 父类 2, ...):
    # 类定义部分
```

- **Python 的多继承** Python 的继承是多继承机制，即一个子类可以同时拥有多个直接父类。

当一个子类有多个直接父类时，该子类会继承得到所有父类的方法，如果多个父类中包含了同名的方法，排在前面的父类中的方法会“遮蔽”排在后面的父类中的同名方法。

```
class Item:
    def info (self):
        print("Item 中方法 :", ' 这是一个商品 ')
class Product:
    def info (self):
        print("Product 中方法 :", ' 这是一个工业产品 ')
class Mouse(Item, Product):      # Mouse 继承了 Item 类和 Product 类
    pass
m = Mouse()
m.info()          # 输出：Item 中方法：这是一个商品
```

- **父类方法重写**

子类包含与父类同名的方法的现象被称为**方法重写**（Override），也称**方法覆盖**。可以说子类重写了父类的方法，也可以说子类覆盖了父类的方法。

```
class Bird:
    def fly(self):              # Bird 类的 fly() 方法
        print(" 我在天空里自由自在地飞翔 ...")
class Ostrich(Bird):
    def fly(self):              # 重写 Bird 类的 fly() 方法
        print(" 我只能在地上奔跑 ...")
os = Ostrich()    # 创建 Ostrich 对象
os.fly()          # 执行 Ostrich 对象的 fly() 方法，将输出 " 我只能在地上奔跑 ..."
```

Python 类相当于类空间，因此 Python 类中的方法本质上相当于类空间内的函数。所以，即使是实例方法，Python 也允许通过类名调用。区别在于：在通过类名调用实例方法时，Python 不会为实例方法的第一个参数 self 自动绑定参数值，而是需要程序显式绑定第一个参数 self。这种机制被称为未绑定方法。

通过使用未绑定方法即可在子类中再次调用父类中被重写的方法。

```
class BaseClass:
    def foo (self):
        print(' 父类中定义的 foo 方法 ')
class SubClass(BaseClass):
    def foo (self):                    # 重写父类的 foo 方法
        print(' 子类重写父类中的 foo 方法 ')
    def bar (self):
        print(' 执行 bar 方法 ')
        self.foo()                     # 直接执行 foo 方法，将会调用子类重写之后的 foo() 方法
        BaseClass.foo(self)            # 使用类名调用实例方法（未绑定方法）调用父类被重写的方法
sc = SubClass()
sc.bar()
```

- **调用父类的构造方法（super() 函数）**

子类也会继承得到父类的构造方法，当子类有多个直接父类时，会优先选择排在最前面的父类的构造方法。为了让子类初始化时同时初始化多个父类中的实例变量，应定义子类的构造方法，即重写父类的构造方法。Python 要求，如果子类重写了父类的构造方法，那么子类的构造方法必须调用父类的构造方法。

子类的构造方法调用父类的构造方法有两种方式：

- ○ **使用未绑定方法**。因为构造方法也是实例方法，当然可以通过这种方式来调用。
- ○ **使用 super() 函数**调用父类的构造方法。

注意，当子类继承多个父类是，super() 函数只能用来调用第一个父类的构造方法，而其它父类的构造方法只能使用未绑定的方式调用。

```
class Employee :
    def __init__ (self, salary):
        self.salary = salary
    def work (self):
        print(' 普通员工正在写代码，工资是 :', self.salary)
class Customer:
    def __init__ (self, favorite, address):
        self.favorite = favorite
        self.address = address
    def info (self):
        print(' 我是一个顾客，我的爱好是 : %s, 地址是 %s' % (self.favorite, self.address))
class Manager(Employee, Customer):    # Manager 继承了 Employee、Customer
    def __init__(self, salary, favorite, address):    # 重写父类的构造方法
        print('--Manager 的构造方法 --')
        super().__init__(salary)                      # 通过 super() 函数调用父类 Employee 的构造方法
        #super(Manager, self).__init__(salary)        # 与上一行代码的效果相同
        Customer.__init__(self, favorite, address)    # 使用未绑定方法调用父类 Customer 的构造方法
m = Manager(25000, 'IT 产品 ', ' 广州 ')  # 创建 Manager 对象：--Manager 的构造方法 --
m.work()                                  # 普通员工正在写代码，工资是：2500
m.info()                                  # 我是一个顾客，我的爱好是：IT 产品，地址是广州
```

- **限制类实例动态添加属性和方法（__slots__）**

为对象动态添加方法，所添加的方法只是对当前对象有效，如果希望为所有实例都添加方法，则可通过为类添加方法来实现。

```
class Cat:
    def __init__(self, name):
        self.name = name
def walk_func(self):
    print('%s 慢慢地走过一片草地 ' % self.name)
d1 = Cat('Garfield')
d2 = Cat('Kitty')
Cat.walk = walk_func    # 为 Cat 动态添加 walk() 方法，该方法的第一个参数会自动绑定
d1.walk()               # d1 调用 walk() 方法：Garfield 慢慢地走过一片草地
d2.walk()               # d2 调用 walk() 方法：Kitty 慢慢地走过一片草地
```

Python 的这种动态性固然有其优势，但也给程序带来了一定的隐患，即程序定义好的类，完全有可能在后面被其他程序修改，这就带来了一些不确定性。如果程序要限制为某个类动态添加属性和方法，则可通过 __slots__ 属性来指定。

__slots__ 属性的值是一个元组，该元组的所有元素列出了该类的实例允许动态添加的所有属性名和方法名，但并不限制通过类来动态添加属性或方法。

```
class Dog:
    __slots__ = ('walk', 'age', 'name')        # 只允许动态为实例添加 walk、age、name 属性或方法
    def __init__(self, name):
        self.name = name
    def test():
        print(' 预先定义的 test 方法 ')
d = Dog('Snoopy')
from types import MethodType
d.walk = MethodType(lambda self: print('%s 正在慢慢地走 ' % self.name), d)
d.age = 5
d.walk()            #Snoopy 正在慢慢地走
d.foo = 30                              # foo 不在 __slots__ 中，报错：AttributeError
Dog.bar = lambda self: print('abc')         #__slots__ 不限制通过类来动态添加属性或方法
d.bar()
```

此外，__slots__ 属性指定的限制只对当前类的实例起作用，对该类派生出来的子类是不起作用的。如果要限制子类的实例动态添加属性和方法，则需要在子类中也定义 __slots__ 属性，这样，子类的实例允许动态添加属性和方法就是子类的 __slots__ 元组加上父类的 __slots__ 元组的和。

```
class GunDog(Dog):
    def __init__(self, name):
        super().__init__(name)
    pass
gd = GunDog('Puppy')
gd.speed = 99               # 完全可以为 Gundog 实例动态添加属性
print(gd.speed)
```

Python 多态

对于弱类型的语言来说，变量并没有声明类型，因此同一个变量完全可以在不同的时间引用不同的对象。当同一个变量在调用同一个方法时，完全可能呈现出多种行为（具体呈现出哪种行为由该变量所引用的对象来决定），这就是所谓的多态（Polymorphism）。

```
class Bird:
    def move(self, field):
        print(' 鸟在 %s 上自由地飞翔 ' % field)
class Dog:
    def move(self, field):
        print(' 狗在 %s 里飞快的奔跑 ' % field)
x = Bird()                  # x 变量被赋值为 Bird 对象
x.move(' 天空 ')            # 调用 x 变量的 move() 方法：鸟在天空上自由地飞翔
x = Dog()                   # x 变量被赋值为 Dog 对象
x.move(' 草地 ')            # 调用 x 变量的 move() 方法：狗在草地里飞快的奔跑
```

多态是一种非常灵活的编程机制。假如我们要定义一个 Canvas（画布）类，这个画布类定义一个 draw_pic() 方法，该方法负责绘制各种图形。至于该方法具有何种行为（到底执行怎样的绘制行为），这与 draw_pic() 方法是完全分离的，这就为编程增加了很大的灵活性。

```
class Canvas:
    def draw_pic(self, shape):
        print('-- 开始绘图 --')
        shape.draw(self)
class Rectangle:
    def draw(self, canvas):
        print(' 在 %s 上绘制矩形 ' % canvas)
class Triangle:
    def draw(self, canvas):
        print(' 在 %s 上绘制三角形 ' % canvas)
class Circle:
    def draw(self, canvas):
        print(' 在 %s 上绘制圆形 ' % canvas)
c = Canvas()
c.draw_pic(Rectangle())         # 传入 Rectangle 参数，绘制矩形
c.draw_pic(Triangle())          # 传入 Triangle 参数，绘制三角形
c.draw_pic(Circle())            # 传入 Circle 参数，绘制圆形
```

从上面这个例子可以体会到 Python 多态的优势。当程序涉及 Canvas 类的 draw_pic() 方法时，该方法所需的参数是非常灵活的，程序为该方法传入的参数对象只要具有指定方法就行，至于该方法呈现怎样的行为特征，则完全取决于对象本身，这大大提高了 draw_pic() 方法的灵活性。

Python 枚举类

在某些情况下，一个类的对象是有限且固定的，比如季节类，它只有 4 个对象；再比如行星类，目前只有 8 个对象。这种实例有限且固定的类，在 Python 中被称为枚举类。程序有两种方式来定义枚举类：

○ **直接使用 Enum 列出多个枚举值来创建枚举类。**

```
import enum
Season = enum.Enum('Season', ('SPRING', 'SUMMER', 'FALL', 'WINTER'))        # 定义 Season 枚举类
# 第一个参数是枚举类的类名；第二个参数是一个元组，用于列出所有枚举值。
print(Season.SPRING)              # 直接访问指定枚举：Season.SPRING
print(Season.SPRING.name)         # 访问枚举成员的变量名：SPRING
print(Season.SPRING.value)        # 访问枚举成员的值：1
print(Season['SUMMER'])           # 根据枚举变量名访问枚举对象： Season.SUMMER
print(Season(3))                  # 根据枚举值访问枚举对象：Season.FALL
# 遍历 Season 枚举的所有成员（__members__ 属性返回一个 dict 字典，包含了该枚举的所有枚举实例）
for name, member in Season.__members__.items():
    print(name, '=>', member, ',', member.value)
```

○ **通过继承 Enum 基类来派生枚举类**（定义更复杂的枚举，可以为枚举额外定义方法）

```
import enum
class Orientation(enum.Enum):
    # 为序列值指定 value 值
    EAST = ' 东 '
    SOUTH = ' 南 '
    WEST = ' 西 '
    NORTH = ' 北 '
    def info(self):
        print(' 这是一个代表方向【 %s 】的枚举 ' % self.value)
print(Orientation.SOUTH)
print(Orientation.SOUTH.value)
print(Orientation['WEST'])          # 通过枚举变量名访问枚举
print(Orientation(' 南 '))          # 通过枚举值来访问枚举
Orientation.EAST.info()             # 调用枚举的 info() 方法
# 遍历 Orientation 枚举的所有成员
for name, member in Orientation.__members__.items():
    print(name, '=>', member, ',', member.value)
```

枚举也是类，也可以定义构造器。为枚举定义构造器之后，在定义枚举实例时必须为构造器参数设置值。

```
import enum
class Gender(enum.Enum):
    MALE = ' 男 ', ' 阳刚之力 '
    FEMALE = ' 女 ', ' 柔顺之美 '
    def __init__(self, cn_name, desc):
        self._cn_name = cn_name
        self._desc = desc
    @property
    def desc(self):
        return self._desc
    @property
    def cn_name(self):
        return self._cn_name
print('FEMALE 的 name:', Gender.FEMALE.name)          # 访问 FEMALE 的 name：FEMALE
print('FEMALE 的 value:', Gender.FEMALE.value)        # 访问 FEMALE 的 value：(' 女 ', ' 柔顺之美 ')
print('FEMALE 的 cn_name:', Gender.FEMALE.cn_name)    # 访问自定义的 cn_name 属性：女
print('FEMALE 的 desc:', Gender.FEMALE.desc)          # 访问自定义的 desc 属性：柔顺之美
```

Python 类特殊成员（属性和方法）

在 Python 类中有些方法名、属性名的前后都添加了双下画线，这种方法、属性通常都属于 Python 的特殊方法和特殊属性，开发者可以通过重写这些方法或直接调用这些方法来实现特殊的功能。

- **__repr__ 方法：显示属性**

__repr__ 是一个非常特殊的方法，是一个“自我描述”的方法，通常用于实现：当程序员直接打印该对象时，系统将会输出该对象的“自我描述”信息，用来告诉外界该对象具有的状态信息。

```
class Apple:
    def __init__(self, color, weight):          # 实现构造器
        self.color = color;
        self.weight = weight;
    def __repr__(self):                 # 重写 __repr__ 方法，用于实现 Apple 对象的“自我描述”
        return "Apple[color=" + self.color + ", weight=" + str(self.weight) + "]"
a = Apple(" 红色 " , 5.68)
print(a)            # 打印 Apple 对象
```

- **__del__ 方法：销毁对象**

__del__ 与 __init__ 方法对应，__init__ 方法用于初始化 Python 对象，而 __del__ 则用于销毁 Python 对象，即在任何 Python 对象将要被系统回收之时，系统都会自动调用该对象的 __del__() 方法。

```
class Item:
    def __init__ (self, name, price):          # 实现构造器
        self.name = name
        self.price = price
    def __del__ (self):                        # 定义析构函数
        print('del 删除对象 ')
im = Item(' 鼠标 ', 29.8)             # 创建一个 Item 对象，将之赋给 im 变量
del im                                 # 删除 Item 对象
```

- **__dir__ 方法：列出对象的所有属性（方法）名**

__dir__ 方法用于列出对象内部的所有属性（方法）名，将会返回包含所有属性（方法）名的序列。当程序对某个对象执行 dir(object）函数时，实际上就是将该对象的 __dir__() 方法返回值进行排序，然后包装成列表。

```
class Item:
    def __init__ (self, name, price):
        self.name = name
        self.price = price
    def info ():
        pass
im = Item(' 鼠标 ', 29.8)  # 创建一个 Item 对象，将之赋给 im 变量
print(im.__dir__())        # 返回所有属性（包括方法）组成列表
print(dir(im))             # 返回所有属性（包括方法）排序之后的列表
```

- **__dict__ 属性：查看对象内部所有属性名和属性值组成的字典**

程序使用 __dict__ 属性既可查看对象的所有内部状态，也可通过字典语法来访问或修改指定属性的值。

```
class Item:
    def __init__ (self, name, price):
        self.name = name
        self.price = price
im = Item(' 鼠标 ', 28.9)
print(im.__dict__)                 # 直接输出对象的 dict 属性：{'name': ' 鼠标 ', 'price': 28.9}
print(im.__dict__['name'])         # 通过 __dict__ 访问 name 属性：鼠标
print(im.__dict__['price'])        # 通过 __dict__ 访问 price 属性：28.9
im.__dict__['name'] = ' 键盘 '
im.__dict__['price'] = 32.8
print(im.name)                     # 键盘
print(im.price)                    # 32.8
```

- setattr()、getattr()、hasattr() 函数
  - hasattr(obj, name)：检查 obj 对象是否包含名为 name 的属性或方法。
  - getattr(object, name[, default])：获取 object 对象中名为 name 的属性的属性值。
  - setattr(obj, name, value)：将 obj 对象的 name 属性设为 value。

```
class Comment:
    def __init__ (self, detail, view_times):
        self.detail = detail
        self.view_times = view_times
    def info ():
        print(" 一条简单的评论，内容是 %s" % self.detail)
c = Comment(' 疯狂 Python 讲义很不错 ', 20)
print(hasattr(c, 'detail'))          # 判断是否包含 detail： True
print(hasattr(c, 'view_times'))      # 判断是否包含 view_times： True
print(hasattr(c, 'info'))            # 判断是否包含 info： True
print(getattr(c, 'detail'))          # 获取 detail：' 疯狂 Python 讲义很不错 '
print(getattr(c, 'view_times'))      # 获取 view_times：20
print(getattr(c, info, ' 默认值 '))  # 获取 info：NameError: name 'info' is not defined（因为 info 是方法）
setattr(c, 'detail', ' 天气不错 ')   # 为指定 detail 设置值
setattr(c, 'view_times', 32)         # 为指定 view_times 设置值
print(c.detail)                      # 输出 detail 重新设置后的属性值
print(c.view_times)                  # 输出 view_times 重新设置后的属性值
```

- issubclass 和 isinstance 函数：检查类型
  - issubclass(cls, class_or_tuple)：检查 cls 是否为后一个类或元组包含的多个类中任意类的子类。
  - isinstance(obj, class_or_tuple)：检查 obj 是否为后一个类或元组包含的多个类中任意类的对象。
  - __bases__ 属性：返回所有直接父类组成的元组。
  - __subclasses__()：返回该类的所有子类组成的列表。

```
hello = "Hello"             # 定义一个字符串
print('"Hello" 是 str 类的实例 : ', isinstance(hello, str))          # "Hello" 是 str 类的实例：True
print('"Hello" 是 object 类的实例 : ', isinstance(hello, object))    # "Hello" 是 object 类的实例：True
print('str 是 object 类的子类 : ', issubclass(str, object))         # str 是 object 类的子类：True
print('"Hello" 是 tuple 类的实例 : ', isinstance(hello, tuple))     # "Hello" 是 tuple 类的实例：False
print('str 是 tuple 类的子类 : ', issubclass(str, tuple))           # str 是 tuple 类的子类：False
my_list = [2, 4]            # 定义一个列表
print('[2, 4] 是 list 类的实例 : ', isinstance(my_list, list))      # [2, 4] 是 list 类的实例：True
print('[2, 4] 是 object 类的实例 : ', isinstance(my_list, object))  # [2, 4] 是 object 类的实例： True
print('list 是 object 类的子类 : ', issubclass(list, object))       # list 是 object 类的子类：True
print('[2, 4] 是 tuple 类的实例 : ', isinstance([2, 4], tuple))     # [2, 4] 是 tuple 类的实例： False
print('list 是 tuple 类的子类 : ', issubclass(list, tuple))         # list 是 tuple 类的子类：False
class A: pass
class B: pass
class C(A, B): pass
print('A 的所有父类 :', A.__bases__)      # A 的所有父类 : (<class 'object'>,)
print('B 的所有父类 :', B.__bases__)      # B 的所有父类 : (<class 'object'>,)
print('C 的所有父类 :', C.__bases__)      # C 的所有父类 : (<class '__main__.A'>, <class '__main__.B'>)
print('A 的所有子类 :', A.__subclasses__())      #A 的所有子类 : [<class '__main__.C'>]
print('B 的所有子类 :', B.__subclasses__())      #B 的所有子类 : [<class '__main__.C'>]
```

- __call__ 属性：判断属性（或方法）是否可调用
  - __call__ 属性：判断属性（或方法）是否包含 __call__ 方法，即是否可调用。

```
class User:
    def __init__(self, name, passwd):
        self.name = name
        self.passwd = passwd
    def validLogin (self):
        print(' 验证 %s 的登录 ' % self.name)
u = User('crazyit', 'leegang')
print(hasattr(u.name, '__call__'))          # 判断 u.name 是否可调用：False
print(hasattr(u.passwd, '__call__'))        # 判断 u.passwd 是否可调用：False
print(hasattr(u.validLogin, '__call__'))    # 判断 u.validLogin 是否可调用：True
```

- 序列相关操作
  - __len__(self)：该方法的返回值决定序列中元素的个数。
  - __getitem__(self, key)：获取指定索引的元素。key 应是整数值或 slice 对象，否则引发 KeyError 异常。
  - __contains__(self, item)：判断序列是否包含指定元素。
  - __setitem__(self, key, value)：设置指定索引的元素。key 应是整数值或 slice 对象，否则引发 KeyError 异常。
  - __delitem__(self, key)：删除指定索引对应的元素。

```
def check_key (key):    #检查序列的索引会否为正整数，可引发 TypeError( 非整数 )和 IndexError( 负整数 )
    if not isinstance(key, int): raise TypeError(' 索引值必须是整数 ')
    if key < 0: raise IndexError(' 索引值必须是非负整数 ')
    if key >= 26 ** 3: raise IndexError(' 索引值不能超过 %d' % 26 ** 3)
class StringSeq:
    def __init__(self):
        self.__changed = {}          # 用于存储被修改的数据
        self.__deleted = []          # 用于存储已删除元素的索引
    def __len__(self):
        return 26 ** 3
    def __getitem__(self, key):              # 根据索引获取序列中元素
        check_key(key)
        if key in self.__changed :   # 如果在 self.__changed 中找到已经修改后的数据
            return self.__changed[key]
        if key in self.__deleted :   # 如果 key 在 self.__deleted 中，说明该元素已被删除
            return None
        three = key // (26 * 26)     # 否则根据计算规则返回序列元素
        two = ( key – three * 26 * 26) // 26
        one = key % 26
        return chr(65 + three) + chr(65 + two) + chr(65 + one)
    def __setitem__(self, key, value):       # 根据索引修改序列中元素
        check_key(key)
        self.__changed[key] = value          # 将修改的元素以 key-value 对的形式保存在 __changed 中
    def __delitem__(self, key):              # 根据索引删除序列中元素
        check_key(key)
        if key not in self.__deleted : self.__deleted.append(key)      # 添加不在 __deleted 的被删除的 key
        if key in self.__changed : del self.__changed[key]             # 删除 __changed 中包含被删除 key
sq = StringSeq()            # 创建序列
print(len(sq))              # 获取序列的长度，实际上就是返回 __len__() 方法的返回值：17576
print(sq[26*26])            #BAA
print(sq[1])                # 打印没修改之后的 sq[1]： AAB
sq[1] = 'fkit'              # 修改 sq[1] 元素
print(sq[1])                # 打印修改之后的 sq[1]：fkit
del sq[1]                   # 删除 sq[1]
print(sq[1])                # None
sq[1] = 'crazyit'           # 再次对 sq[1] 赋值
print(sq[1])                # crazyit
```

- **__iter__ 和 __reversed__：实现迭代器**
  - __iter__(self)：返回一个迭代器（iterator），必须包含一个 __next__() 方法，返回迭代器的下一个元素。
  - __reversed__(self)：为内建的 reversed() 反转函数提供支持，reversed() 函数实际上是由该方法实现的。

```
class Fibs:        # 定义一个代表斐波那契数列的迭代器
    def __init__(self, len):
        self.first = 0
        self.sec = 1
        self.__len = len
    def __next__(self):    # 定义迭代器所需的 __next__ 方法
        if self.__len == 0:  # 如果 __len__ 属性为 0，结束迭代
            raise StopIteration
        self.first, self.sec = self.sec, self.first + self.sec        # 完成数列计算
        self.__len -= 1      # 数列长度减 1
        return self.first
    def __iter__(self):      # 定义 __iter__ 方法，该方法返回迭代器
        return self
fibs = Fibs(10)              # 创建 Fibs 对象
print(next(fibs))            # 获取迭代器的下一个元素
for el in fibs:              # 使用 for 循环遍历迭代器
    print(el, end=' ')
```

- **生成器**

生成器和迭代器的功能非常相似，也提供 __next__() 方法，同样可调用内置的 next() 函数来获取生成器的下一个值，也可使用 for 循环来遍历生成器。生成器与迭代器的区别在于，迭代器通常是先定义一个迭代器类，然后通过创建实例来创建迭代器；而生成器则是先定义一个包含 yield 语句的函数，然后通过调用该函数来创建生成器。

```
def test(val, step):
    cur = 0
    for i in range(val):      # 遍历 0 ~ val
        cur += i * step       # cur 添加 i*step
        yield cur
        #yield cur 语句每次返回一个值，并暂停，需 next() 函数获取生成器的下一个值时，继续向下执行
t=test(10, 2)
print(next(t))     # 0，生成器“冻结”在 yield 处
print(next(t))     # 2，生成器再次“冻结”在 yield 处
t=test(10, 2)
for ele in t:      # 遍历
    print(ele, end=' ')
```

外部程序通过 send() 方法发送数据，生成器函数使用 yield 语句接收收据。

```
def square_gen(val):
    i = 0
    out_val = None
    while True:
        # 使用 yield 语句生成值，使用 out_val 接收 send() 方法发送的参数值
        out_val = (yield out_val ** 2) if out_val is not None else (yield i ** 2)
        # 如果程序使用 send() 方法获取下一个值，out_val 会获取 send() 方法的参数
        if out_val is not None : print("====%d" % out_val)
        i += 1
sg = square_gen(5)
print(sg.send(None))     # 第一次调用 send() 方法获取值，只能传入 None 作为参数：0
print(next(sg))          # 1
print(sg.send(9))        # 调用 send() 方法获取生成器的下一个值，参数 9 会被发送给生成器：81
print(next(sg))          # 再次调用 next() 函数获取生成器的下一个值：9
sg.throw(ValueError)     # 让生成器引发异常
sg.close()               # 关闭生成器
```

Python 异常处理机制

异常（Exceptions）：程序运行时产生错误的情况叫做异常。默认情况下，当异常发生时，程序将会终止的。Python 常见异常类型有：

异常类型	含义
AssertionError	当 assert 关键字后的条件为假时，程序运行会停止并抛出 AssertionError 异常
AttributeError	当试图访问的对象属性不存在时抛出的异常
IndexError	索引超出序列范围会引发此异常
KeyError	字典中查找一个不存在的关键字时引发此异常
NameError	尝试访问一个未声明的变量时，引发此异常
TypeError	不同类型数据之间的无效操作
ZeroDivisionError	除法运算中除数为 0 引发此异常

异常处理：为了避免程序退出，可以使用捕获异常的方式获取这个异常的名称，再通过其他的逻辑代码让程序继续运行。开发者可以使用异常处理全面地控制自己的程序。异常处理不仅能够管理正常的流程运行，还能够在程序出错时对程序进行必的处理。大大提高了程序的健壮性和人机交互的友好性。

异常处理机制：可以使程序中的异常处理代码和正常业务代间分离，保证程序代码更加优雅，并可以提高程序的健壮性。Python 的异常机制主要依赖 try 、except 、else、finally 和 raise 五个关键字：

- try 关键字后缩进的代码块简称 try 块，放置的是可能引发异常的代码；
- except 后对应的是异常类型和一个代码块，用于表明该 except 块处理这种类型的代码块；
- 多个 except 块之后可以放一个 else 块，表明程序不出现异常时还要执行 else 块；
- 最后还可跟一个 finally 块，用于回收在 try 块里打开的物理资源，异常机制会保证 finally 块总被执行；
- raise 用于引发一个实际的异常，raise 可以单独作为语句使用，引发一个具体的异常对象；

● try except 异常处理

```
try:
    可能产生异常的代码块
except [(Error1, Error2, ...) [as e]]:
    处理异常的代码块 1
except [(Error3, Error4, ...) [as e]]:
    处理异常的代码块 2
```

except 后面可以不指定具体的异常名称，表示捕获所有类型的异常。在 try 块后提供多个 except 块可以无须在异常处理块中使用 if 判断异常类型，但依然可以针对不同的异常类型提供相应的处理逻辑，从而提供更细致、更有条理的异常处理逻辑。**try except 语句的执行流程如下：**

1. 首先执行 try 中的代码块，如果执行过程中出现异常，系统会自动生成一个异常对象，并提交给 解释器，此过程被称为**引发异常**。
2. 解释器收到异常对象时，寻找处理该异常对象的 except 块，把异常对象交给 except 块处理，这个过程被称为**捕获异常**。若解释器未找不到捕获异常的 except 块，则程序运行终止，解释器退出。

```
try:
    a = int(input(" 输入被除数："))
    b = int(input(" 输入除数："))
    c = a / b
    print(" 您输入的两个数相除的结果是：", c )
except (ValueError, ArithmeticError):
    print(" 程序发生了数字格式异常、算术异常之一 ")
except :
    print(" 未知异常 ")
print(" 程序继续运行 ")
```

- **访问异常信息**

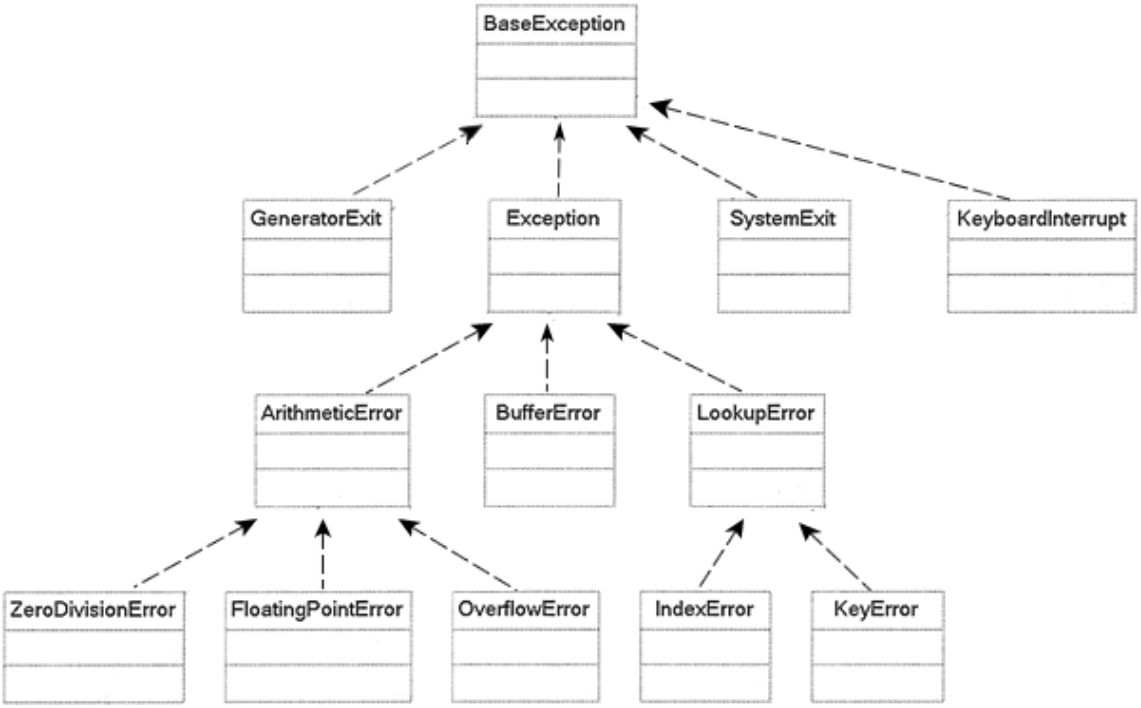
如果程序需要在 except 块中访问异常对象的相关信息，可通过为 except 块添加 as a 来实现。当解释器决定调用某个 except 块来处理该异常对象时，会将异常对象赋值给异常变量 e，即可通过该变量来获得异常对象的相关信息。所有的异常对象都包含了如下几个常用属性和方法：

- args：该属性返回异常的**错误编号和描述字符串**。
- errno：该属性返回异常的**错误编号**。
- strerror：该属性返回异常的**描述字符串**。
- with_traceback()：通过该方法可**处理异常的传播轨迹信息**。

- **异常类的继承**

解释器接收到异常对象后，会依次判断该异常对象是否是 except 块后的异常类或其子类的实例，如果是，解释器将调用该 except 块来处理该异常；否则，再次拿该异常对象和下一个 except 块里的异常类进行比较。

Python 的所有异常类都从 BaseException 派生而来，提供了丰富的异常类，不管是系统的异常类，还是用户自定义的异常类，都应该从 Exception 派生。下图显示了 Python 的常见异常类之间的继承关系。



在进行异常捕获时，不仅应该把 Exception 类对应的 except 块放在最后，而且所有父类异常的 except 块都应该排在子类异常的 except 块的后面（即：先处理小异常，再处理大异常）。

- **try except else 异常处理**

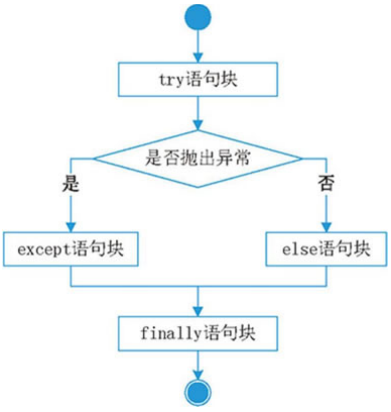
在原来 try except 语句的基础上再添加一个 else 子句和代码块，当 try 块中没有发现异常时要执行的代码。当 try 块中发现异常，则 else 块中的语句将不会被执行。else 块中的代码所引发的异常不会被 except 块捕获。所以，如果希望某段代码的异常能被后面的 except 块捕获，那么就应该将这段代码放在 try 块的代码之后；如果希望某段代码的异常能向外传播（不被 except 块捕获），那么就应该将这段代码放在 else 块中。

- **try except finally 异常处理**

finally 语句用来做清理工作的。无论 try 中的语句是否跳入 except 中，最终都要进入 finally 语句，并执行其中的代码块。

try 块是必需的，except 块和 finally 块至少出现其中之一，或同时出现。except 块可以有多个，finally 块必须位于所有的 except 块之后。尽量避免在 finally 块里使用 return 或 raise 等导致方法中止的语句，否则可能出现一些很奇怪的情况。

finally 语句块和 else 语句块的区别：else 语句块只有在没有异常发生的情况下才会执行，而 finally 语句则不管异常是否发生都会执行。不仅如此，无论是正常退出、异常退出，还是通过 break、continue、return 语句退出，finally 语句块都会执行。



- raise 语句：手动抛出一个异常

如果需要在程序中自行引发异常，可使用 raise 语句，其基本语法格式为：raise [exceptionName [(reason)]]

raise 语句有如下三种常用的用法：

1. raise：单独一个 raise。引发当前上下文中捕获的异常（比如在 except 块中）或默认 RuntimeError 异常。
2. raise 异常类名称：raise 后带一个异常类名称。引发指定异常类的默认实例。
3. raise 异常类名称 (描述信息)：在引发指定异常的同时，附带异常的描述信息。

在实际应用中对异常可能需要更复杂的处理方式。当一个异常出现时，单靠某个方法无法完全处理该异常，必须由几个方法协作才可完全处理该异常。也就是说，在异常出现的当前方法中，程序只对异常进行部分处理，还有些处理需要在该方法的调用者中才能完成，所以应该再次引发异常，让该方法的调用者也能捕获到异常。为了实现这种通过多个方法协作处理同一个异常的情形，可以在 except 块中结合 raise 语句来完成。

```
class AuctionException(Exception): pass          # 自定义异常类
class AuctionTest:
    def __init__(self, init_price):
        self.init_price = init_price
    def bid(self, bid_price):
        d = 0.0
        try:
            d = float(bid_price)
        except Exception as e:
            print(" 转换出异常：", e)          # 此处只是简单地打印异常信息
            raise AuctionException(" 竞拍价必须是数值，不能包含其他字符！ ") # 再次引发自定义异常
            raise AuctionException(e)
        if self.init_price > d:
            raise AuctionException(" 竞拍价比起拍价低，不允许竞拍！ ")
        initPrice = d
def main():
    at = AuctionTest(20.4)
    try:
        at.bid("df")
    except AuctionException as ae:
        print('main 函数捕捉的异常：', ae)  # 再次捕获到 bid() 方法中的异常，并对该异常进行处理
main()
```

- sys.exc_info() 方法：获取异常信息

在实际调试程序的过程中，有时只获得异常的类型是远远不够的，还需要借助更详细的异常信息才能解决问题。捕获异常时，有 2 种方式可获得更多的异常信息，分别是：

◎ 使用 sys 模块中的 exc_info 方法；

exc_info() 方法会将当前的异常信息以元组的形式返回，该元组中包含 3 个元素，分别是：

○ type：异常类型的名称，它是 BaseException 的子类

○ value：捕获到的异常实例。

○ traceback：是一个 traceback 对象。需调用 traceback 模块的 print_tb 方法（需先引进 traceback 模块）获得对象信息，此对象作为 print_tb 方法的参数。

```
import sys                  # 引入 sys 模块
import traceback            # 引入 traceback 模块（要使用 traceback 模块时引入）
try:
    x = int(input(" 请输入一个被除数："))  # 请输入一个被除数：0
    print("30 除以 ",x," 等于 ",30/x)
except:
    print(sys.exc_info())   或   traceback.print_tb(sys.exc_info()[2])
    print(" 其他异常 ...")
```

◎ **使用 traceback 模块中的相关函数。**traceback 模块可以用来查看异常的传播轨迹，追踪异常触发的源头（首先需要将 traceback 模块引入）。常用方法有两个：

1. traceback.print_exc()：将异常传播轨迹信息输出到控制台或指定文件中。
2. format_exc()：将异常传播轨迹信息转换成字符串。

```
import traceback          # 导入 trackback 模块
class SelfException(Exception): pass
def main():
    firstMethod()
def firstMethod():
    secondMethod()
def secondMethod():
    thirdMethod()
def thirdMethod():
    raise SelfException(" 自定义异常信息 ")
try:
    main()
except:
    traceback.print_exc()                          # 捕捉异常，并将异常传播信息输出控制台
    traceback.print_exc(file=open('log.txt', 'a'))     # 捕捉异常，并将异常传播信息输出指定文件中
```

● **异常机制使用细则** 成功的异常处理应该实现如下 4 个目标：①使程序代码混乱最小化。②捕获并保留诊断信息。③通知合适的人员。④采用合适的方式结束异常活动。为达到这些效果的基本准则：

○ **不要过度使用异常** 过度使用异常主要表现在两个方面：

1. 把异常和普通错误混淆在一起，不再编写任何错误处理代码，而是以简单地引发异常来代苦所有的错误处理。
2. 使用异常处理来代替流程控制。

○ **不要使用过于庞大的 try 块** try 块里的代码过于庞大，业务过于复杂，就会造成 try 块中出现异常的可能性大大增加，从而导致分析异常原因的难度也大大增加。正确的做法是，把大块的 try 块分割成多个可能出现异常的程序段，并把它们放在单独的 try 块中，从而分别捕获并处理异常。

○ **不要忽略捕获到的异常** 既已捕获到异常，except 块理应做些有用的事情，及处理并修复异常。except 块为空，程序出了错误，所有人都看不到任何异常，但整个应用可能已经彻底坏了。仅在 except 块里打印异常传播信息稍微好一点，但仅仅比空白多了几行异常信息。通常建议对异常采取适当措施：

» **处理异常。**对异常进行合适的修复，然后绕过异常发生的地方继续运行；或者用别的数据进行计算，以代替期望的方法返回值；或者提示用户重新操作……总之，程序应该尽量修复异常，使程序能恢复运行。

» **重新引发新异常。**把在当前运行环境下能做的事情尽量做完，然后进行异常转译，把异常包装成当前层的异常，重新传给上层调用者。

» **在合适的层处理异常。**如果当前层不清楚如何处理异常，就不要在当前层使用 except 语句来捕获该异常，让上层调用者来负责处理该异常。

● assert 条件表达式 [, 描述信息]：当条件表达式的值为真时，该语句什么也不做，程序正常运行；反之，则 assert 会抛出 AssertionError 异常。其中，[, 描述信息] 作为可选参数，用于对条件表达式可能产生的异常进行描述。assert 可以和 try except 异常处理语句配合使用：

```
try:
    s_age = input(" 请输入您的年龄 :")
    age = int(s_age)
    assert 20 < age < 80 , " 年龄不在 20–80 之间 "
    print(" 您输入的年龄在 20 和 80 之间 ")
except AssertionError as e:
    print(" 输入年龄不正确 ",e)
```

Python 模块

Python 提供了强大的模块支持，Python 标准库中包含了大量的模块（称为标准模块），还有大量的第三方模块，开发者自己也可以开发自定义模块。通过这些强大的模块可以极大地提高开发者的开发效率。

模块实际上就是 Python 程序（任何 Python 程序都可以作为模块）。一个模块（.py 文件）中可以包含多个函数。这样不仅可以提高代码的可维护性，还可以提高代码的可重用性。

举例：在某一目录下（桌面也可以）创建一个名为 hello.py 文件：

```
def say ():    print("Hello,World!")
```

在同一目录下，再创建一个 say.py 文件：

```
import hello      # 通过 import 关键字，将 hello.py 模块引入此文件
hello.say()       # 通过“模块名 . 函数”方式调用 hello.py 模块中的函数，输出：Hello,World!
```

- **import 导入模块**

◎ **导入模块的 2 种语法：**

1. **import 模块名 1 [as 别名 1], 模块名 2 [as 别名 2]，…：**导入指定模块中的所有成员（包括变量、函数、类等）。使用模块中的成员时，需用该模块名（或别名）作为前缀。

```
import sys                  # 导入 sys 整个模块
print(sys.argv[0])          # 使用模块名作为前缀来访问模块中的成员
import sys as s             # 导入 sys 整个模块，并指定别名为 s
import sys as s,os as o     # 导入 sys、os 两个模块，并为 sys 指定别名 s，为 os 指定别名 o
print(o.sep)                # 指定了别名，使用别名作为前缀来访问模块中的成员
#sys 模块下的 argv 变量用于获取运行 Python 程序的命令行参数，其中 argv[0] 用于获取当前 Python 程序的存储路径，如：C:\Users\mengma\Desktop\hello.py
#os 模块的 sep 变量代表平台上的路径分隔符，在 Windows 平台上：\
```

2. **from 模块名 import 成员名 1 [as 别名 1]，成员名 2 [as 别名 2]，…：**只导入模块中指定的成员，而不是全部成员。使用模块中的成员时，无需附加任何前缀，直接使用成员名（或别名）即可。

```
from sys import argv          # 导入 sys 模块的 argv 成员
print(argv[0])                # 使用导入成员的语法，直接使用成员名访问（无需前缀）
from sys import argv as v                  # 导入 sys 模块的 argv 成员，并为其指定别名 v
from sys import argv as v, winver as wv    # 导入 sys 模块的 argv,winver 成员，并为其指定别名 v、wv
print(wv)                     # 指定了别名，直接使用成员的别名访问
#sys 的 winver 成员记录了该 Python 的版本号，如：3.6
```

◎ **导入模块的 3 种方式：**

通常情况下，当使用 import 语句导入模块后，Python 会按照以下顺序查找指定的模块文件：

- 在**当前目录**，即当前执行的程序文件所在目录下查找；
- 到 **PYTHONPATH（环境变量）**下的每个目录中查找；
- 到 **Python 默认的安装**目录下查找。

以上所有涉及到的目录保存在标准模块 sys 的 sys.path 变量中（通过此变量，可以输出指定程序文件支持查找的所有目录），若要导入的模块所在的目录不在 sys.path 变量中，解释器会抛出 ModuleNotFoundError（未找到模块）异常。为了让 Python 能找到自定义（或第三方提供）的模块，可以用以下 3 种方式来告诉它（假设：模块 hello 的文件保存路径是 D:\python_module\hello.py）：

1. **向 sys.path 中临时添加模块文件存储位置的完整路径；**

```
import sys
sys.path.append('D:\\python_module')
```

2. 将模块放在 sys.path 变量中已包含的模块加载路径中。
3. 设置系统环境变量的用户变量 PYTHONPATH 为模块所在的目录。

◎ 导入模块的本质

使用“import fk_module”导入模块的本质就是，将 fk_module.py 中的全部代码加载到内存并执行，然后将整个模块内容赋值给与模块同名的变量，该变量的类型是 module，而在该模块中定义的所有程序单元都相当于该 module 对象的成员。当程序重复导入同一个模块时，Python 只会导入一次。

◎ __all__ 变量

__all__ 变量为模块定义了一个开放的公共接口，只有 __all__ 变量列出的成员，才是希望该模块被外界使用的成员。如果确实使用模块内 __all__ 列表之外的成员，有两种解决方法：

- 第一种是使用“import 模块名”来导入模块。在通过这种方式导入模块之后，总可以通过模块名前缀（如果为模块指定了别名，则可以使用模块的别名作为前缀）来调用模块内的成员。
- 第二种是使用“from 模块名 import 成员”来导入指定成员。在这种方式下，即使想导入的成员没有位于 __all__ 列表中，也依然可以导入。

● 自定义模块

模块文件的文件名就是它的模块名，比如 module1.py 的模块名就是 module1。

当自定义模块编写完成之后，需要编写一些测试代码，检验模块中各个功能是否都能正常运行。借助 Python 内置的 __name__ 变量，可以使直接运行模块文件（相当于测试文件），程序会执行该模块的测试函数；如果是其他程序导入该模块，程序不应该执行该模块的测试函数。

在实际开发中应该为模块编写说明文档；编写说明文档很简单，只要在模块开头位置定义一个字符串。

```
'''
这是我们编写的第一个模块，该模块包含以下内容：
my_book：字符串变量
say_hi：简单的函数
User：代表用户的类
'''
my_book = 'Python 入门教程 '
def say_hi(user):
    print('%s, 您好，欢迎学习 Python' % user)
class User:
    def __init__(self, name):
        self.name = name
    def walk(self):
        print('%s 正在慢慢地走路 ' % self.name)
    def __repr__(self):
        return 'User[name=%s]' % self.name
# === 以下是测试代码 ===
if __name__=='__main__':
    def test_my_book ():        print(my_book)
    def test_say_hi():        say_hi(' 孙悟空 ')
    def test_User():
        u = User(' 白骨精 ')
        u.walk()
        print(u)
    test_my_book()
    test_say_hi()
    test_User()
```

if __name__ == '__main__' 的作用是确保只有单独运行该模块时，此表达式才成立，才可以进入此判断语法，执行其中的测试代码；反之，如果只是作为模块导入到其他程序文件中，则此表达式将不成立，运行其它程序时，也就不会执行该判断语句中的测试代码。

Python 包

包是一个包含多个模块的文件夹（它的本质依然是模块，因此包中也可以包含包），每个包的目录下都必须建立一个 __init__.py 的模块（这是 Python 的规定，可以是一个空模块，可以写一些初始化代码），其作用就是告诉 Python 要将该目录当成包来处理。

- **包的创建** 主要分为如下两步：
  1. 创建一个文件夹，该文件夹的名字就是该包的包名。如：first_package
  2. 在该文件夹内创建一个名为 __init__.py 的 Python 文件，在此文件中可以不编写任何代码，也可以编写一些 Python 初始化代码（其他程序文件导入包时，会自动执行），并添加模块文件。

```
# __init__.py 文件
'''
这是学习包的第一个示例
'''
print('this is first_package')
# module1.py 模块文件
display(n):
    print(n)
# module2.py 模块文件
class Item:
    display(self):
        print(" 调用模块中类的方法 ")
```

- **包的导入** 和模块导入的语法非常类似，可归结为以下 3 种：
  1. import 包名 [. 模块名 [as 别名]] 导入包中的模块
     使用模块中的成员（变量、函数、类）时，以 **包名 . 模块名** 为前缀。

```
import first_package.module1
first_package.module1.display(20)
```

  2. from 包名 import 模块名 [as 别名] 导入包中的模块
     使用模块中的成员（变量、函数、类）时，不需要带包名前缀，但需要带**模块名**前缀。

```
from first_package import module1
module1.display(20)
```

  3. from 包名 . 模块名 import 成员名 [as 别名] 导入包中模块的成员
     使用成员（变量、函数、类）时，可直接使用成员名（函数名、类名）调用。

```
from first_package.module1 import display
display(20)
```

- __init__.py 的作用

导入包就等同于导入该包中的 __init__.py 文件，因此完全可以在 __init__.py 文件中直接编写实现模块功能的变量、函数和类，推荐大家在 __init__.py 文件中导入该包内的其他模块。

```
# __init__.py 文件可通过如下 2 种方式来导入包中模块的：
from . import module1    # 从当前包导入 module1 模块，使用成员时加“包名 . 模块名”前缀
from .module2 import *  # 从当前包中 module2 模块的所有成员，使用成员时加“包名”前缀
```

Python 库

库相比模块和包，是一个更大的概念，例如在 Python 标准库中的每个库都有好多个包，而每个包中都有若干个模块。

除了使用 Python 内置的标准模块以及我们自定义的模块之外，还有很多第三方模块可以使用，这些第三方模块可以借助 Python 官方提供的查找包页面（https://pypi.org/）找到。

使用第三方模块之前，需要先下载并安装该模块，然后就能像使用标准模块和自定义模块那样导入并使用了。下载和安装第三方模块，可以使用 Python 提供的 pip 命令实现。pip 命令的语法格式如下：

pip install|uninstall|list 模块名

其中 install：用于安装第三方模块，当 pip 使用 install 作为参数时，后面的模块名不能省略。

uninstall：用于卸载已经安装的第三方模块，选择 uninstall 作为参数时，后面的模块名也不能省略。

list：用于显示已经安装的第三方模块。

Python 模块信息

在导入模块之后，开发者往往需要了解模块包含哪些功能，比如包含哪些变量、哪些函数、哪些类等，还希望能查看模块中各成员的帮助信息，掌握这些信息才能正常地使用该模块。

- **dir() 函数　查看模块（变量、函数、类）成员**

dir() 函数的基本用法，该函数可用于返回模块或类所包含的全部成员（包括变量、函数、类和方法等），但直接使用 dir() 函数默认会列出模块内所有的成员，包括以下画线开头的成员，而这些以下画线开头的成员其实并不希望被外界使用。为了过滤这些成员，可以使用表推导式（以 string 为例）：

```
>>> [e for e in dir(string) if not e.startswith('_')]
```

此外，还可以查看模块中的 __all__ 变量来查看模块内的成员：

```
>>> string.__all__
```

注意，并不是所有模块都会提供 __all__ 变量的，有些模块并不提供 __all__ 变量，在这种情况下，只能使用列表推导式来查看模块中的成员。

- **__doc__ 属性　查看文档**

可使用 help() 函数来查看指定程序单元的帮助信息。如查看 string 模块下 capwords() 函数的作用：

```
>>> help(string.capwords)
```

使用 help() 函数查看到的帮助信息其实就是程序单元的文档信息，即程序单元的 __doc__ 属性值。

```
>>> print(string.capwords.__doc__)
```

从理论上说，应该为每个程序单元都编写完备而详细的文档信息，这样开发者只要通过 help() 函数即可查看该程序单元的文档信息，完全不需要查看文档。

- **__file__ 属性　查看模块的源文件路径**

通过模块的 __file__ 属性即可查看到指定模块的源文件路径。如查看 string 模块对应的文件路径：

```
>>> import string
>>> string.__file__
'D:\\python3.6\\lib\\string.py'
```

这说明 string 模块对应的文件就是 D:\Python\Python36\lib\string.py。

需要说明的是，并不是所有模块都是使用 Python 语言编写的，有些与底层交互的模块可能是用 C 语言编写的，而且是 C 程序编译之后的效果，因此这种模块可能没有 file 属性。

Python 常见模块

Python 语言本身也内置了大量模块，对于常规的日期、时间、正则表达式、JSON 支持、容器类等，内置的模块已经非常完备。Python 内置的模块总是在不断的更新中，更详细、更完备的模块介绍文档可参考 Python 库的参考手册：https://docs.python.org/3/library/index.html。

1. Python sys

sys 是一个和 Python 解释器关系密切的标准库，能帮助我们访问和 Python 解释器联系紧密的变量和函数。

成员（变量和函数）	功能描述
sys.argv	获取运行 Python 程序的命令行参数。其中 sys.argv[0] 通常就是指该 Python 程序，sys.argv[1] 代表为 Python 程序提供的第一个参数，sys.argv[2] 代表为 Python 程序提供的第二个参数……依此类推。
sys.path	是一个字符串列表，其中每个字符串都是一个目录名，在使用 import 语句导入模块时，解释器就会从这些目录中查找指定的模块。
sys.exit()	通过引发 SystemExit 异常来退出程序。将其放在 try 块中不能阻止 finally 块的执行。你可以提供一个整数作为参数（默认为 0，标识成功），用来标识程序是否成功运行，这是 UNIX 的一个惯例。
sys.modules	返回模块名和载入模块对应关系的字典。
sys.platform	字符串变量，标识解释器当前正在运行的平台名称，可能是操作系统的名称，也可能是其他种类的平台，如果运行 Jython 的话，就是 Java 虚拟机。
sys.stdin、sys.stdout、sys.stderr	这三个模块变量是类文件流对象，分别表示标准输入、标准输出和标准错误。简单理解，Python 利用 sys.stdin 获得输入，利用 sys.stdout 输出。
sys.flags	该只读属性返回运行 Python 命令时指定的旗标。
sys.getfilesystemencoding()	返回在当前系统中保存文件所用的字符集
sys.getrefcount(object)	返回指定对象的引用计数。当对象的引用计数为 0 时，系统会回收该对象。
sys.getrecursionlimit()	返回 Python 解释器当前支持的递归深度。该属性可通过 setrecursionlimit() 方法重新设置。
sys.getswitchinterval()	返回在当前 Python 解释器中线程切换的时间间隔。该属性可通过 setswitchinterval() 函数改变。
sys.implementation	返回当前 Python 解释器的实现。
sys.maxsize	返回 Python 整数支持的最大值。在 32 位平台上，该属性值为 2**31−1；在 64 位平台上，该属性值为 2**63−1。
sys.executable	该属性返回 Python 解释器在磁盘上的存储路径。
sys.byteorder	显示本地字节序的指示符。如果本地字节序是大端模式，则该属性返回 big；否则返回 little。
sys.copyright	该属性返回与 Python 解释器有关的版权信息。
sys.version	返回当前 Python 解释器的版本信息。
sys.winve	返回当前 Python 解释器的主版本号。

2. Python os

os 模块代表了程序所在的操作系统，主要用于获取程序运行所在操作系统的相关信息。

成员（变量和函数）	功能描述
os.name	返回导入依赖模块的操作系统名称，通常可返回 'posix'、'nt'、 'java' 等值其中之一。
os.environ	返回在当前系统上所有环境变量组成的字典。
os.sep	返回路径分隔符。
os.fsencode(filename)	该函数对类路径（path−like）的文件名进行编码。
os.fsdecode(filename)	该函数对类路径（path−like）的文件名进行解码。
os.PathLike	这是一个类，代表一个类路径（path−like）对象。
os.getenv(key, default=None)	获取指定环境变量的值。

os.getlogin()	返回当前系统的登录用户名（只在 UNIX 系统有效）。对应的还有 os.getuid()、os.getgroups()、os.getgid() 等，用于获取用户 ID、用户组、组 ID 等。
os.getpid()	获取当前进程 ID。
os.getppid()	获取当前进程的父进程 ID。
os.putenv(key, value)	该函数用于设置环境变量。
os.cpu_count()	返回当前系统的 CPU 数量。
os.pathsep	返回当前系统上多条路径之间的分隔符。一般在 Windows 系统上多条路径之间的分隔符是英文分号（;）；在 UNIX 及类 UNIX 系统（如 Linux、Mac os X）上多条路径之间的分隔符是英文冒号（:）。
os.linesep	返回当前系统的换行符。一般在 Windows 系统上换行符是“\r\n”；在 UNIX 系统上换行符是“\n”；在 Mac os X 系统上换行符是“\r”。
os.urandom(size)	返回适合作为加密使用的、最多由 N 个字节组成的 bytes 对象。该函数通过操作系统特定的随机性来源返回随机字节，该随机字节通常是不可预测的，因此适用于绝大部分加密场景。
os.system(command)	运行操作系统上的指定命令。
os.abort()	生成一个 SIGABRT 信号给当前进程。在 UNIX 系统上，默认行为是生成内核转储；在 Windows 系统上，进程立即返回退出代码 3。
os.execl(path, arg0, arg1, ...)	该函数还有一系列功能类似的函数，如 os.execle()、os.execlp() 等，都是使用参数列表 arg0, arg1,... 来执行 path 所代表的执行文件的。由于 os.execl() 函数是 PosIX 系统的直接映射，故使用该命令来执行 Python 程序，传入的 arg0 参数没有什么作用。os._exit(n) 用于强制退出 Python 解释器。将其放在 try 决中可以阻止 finally 块的执行。
os.forkpty()	fork 一个子进程。
os.kill(pid, sig)	将 sig 信号发送到 pid 对应的过程，用于结束该进程。
os.killpg(pgid, sig)	将 sig 信号发送到 pgid 对应的进程组。
os.popen(cmd, mode='r', buffering=−1)	用于向 cmd 命令打开读写管道（当 mode 为 r 时为只读管道，当 mode 为 rw 时为读写管道），buffering 缓冲参数与内置的 open() 函数有相同的含义。该函数返回的文件对象用于读写字符串，而不是字节。
os.spawnl(mode, path, ...)	该函数还有一系列功能类似的函数，比如 os.spawnle()、os.spawnlp() 等，这些函数都用于在新进程中执行新程序。
os.startfile(path[,operation])	对指定文件使用该文件关联的工具执行 operation 对应的操作。若不指定 operation 操作，则默认执行打开（open）操作。operation 参数必须是有效的命令行操作项目，比如 open（打开）、edit（编辑）、print（打印）等。

3. Python random

random 模块包括返回随机数的函数，可以用于模拟或者任何产生随机输出的程序。

成员（变量和函数）	功能描述
random.seed(a=None, version=2)	指定种子来初始化伪随机数生成器。
random.randrange(start, stop[, step])	返回从 start 开始到 stop 结束、步长为 step 的随机数。实就相当于 choice(range(start, stop, step)) 的效果，只是实际底层并不生成区间对象。
random.randint(a, b)	生成一个范围为 a ≤ N ≤ b 的随机数。等同于 randrange(a, b+1) 的效果。
random.choice(seq)	从 seq 中随机抽取一个元素，如果 seq 为空，则引发 IndexError 异常。
random.choices(seq, weights=None, cum_weights=None, k=1)	从 seq 序列中抽取 k 个元素，还可通过 weights 指定各元素被抽取的权重（代表被抽取的可能性高低）。
random.shuffle(x[, random])	对 x 序列执行洗牌“随机排列”操作。
random.sample(population, k)	从 population 序列中随机抽取 k 个独立的元素。
random.random()	生成一个从 0.0（包含）到 1.0（不包含）之间的伪随机浮点数。
random.uniform(a, b)	生成一个范围为 a ≤ N ≤ b 的随机数。
random.expovariate(lambd)	生成呈指数分布的随机数。其中 lambd 参数（其实应该是 lambda，只是 lambda 是 Python 关键字，所以简写成 lambd）为 1 除以期望平均值。如果 lambd 是正值，则返回的随机数是从 0 到正无穷大；如果 lambd 为负值，则返回的随机数是从负无穷大到 0。

4. Python time

time 模块主要包含各种提供日期、时间功能的类和函数。提供日期、时间和字符串之间的互相转换功能。

成员（变量和函数）	功能描述
time.asctime([t])	将时间元组或 struct_time 转换为时间字符串。如果不指定参数 t，则默认转换当前时间。
time.ctime([secs])	将以秒数代表的时间（格林威治时间）转换为时间字符串。
time.gmtime([secs])	将以秒数代表的时间转换为 struct_time 对象。如果不传入参数，则使用当前时间。
time.localtime([secs])	将以秒数代表的时间转换为代表当前时间的 struct_time 对象。如果不传入参数，则使用当前时间。
time.mktime(t)	它是 localtime 的反转函数，用于将 struct_time 对象或元组代表的时间转换为从 1970 年 1 月 1 日 0 点整到现在过了多少秒。
time.perf_counter()	返回性能计数器的值。以秒为单位。
time.process_time()	返回当前进程使用 CPU 的时间，以秒为单位。
time.sleep(secs)	暂停 secs 秒，什么都不干。
time.strftime(format[, t])	将时间元组或 struct_time 对象格式化为指定格式的时间字符串。如果不指定参数 t，则默认转换当前时间。
time.strptime(string[, format])	将字符串格式的时间解析成 struct_time 对象。
time.time()	返回从 1970 年 1 月 1 日 0 点整到现在过了多少秒。
time.timezone	返回本地时区的时间偏移，以秒为单位。
time.tzname	返回本地时区的名字。

time 模块内提供了一个代表一个时间 time.struct_time 类，包含 9 个属性：

字段名	含义	值	字段名	含义	值	字段名	含义	值
tm_year	年	如 2019	tm_hour	时	范围为 0~23	tm_wday	周	范围为 0~6
tm_mon	月	范围为 1~12	tm_min	分	范围为 0~59	tm_yday	一年内第几天	范围 1~366
tm_mday	日	范围为 1~31	tm_sec	秒	范围为 0~61	tm_isdst	夏令时	0、1 或 −1

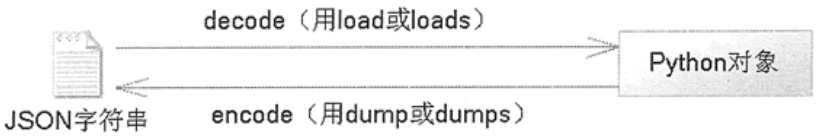
time 模块中的 strftime() 和 strptime() 两个函数互为逆函数，strftime() 用于将 struct_time 对象或时间元组转换为时间字符串；而 strptime() 函数用于将时间字符串转换为 struct_time 对象。这两个函数都涉及编写格式模板，格式模板所需要的时间格式字符串支持的指令如下表：

指令	含义	指令	含义
%a	本地化的星期缩写名，如 Sun 代表星期天	%A	本地化的星期几的完整名
%b	本地化的月份缩写名，比如 Jan 代表一月	%B	本地化的月份的完整名
%c	本地化的日期和时间的表示形式	%d	一个月中第几天的数值，范围 01~31
%H	24 小时制的小时，范围 00~23	%I	12 小时制的小时，范围 01~12
%j	一年中第几天，范围 001~366	%m	月份的数值，范围 01~12
%M	分钟的数值，范围 00~59	%S	秒钟的数值，范围 00~61。60 为闰秒，61 无效
%p	上午或下午的本地化方式。当使用 strptime() 函数并使用 %I 指令解析小时时，%p 只影响小时字段	%w	星期几的数值，范围 0~6，其中 0 代表周日
%U	一年中的第几周，周日为每周的第一天，范围 00~53。一年中第一个周日被认为处于第一周。当使用 strptime() 函数解析时间字符串时，只有同时指定了星期几和年份该指令才会有效	%W	一年中的第几周，周一为每周的第一天，范围 00~53。一年中第一个周一被认为处于第一周。当使用 strptime() 函数解析时间字符串时，只有同时指定了星期几和年份该指令才会有效
%x	本地化的日期的表示形式	%X	本地化的时间的表示形式
%y	年份的缩写，范围 00~99，如 2018 年为 18	%Y	年份的完整形式。如 2018
%z	显示时区偏移	%Z	时区名（如果时区不行在，则显示为空）
%%	用于代表 % 符号		

5. Python json

json 模块提供了对 JSON 的支持，包含 JSON 字符串和 Python 对象之间的互相转换函数。

Python 类型转换 --> JSON 字符串		JSON 字符串 --> Python 类型
Python 类型	JSON 字符串	Python 类型
字典（dict）	对象（object）	字典（dict）
列表（list）和元组（tuple）	数组（array）	列表（list）
字符串（str）	字符串（string	字符串（str）
整型、浮点数，以及整型、浮点型派生的枚举（float,int–& float–derived Enums）	数值型（number） 整数（number(int)）	整数（int）
	数值型（number） 实数（number(real)）	浮点数（float）
True	true	True
False	false	False
None	null	None



成员（变量和函数）	功能描述
json.JSONEncoder().encode(obj)	将 Python 对象转换为 JSON 字符串
json.load(fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)	从 fp 流读取 JSON 字符串，转换成 Python 对象，（fp 是一个支持 write() 方法的类文件对象）
json.loads(s, *, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)	将 JSON 字符串 s 转换成 Python 对象并返回该 Python 对象
json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)	将 Python 对象转换成 JSON 字符串输出到 fp 流中（fp 是一个支持 write() 方法的类文件对象）
json.dumps(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan= True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)	将 Python 对象转换为 JSON 字符串并返回该 JSON 字符串

将 Python 对象转换成 JSON 字符串（dumps() 和 dump() 函数的 encode 操作）

```
import json                                                    # 导入 json
s1= json.dumps(['yeeku', {'favorite': ('coding', None, 'game', 25)}])    # 元祖会当成数组
s2 = json.dumps("\"foo\bar")                                   # 简单的 Python 字符串
s3 = json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True)      # 对 key 排序
s4 = json.dumps([1, 2, 3, {'x': 5, 'y': 7}], separators=(',', ':'))    # 并指定 JSON 分隔符
s5 = json.dumps({'Python': 5, 'Kotlin': 7}, sort_keys=True, indent=4)  #JSON 字符串指定缩进 4
s6 = json.JSONEncoder().encode({"names": (" 孙悟空 ", " 齐天大圣 ")})  #JSONEncoder 的 encode 方法
f = open('a.json', 'w');json.dump(['Kotlin', {'Python': 'excellent'}], f)    #JSON 字符串输出到文件
```

将 JSON 字符串转换成 Python 对象（loads() 和 load() 函数的 decode 操作）

```
import json                                                    # 导入 json
result1 = json.loads('["yeeku", {"favorite": ["coding", null, "game", 25]}]')
result2 = json.loads('"\\"foo\\"bar"')
def as_complex(dct):     # 自定义一个复数转化函数，供 loads 使用
    if '__complex__' in dct:
        return complex(dct['real'], dct['imag'])
    return dct
result3 = json.loads('{"__complex__": true, "real": 1, "imag": 2}',object_hook=as_complex)
f = open('a.json');result4 = json.load(f)                      # 从文件流恢复 JSON 列表
```

6. Python re

成员（变量和函数）	功能描述
re.compile(pattern, flags=0)	将正则表达式字符串编译成 _sre.SRE_Pattern 对象（正则表达式编译之后在内存中的对象），它可以缓存并复用正则表达式字符串。如果程序需要多次使用同一个正则表达式字符串，则可考虑先编译它。
re.match(pattern, string, flags=0)	从字符串的开始位置来匹配正则表达式。匹配不成功，返回 None；匹配成功，返回 _sre.SRE_Match 对象，对象的 span(n) 方法用于获取第 n+1 个组的匹配位置，group(n) 方法用于获取第 n+1 个组所匹配的子串。
re.search(pattern, string, flags=0)	扫描整个字符串，并返回字符串中第一处匹配 pattern 的匹配对象。与 search() 的区别在于 search() 可以搜索整个字符串。
re.findall(pattern, string, flags=0)	扫描整个字符串，并返回字符串中所有匹配 pattern 的子串组成的列表。
re.finditer(pattern, string, flags=0)	扫描整个字符串，并返回字符串中所有匹配 pattern 的子串组成的迭代器，迭代器的元素是 _sre.SRE_Match 对象。
re.fullmatch(pattem, string, flags=0)	要求整个字符串能匹配 pattern，如果匹配则返回包含匹配信息的 _sre.SRE_Match 对象；否则返回 None。
re.sub(pattern, repl, string, count=0, flags=0)	将 string 字符串中所有匹配 pattern 的内容替换成 repl；repl 既可是被替换的字符串，也可是一个函数。count 控制最多替换多少次，0 表示全部替换。
re.split(pattem, string, maxsplit=0, flags=0)	使用 pattern 对 string 进行分割，该函数返回分割得到的多个子串组成的列表。maxsplit 控制最多分割几次。
re.purge()	清除正则表达式缓存
re.escape(pattern)	对模式中除 ASCII 字符、数值、下画线（_）之外的其他字符进行转义。

re 模块中的 **Match 对象**（其具体类型为 _sre.SRE_Match）则是 match()、search() 方法的返回值，该对象中包含了详细的正则表达式匹配信息，包括正则表达式匹配的位置、正则表达式所匹配的子串。

sre.SRE_Match 的方法或属性	功能描述
match.group([group1,...])	获取该匹配对象中指定组所匹配的字符串。
match.__getitem__(g)	match.group(g) 的简写。
match.groups(default=None)	返回 match 对象中所有组所匹配的字符串组成的元组。
match.groupdict(default= None)	返回 match 对象中所有组所匹配的字符串组成的字典。
match.start([group])	获取该匹配对象中指定组所匹配的字符串的开始位置。
match.end([group])	获取该匹配对象中指定组所匹配的字符串的结束位置。
match.span([group])	获取该匹配对象中指定组所匹配的字符串的开始位置和结束位置。该方法相当于同时返回 start() 和 end() 方法的返回值。
match.pos	返回传给正则表达式对象的 search()、match() 等方法的 pos 参数。
match.endpos	返回传给正则表达式对象的 search()、match() 等方法的 endpos 参数。
match.lastindex	返回最后一个匹配的捕获组的整数索引。若没有组匹配，返回 None。
match.lastgroup	返回最后一个匹配的捕获组的名字；若没有名字或组匹配，返回 None。
match.re	返回执行正则表达式匹配时所用的正则表达式。
match.string	返回执行正则表达式匹配时所用的字符串。

7. Python set 和 frozenset　set 是无序的不重复元素集合。frozenset 是 set 的不可变版本

set 运算符	功能描述
<=	相当于调用 issubset() 方法，判断前面的 set 集合是否为后面的 set 集合的子集合。
>=	相当于调用 issuperset() 方法，判断前面的 set 集合是否为后面的 set 集合的父集合。
–	相当于调用 difference() 方法，用前面的 set 集合减去后面的 set 集合的元素。
&	相当于调用 intersection() 方法，用于获取两个 set 集舍的交集
^	计算两个集合异或的结果，就是用两个集合的并集减去交集的元素。
交集运算	intersection() 不改变集合本身，返回两个集合的交集；intersection_update() 改变第一个集合。
并集运算	union() 不改变集合本身，而是返回两个集合的并集；update() 改变第一个集合。
减法运算	difference() 不改变集合本身，返回两个集合的成法结果；difference_update() 改变第一个集合。

8. Python queue

堆栈是一种特殊的线性表，栈一端被称为栈顶（top），另一端则被称为栈底（bottom）。只允许在栈顶进行插入（**进栈** push）、删除（**出栈** pop）操作。因此栈是一种**后进先出**（LIFO）的线性表。

队列也是一种特殊的线性表，只允许在表的前端（**队头** front）进行删除操作，在表的后端（**队尾** rear）进行插入操作。因此，队列是一种**先进先出**（FIFO）的线性表。

双端队列（queue）是一种特殊的队列，可以在两端同时进行插入、删除操作。如果程序将所有的插入、删除操作都固定在一端进行，那么这个双端队列就变成了堆栈；如果固定在一端只添加元素，在另一端只删除元素，那么它就变成了队列。因此，deque 既可被当成队列使用，也可被当成栈使用。

deque 的左边（left）就相当于队列头（front），右边（right）就相当于队列尾（rear）。

成员（变量和函数）	功能描述
append	在 deque 的右边添加元素
appendleft	在 deque 的左边添加元素
pop	在 deque 的右边弹出元素
popleft	在 deque 的左边弹出元素
extend	在 deque 的右边添加多个元素
extendleft	在 deque 的左边添加多个元素
rotate	将队列的队尾元素移动到队头，使之首尾相连
clear	清空队列
insert	插入元素

把 deque 当成**堆栈**使用

```
from collections import deque                  # 导入 deque，deque 位于 collections 包中
stack = deque(('Kotlin', 'Python'))            # 定义堆栈 stack
stack.append('Erlang')                         # 元素 Erlang 入栈
stack.append('Swift')                          # 元素 Swift 入栈
print('stack 中的元素：', stack)                # stack 中的元素： deque(['Kotlin', 'Python', 'Erlang', 'Swift'])
print(stack.pop())                             # 元素出栈： Swift
print(stack.pop())                             # 元素出栈：Erlang
print(stack)                                   # deque(['Kotlin', 'Python'])
```

把 deque 当成**队列**使用

```
from collections import deque                  # 导入 deque，deque 位于 collections 包中
q = deque(('Kotlin', 'Python'))                # 定义队列
q.append('Erlang')                             # 元素 Erlang 入列
q.append('Swift')                              # 元素 Swift 入列
print('q 中的元素：', stack)                    # q 中的元素： deque(['Kotlin', 'Python', 'Erlang', 'Swift'])
print(q.popleft())                             # 元素出列： Kotlin
print(q.popleft())                             # 元素出列：Python
print(q)                                       # deque(['Erlang', 'Swift'])
```

把 deque 的 rotate() 方法示例

```
from collections import deque                  # 导入 deque，deque 位于 collections 包中
q = deque(range(5))                            # 定义队列
print('q 中的元素：', stack)                    # q 中的元素： deque([0, 1, 2, 3, 4])
q.rotate()                                     # 执行旋转，使之首尾相连
print('q 中的元素：', stack)                    # q 中的元素： deque([4, 0, 1, 2, 3])
q.rotate()                                     # 执行旋转，使之首尾相连
print('q 中的元素：', stack)                    # q 中的元素： deque([3, 4, 0, 1, 2])
```

9. Python heapq

小顶堆（小根堆）：n 个数据元素顺序排成的完全二叉树，树的所有节点值都小于其左、右节点的值，此树的根节点的值必然最小。小顶堆的任意子树也是小顶堆。

大顶堆（大根堆）：n 个数据元素顺序排成的完全二叉树，树的所有节点值都大于其左、右节点的值，此树的根节点的值必然最大。大顶堆的任意子树还是大顶堆。

Python 并没有提供“堆”这种数据类型，它是直接把列表当成堆处理的。**heapq 包**中有一些函数，当程序用这些函数来操作列表时，该列表就会表现出“堆”的行为。

成员（变量和函数）	功能描述
heappush(heap, item)	将 item 元素加入堆。
heappop(heap)	将堆中最小元素弹出。
heapify(heap)	将堆属性应用到列表上。
heapreplace(heap, x)	将堆中最小元素弹出，并将元素 x 入堆。
merge(*iterables, key=None, reverse=False)	将多个有序的堆合并成一个大的有序堆，然后再输出。
heappushpop(heap, item)	将 item 入堆，然后弹出并返回堆中最小的元素。
nlargest(n, iterable, key=None)	返回堆中最大的 n 个元素。
nsmallest(n, iterable, key=None)	返回堆中最小的 n 个元素。

```
from heapq import *                    # 导入 heapq 中的所有函数
my_data = list(range(10))              # 定义 my_data 列表
my_data.append(0.5)                    # 添加 0.5
print('my_data 元素：', my_data)          # my_data 元素：[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0.5]（为列表）
heapify(my_data)                       # 对 my_data 应用堆属性
print(' 应用堆后的：', my_data)           # 应用堆后： [0, 0.5, 2, 3, 1, 5, 6, 7, 8, 9, 4]
heappush(my_data, 7.2)                 # 将 7.2 加入堆
print(' 添加 7.2 后：', my_data)          # 添加 7.2 后： [0, 0.5, 2, 3, 1, 5, 6, 7, 8, 9, 4, 7.2]
print(heappop(my_data))                # 弹出堆中最小的元素： 0
print(heappop(my_data))                # 弹出堆中最小的元素： 0.5
print(' 弹出两个元素后：', my_data)       # 弹出两个元素： [1, 3, 2, 7, 4, 5, 6, 7.2, 8, 9]
print(heapreplace(my_data, 8.1))       # 弹出堆中最小的元素，压入 8.1： 1
print(' 执行 replace 后：', my_data)      # 执行 replace 后： [2, 3, 5, 7, 4, 8.1, 6, 7.2, 8, 9]
print(' 最大的 3 个元素：', nlargest(3, my_data))      # 最大的 3 个元素： [9, 8.1, 8]
print(' 最小的 4 个元素：', nsmallest(4, my_data))     # 最小的 4 个元素： [2, 3, 4, 5]
```

10. Python ChainMap

collections 包下的 ChainMap 是一个方便的工具类，使用链的方式将多个 dict“链”在一起（实际上底层并未真正合并这些 dict），从而允许程序可直接获取任意一个 dict 所包含的 key 对应的 value。当多个 dict 中具有重复的 key 时，排在“链”前面的 dict 中的 key 具有更高的优先级。

```
from collections import ChainMap       # 导入 ChainMap（在 collections 包中）
a = {'Kotlin': 90, 'Python': 86}       # 定义 dict 对象 a
b = {'Go': 93, 'Python': 92}           # 定义 dict 对象 b
c = {'Swift': 89, 'Go': 87}            # 定义 dict 对象 c
cm = ChainMap(a, b , c)                # 将 3 个 dict 对象链在一起，就像变成了一个大的 dict
print(cm)      # ChainMap({'Kotlin': 90, 'Python': 86}, {'Go': 93, 'Python': 92}, {'Swift': 89, 'Go': 87})
print(cm['Kotlin'])                    # 获取 Kotlin 对应的 value：90（唯一）
print(cm['Python'])                    # 获取 Python 对应的 value：86（a 的级别高于 b，a 中的值）
print(cm['Go'])                        # 添获取 Go 对应的 value：93（b 的级别高于 c，b 中的值）
```

ChainMap 可用于**局部定义、全局定义、内置定义**的变量的正确取值；还可用于**指定参数、系统环境变量、系统默认值**的正确取值。

11. Python Counter

collections 包下的 Counter 也是一个很有用的工具类，它可以自动统计容器中各元素出现的次数。其本质是一个特殊的 dict，key 是其所包含的元素，value 记录该 key 出现的次数。不存在的 key，其 value 为 0。

```
from collections import Counter                    # 导入 Counter（在 collections 包中）
c1 = Counter()                                     # 创建空的 Counter
c2 = Counter('hannah')                             # 以可迭代对象创建 Counter
c3 = Counter(['Python', 'Swift', 'Swift', 'Python', 'Kotlin', 'Python'])  # 以可迭代对象创建 Counter
c4 = Counter({'red': 4, 'blue': 2})                # 以 dict 来创建 Counter
c5 = Counter(Python=4, Swift=8)                    # 使用关键字参数创建 Counter
```

Counter 继承 dict 类，可以调用 dict 支持的方法。Counter 还提供了如下三个常用的方法：

Counter 方法	功能描述
elements()	返回 Counter 所包含的全部元素组成的迭代器。
most_common([n])	返回 Counter 中出现最多的 n 个元素。
subtract([iterable-or-mapping])	计算 Counter 的减法，就是计算减去之后各元素出现的次数。

```
from collections import Counter                    # 导入 Counter（在 collections 包中）
cnt = Counter()                                    # 创建空的 Counter
print(cnt['Python'])                               # 访问并不存在的 key，将输出该 key 的次数为：0
for word in ['Swift', 'Python', 'Kotlin', 'Kotlin', 'Swift', 'Go']:  cnt[word] += 1
print(cnt)                          # Counter({'Swift': 2, 'Kotlin': 2, 'Python': 1, 'Go': 1})
print(list(cnt.elements()))         # 访问 Counter 对象元素：['Swift', 'Swift', 'Python', 'Kotlin', 'Kotlin', 'Go']
chr_cnt = Counter('abracadabra')                   # 将字符串（迭代器）转换成 Counter
print(chr_cnt.most_common(3))                      # 获取出现最多的 3 个字母：[('a', 5), ('b', 2), ('r', 2)]
c = Counter(a=4, b=2, c=0, d=-2)                   # Counter({'a': 4, 'b': 2, 'c': 0, 'd': -2})
d = Counter(a=1, b=2, c=3, d=4)                    # Counter({'d': 4, 'c': 3, 'b': 2, 'a': 1})
c.subtract(d)                                      # 用 Counter 对象执行减法，其实就是减少各元素的出现次数
print(c)                                           # Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
e = Counter({'x': 2, 'y': 3, 'z': -4})             # Counter({'y': 3, 'x': 2, 'z': -4})
del e['y']                                         # 调用 del 删除 key-value 对，会真正删除该 key-value 对
print(e)                                           # Counter({'x': 2, 'z': -4})
print(e['w'])                                      # 访问 'w' 对应的 value，'w' 没有出现过，因此返回：0
del e['w']                                         # 删除 e['w']，删除该 key-value 对
print(e['w'])                                      # 再次访问 'w' 对应的 value，'w' 还是没有，因此返回：0
```

此外，Counter 对象还有一些很常用的操作　设 c = Counter(Python=4, Swift=2, Kotlin=3, Go=-2)
c1 = Counter(a=3, b=1, c=-1)
c2 = Counter(a=1, b=-2, d=3)

操作	语法	描述
转换成集合	set(c)	只保留 key：{'Python', 'Swift', 'Kotlin', 'Go'}
转换成列表	list(c)	只保留 key：['Python', 'Swift', 'Kotlin', 'Go']
转换成字典	dict(c)	{'Python': 4, 'Swift': 2, 'Kotlin': 3, 'Go': -2}
加	c1 + c2	各 key 出现的次数相加，且只保留出现次数为正的元素：Counter({'a': 4, 'd': 3})
减	c1 - c2	各 key 出现的次数相减，且只保留出现次数为正的元素：Counter({'b': 3, 'a': 2})
交	c1 & c2	取都出现的 key，且各 key 对应的次数的最小数：Counter({'a': 1})
并	c1 | c2	取全部的 key，且各 key 对应的出现次数的最大数：Counter({'a': 3, 'd': 3, 'b': 1})
求正	+c	保留 value>=0 的 key-value：Counter({'Python': 4, 'Kotlin': 3, 'Swift': 2})
求负	-c	保留 value<0 的 key-value 对，value 改为正数。Counter({'Go': 2})

12. Python defaultdict

defaultdict 是 dict 的子类，与 dict 最大的区别在于，访问 dict 中不存在的 key 的 value 时会引发 KeyError 异常；而 defaultdict 会提供一个 default_factory 属性，该属性所指定的函数负责为该 key 生成 value 值。

```
from collections import defaultdict          # 导入 defaultdict（在 collections 包中）
s = [('Python', 1), ('Swift', 2), ('Python', 3), ('Swift', 4), ('Python', 9)]
d = defaultdict(list)                        # 创建 defaultdict，设置由 list() 函数来生成默认值
for k, v in s:
    d[k].append(v)          # 访问 defaultdict 中 key 的 value，若 key 不存在，自动为该 key 生成默认值
print(list(d.items()))      # [('Python', [1, 3, 9]), ('Swift', [2, 4])]
```

13. Python namedtuple

namedtuple() 是一个工厂函数，使用该函数可以创建一个 tuple（元组）类的子类，该子类可以为 tuple（元组）的每个元素都指定字段名，这样程序就可以根据字段名来访问 namedtuple 的各元素了。当然，如果有需要，程序依然可以根据索引来访问 namedtuple 的各元素。

namedtuple 是轻量级的，性能很好，其并不比普通 tuple 需要更多的内存。函数的语法格式如下：

namedtuple(typename, field_names, *, verbose=False, rename=False, module=None)

- typename：指定所创建的 tuple 子类的类名，相当于用户定义了一个新类。
- field_names：字段名字符串序列（如 ['x', 'y']）或字段名字符串（多个字段名用空格、逗号隔开，如 'x y' 或 'x,y'）。字段名可由字母、数字、下画线组成（不能以数字、下画线开头，也不能是关键字）。
- rename：如果将该参数设为 True，那么无效的字段名将会被自动替换为位置名。例如指定 ['abc','def','ghi','abc']，它将会被替换为 ['abc', '_1','ghi','_3']（def 字段名是关键字，abc 字段名重复）。
- verbose：如果设为 True，当该子类被创建之后，该类定义就会被立即打印出来。
- module：如果设置了该参数，那么该类将位于该模块下，因此该自定义类的 __module__ 属性将被设为该参数值。

```
from collections import namedtuple      # 导入 namedtuple（在 collections 包中）
Point = namedtuple('Point', ['x', 'y'])      # 定义命名元组类：Point
p = Point(11, y=22)      # 初始化 Point 对象，即可用位置参数，也可用命名参数
print(p[0] + p[1])       # 像普通元组一样用根据索引访问元素：33
a, b = p                 # 执行元组解包，按元素的位置解包
print(a, b)              # 11, 22
print(p.x + p.y)         # 根据字段名访问各元素：33
print(p)                 # Point(x=11, y=22)
```

方法	功能描述
_make(iterable)	类方法。用于根据序列或可迭代对象创建命名元组对象。
_asdict()	将当前命名元组对象转换为 OrderedDict 字典。
_replace(**kwargs)	替换命名元组中一个或多个字段的值。
_source	返回定义该命名元组的源代码。
_fields	返回该命名元组中所有字段名组成的元组。

```
p2 = Point._make(['East', 'North'])          # 创建命名元组对象：Point(x='East', y='North')
p2._asdict()    # 将命名元组对象转换成 OrderedDict：OrderedDict([('x', 'East'), ('y', 'North')])
p2._replace(y='South')                       # 替换命名元组对象的字段值： Point(x='East', y='North')
print(p2._fields)                            # 输出 p2 包含的所有字段： ('x', 'y')
Color = namedtuple('Color', 'red green blue')                # 定义命名元组类 Color
Pixel = namedtuple('Pixel', Point._fields + Color._fields)   # Pixel 类的字段为 Point 字段加 Color 字段
pix = Pixel(11, 22, 128, 255, 0)             # 创建 Pixel 对象，分别为 x、y、red、green、blue 字段赋值
print(pix)                                   # Pixel(x=11, y=22, red=128, green=255, blue=0)
```

14. Python OrderedDict

OrderedDict 也是 dict 的子类，其最大特征是可以“维护”添加 key-value 对的顺序。因此即使两个 OrderedDict 中的 key-value 对完全相同，只要它们的顺序不同，它们就不相等。

```
from collections import OrderedDict          # 导入 OrderedDict（在 collections 包中）
dx = OrderedDict(b=5, c=2, a=7)              # 创建 OrderedDict 对象
print(dx)                                    # OrderedDict([('b', 5), ('c', 2), ('a', 7)])
d = OrderedDict()                            # 创建空的 OrderedDict 对象
d['Python'] = 89                             # 向 OrderedDict 中添加 key-value 对
d['Swift'] = 92                              # 向 OrderedDict 中添加 key-value 对
d['Kotlin'] = 97                             # 向 OrderedDict 中添加 key-value 对
d['Go'] = 87                                 # 向 OrderedDict 中添加 key-value 对
for k,v in d.items():                        # 遍历 OrderedDict 的 key-value 对
    print(k, v,end=" ")                      # Python 89 Swift 92 Kotlin 97 Go 87
my_data = {'Python': 20, 'Swift':32, 'Kotlin': 43, 'Go': 25}  # 普通的 dict 对象
d1 = OrderedDict(sorted(my_data.items(), key=lambda t: t[0]))    # 基于 key 排序的 OrderedDict
d2 = OrderedDict(sorted(my_data.items(), key=lambda t: t[1]))    # 基于 value 排序的 OrderedDict
print(d1)          # OrderedDict([('Go', 25), ('Kotlin', 43), ('Python', 20), ('Swift', 32)])
print(d2)          # OrderedDict([('Python', 20), ('Go', 25), ('Swift', 32), ('Kotlin', 43)])
print(d1 == d2)    #key-value 对完全相同，顺序不同，所以不等：false
```

方法	功能描述
popitem(last=True)	last=True 时（默认），弹出并返回最右边（最后加入）的 key-value 对； last=False 时，弹出并返回最左边（最先加入）的 key-value 对。
move_to_end(key, last=True)	last=True 时（默认），将指定的 key-value 对移动到最右边（最后加入）； ast=False 时，将指定的 key-value 对移动到最左边（最先加入）。

```
from collections import OrderedDict          # 导入 OrderedDict（在 collections 包中）
d = OrderedDict.fromkeys('abcd')             # OrderedDict([('a', None), ('b', None), ('c', None), ('d', None)])
d.move_to_end('b')                           # 将 b 对应的 key-value 对移动到最右边（最后加入）
print(d.keys())                              # odict_keys(['a', 'c', 'd', 'b'])
d.move_to_end('b', last=False)               # 将 b 对应的 key-value 对移动到最左边（最先加入）
print(d.keys())                              # odict_keys(['b', 'a', 'c', 'd'])
print(d.popitem()[0])                        # 弹出并返回最右边（最后加入）的 key-value 对：d
print(d.popitem(last=False)[0])              # 弹出并返回最左边（最先加入）的 key-value 对：b
print(d.keys())                              # odict_keys(['a', 'c'])
```

15. Python itertools

itertools 模块中主要包含了一些用于生成迭代器的函数。itertools 模块中生成迭代器的函数有：

函数	功能描述
count(start, [step])	生成 start, start+step, start+2*step，... 的迭代器，step 默认为 1。如： count(10) 生成：10, 11, 12, 13, 14, ...；count(10, 3) 生成：10, 13, 16, 19, 22, ...。
cycle(p)	对序列 p 生成无限循环 p0, p1,..., p0, p1,... 的迭代器。如： cycle('ABCD') 生成：A, B, C, D, A, B, C, D, ...。
repeat(elem [,n])	生成无限个 elem 元素重复的迭代器，如果指定了 n，则生成 n 个 elem 元素。如： repeat(10) 生成：10, 10, 10, 10, 10, ...；repeat(10, 3) 生成：10, 10, 10。
accumulate(p[,func])	生成根据序列 p 元素累加的迭代器，p0, p0+p1, p0+p1+p2，... 序列， 如果指定了 func 函数，则用 func 函数来计算下一个元素的值。如： it.accumulate(range(6)) 生成：0, 1, 3, 6, 10, 15 it.accumulate(range(1, 6), lambda x, y: x * y) 生成：1, 2, 6, 24, 120
chain(p, q, ...)	将多个序列里的元素“链”在一起生成新的序列。如： it.chain(['a', 'b'], ['Kotlin', 'Swift']) 生成：'a', 'b', 'Kotlin', 'Swift'
compress(data, selectors)	根据 selectors 序列的值对 data 序列的元素进行过滤。如果 selector[0] 为真，则保留 data[0]；如果 selector[1] 为真，则保留 data[1]..... 依此类推。如： it.compress(['a', 'b', 'Kotlin', 'Swift'], [0, 1, 1, 0]) 生成：'b', 'Kotlin'
dropwhile(pred, seq)	使用 pred 函数对 seq 序列进行过滤，从 seq 中第一个使用 pred 函数计算为 False 的元素开始，保留从该元素到序列结束的全部元素。如： it.dropwhile(lambda x:len(x)<4, ['a', 'b', 'Kotlin', 'x', 'y']) 生成：'Kotlin', 'x', 'y'
takewhile(pred, seq)	使用 pred 函数对 seq 序列进行过滤，从 seq 中第一个使用 pred 函数计算为 False 的元素开始，去掉从该元素到序列结束的全部元素。如： it.takewhile(lambda x:len(x)<4, ['a', 'b', 'Kotlin', 'x', 'y']) 生成：'a', 'b'
filterfalse(pred, seq)	使用 pred 函数对 seq 序列进行过滤，保留 seq 中使用 pred 计算为 True 的元素。如： it.filterfalse(lambda x:len(x)<4, ['a', 'b', 'Kotlin', 'x', 'y']) 生成：'Kotlin'
islice(seq, [start,] stop [, step])	其功能类似于序列的 slice 方法，实际上就是返回 seq[start:stop:step] 的结果。如： it.islice(range(10),2,7,2) 生成：2, 4, 6
starmap(func, seq)	使用 func 对 seq 序列的每个元素进行计算，将计算结果作为新的序列元素。如： it.starmap(pow, [(2,5), (3,2), (10,3)]) 生成：32, 9, 1000
zip_longest(p,q,...)	将 p、q 等序列中的元素按索引合并成元组，这些元组将作为新序列的元素。如： it.zip_longest('ABCD', 'xy', fillvalue='-') 生成：('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')
product(p, q, ...[repeat= 1)]	用序列 p、q、... 中的元素进行排列组合，就相当于使用嵌套循环组合。如： it.product('AB', 'CD') 生成 AC, AD, BC, BD it.product('AB', repeat=2) 生成 AA, AB, BA, BB
permutations(p[, r])	从序列 p 中取出 r 个元素组成全排列，将排列得到的元组作为新迭代器的元素。如： it.permutations('ABCD', 2) 生成：# AB, AC, AD, BA, BC, BD, CA, CB, CD, DA, DB, DC
combinations(p, r)	从序列 p 中取出 r 个元素组成全组合，元素不允许重复，将组合得到的元组作为新迭代器的元素。如：it.combinations('ABCD', 2) 生成：AB, AC, AD, BC, BD, CD
combinations with_ replacement(p, r)	从序列 p 中取出 r 个元素组成全组合，元素允许重复，将组合得到的元组作为新迭代器的元素。如：it.combinations_with_replacement('ABCD', 2) 生成：AA, AB, AC, AD, BB, BC, BD, CC, CD, DD

16. Python functools

functools 模块中主要包含了一些函数装饰器和便捷的功能函数。

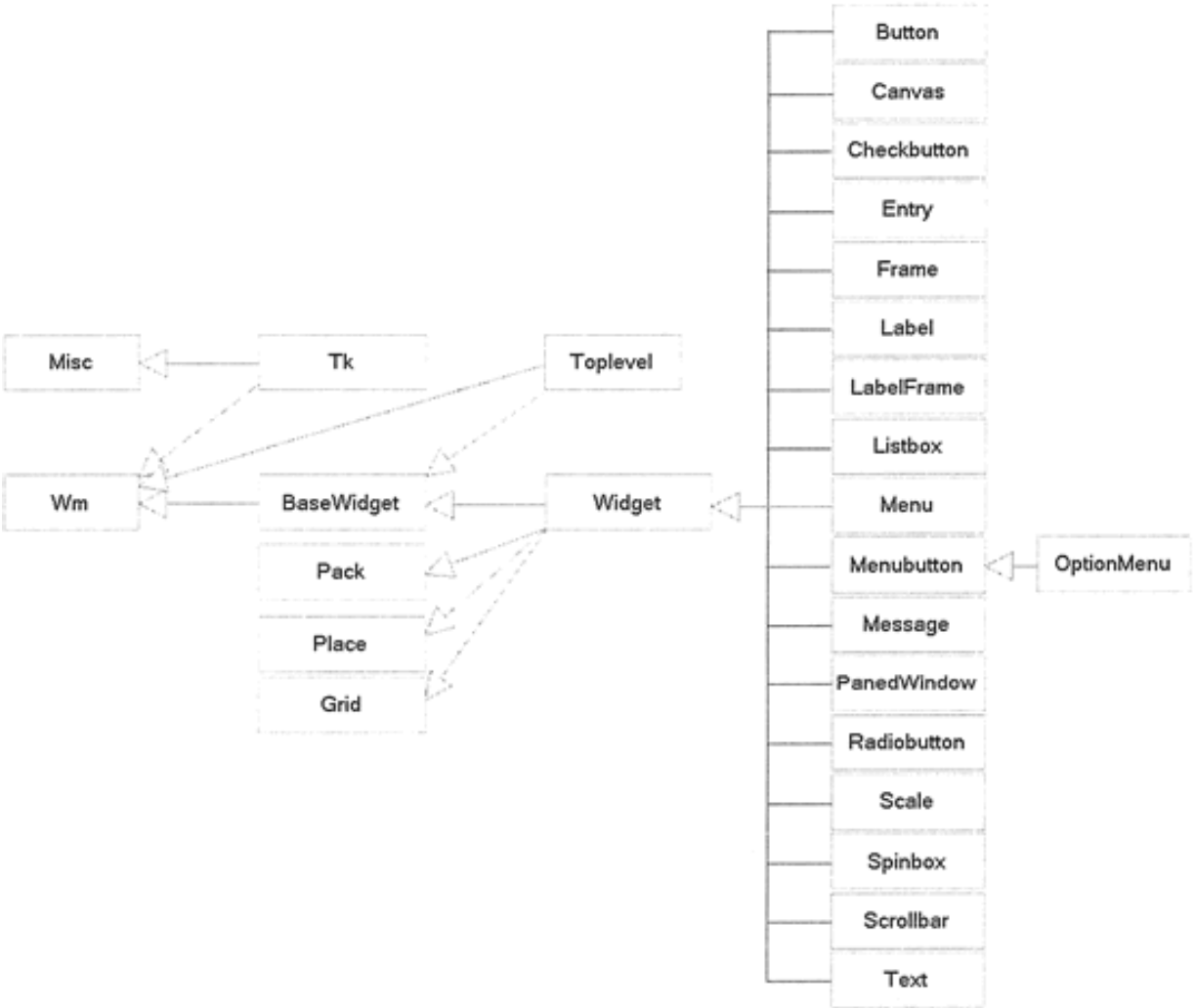
函数	功能描述
cmp_to_key(func)	将老式的比较函数（func）转换为关键字函数（key function）。在 Python 3 中比较大小、排序都是基于关键字函数的，Python 3 不支持老式的比较函数。
@lru_cache(maxsize=128, typed=False)	该函数装饰器使用 LRU（最近最少使用）缓存算法来缓存相对耗时的函数结果，避免传入相同的参数重复计算。同时，缓存并不会无限增长，不用的缓存会被释放。其中 maxsize 参数用于设置缓存占用的最大字节数，typed 参数用于设置将不同类型的缓存结果分开存放。
@total_ordering	这个类装饰器（作用类似于函数装饰器，只是它用于修饰类）用于为类自动生成比较方法。通常来说，开发者只要提供 __lt__()、__le__()、__gt__()、__ge__() 其中之一（最好能提供 __eq__() 方法），@functools.total_ordering 装饰器就会为该类生成剩下的比较方法。
partial(func, *args, **keywords)	该函数用于为 func 函数的部分参数指定参数值，从而得到一个转换后的函数，程序以后调用转换后的函数时，就可以少传入那些已指定值的参数。
partialmethod(func, *args, **keywords)	该函数与 partial 函数的含义完全相同，只不过该函数用于为类中的方法设置参数值。
reduce(function, iterable[, initializer])	将初始值（由 initializer 参数指定，默认 0）、迭代器的当前元素传入 function 函数，将计算出来的函数结果作为下一次计算的初始值、迭代器的下一个元素再次调用 function 函数……依此类推，直到迭代器的最后一个元素。
@ singledispatch	该函数装饰器用于实现函数对多个类型进行重载。比如同样的函数名称，为不同的参数类型提供不同的功能实现。该函数的本质就是根据参数类型的变换，将函数转向调用不同的函数。
update_wrapper 函数 和 @ wraps 装饰器	functools.update_wrapper(wrapper, wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)（函数）和 @functools.wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)（装饰器）的功能都是对 wrapper 函数进行包装，使之看上去就像 wrapped（被包装）函数。

Python Tkinter GUI

Python GUI 库（Graphics User Interface, 图形用户界面）有很多（如下），推荐使用 PyQt 或 wsPython。

- PyGObject 库为基于 GObject 的 C 函数库提供了内省绑定，可以支持 GTK+3 图形界面工具集，提供了丰富的图形界面组件；
- PyGTK 基于老版本的 GTK+2 的库提供绑定，借助于底层 GTK+2 所提供的各种可视化元素和组件，同样可以开发出在 GNOME 桌面系统上运行的软件，因此它主要适用于 Linux/UNIX 系统。PyGTK 对 GTK+2 的 C 语言进行了简单封装，提供了面向对象的编程接口。其官方网址是 http://www.pygtk.org/。
- PyQt 是 Python 语言和 Qt 库的成功融合。Qt 本身是一个扩展的 C++ GUI 应用开发框架，可以在 UNIX、Windows 和 Mac OS X 上完美运行，PyQt 是建立在 Qt 基础上的 Python 包装，也能跨平台使用。
- PySide 是由 Nokia 提供的对 Qt 工具集的新的包装库，目前成熟度不如 PyQt。
- wxPython 是一个跨平台的 GUI 工具集，以流行的 wxWidgets（原名 wxWindows）为基础，提供了良好的跨平台外观。wxPython 在 Windows 上调用 Windows 的本地组件、在 Mac OS 上调用 Mac OS X 的本地组件、在 Linux 上调用 Linux 的本地组件，这样让 GUI 程序在不同的平台上显示平台对应的风格。wxPython 是一个非常流行的跨平台的 GUI 库。其官方网址是 http://www.wxpython.org/。
- Tkinter 库为 Python 自带的 GUI 库，无须进行额外的下载安装，只要导入 tkinter 包即可使用。

Tkinter 的 GUI 组件之间的继承关系如图所示：



Tkinter 的 GUI 组件有两个根父类：Misc（所有组件的根父类）和 Wm（提供与窗口管理器通信的功能函数），都直接继承 object 类。Tk（Misc 和 Wm 派生的子类）代表应用程序的主窗口。BaseWidget 是所有组件的基类。

Widget （BaseWidget 的子类）代表一个通用的 GUI 组件，Tkinter 所有的 GUI 组件都是 Widget 的子类。Widget 一共有四个父类，除 BaseWidget 之外，还有 Pack、Place 和 Grid，这三个父类都是布局管理器，负责管理所包含的组件的大小和位置。Widget 子类都是 Tkinter GUI 编程的各种 UI 组件，各 GUI 组件的功能如表：

Tkinter 类	名称	简介
Toplevel	顶层	容器类，可用于为其他组件提供单独的容器；Toplevel 有点类似于窗口
Button	按钮	代表按钮组件
Canvas	画布	提供绘图功能，包括绘制直线、矩形、椭圆、多边形、位图等
Checkbutton	复选框	可供用户勾选的复选框
Entry	单行输入框	用户可输入内容
Frame	容器	用于装载其它 GUI 组件
Label	标签	用于显示不可编辑的文本或图标
LabelFrame	容器	也是容器组件，类似于 Frame，但它支持添加标题
Listbox	列表框	列出多个选项，供用户选择
Menu	菜单	菜单组件
Menubutton	菜单按钮	用来包含菜单的按钮（包括下拉式、层叠式等）
OptionMenu	菜单按钮	Menubutton 的子类，也代表菜单按钮，可通过按钮打开一个菜单
Message	消息框	类似于标签，但可以显示多行文本；后来当 Label 也能显示多行文本之后，该组件基本处于废弃状态
PanedWindow	分区窗口	该容器会被划分成多个区域，每添加一个组件占一个区域，用户可通过拖动分隔线来改变各区域的大小
Radiobutton	单选钮	可供用户点边的单选钮
Scale	滑动条	拖动滑块可设定起始值和结束值，可显示当前位置的精确值
Spinbox	微调选择器	用户可通过该组件的向上、向下箭头选择不同的值
Scrollbar	滚动条	用于为组件（文本域、画布、列表框、文本框）提供滚动功能
Text	多行文本框	显示多行文本

创建一个窗口（直接使用 Tk）：

```
from tkinter import *      # 导入 tkinter 中的所有函数
root = Tk()                # 创建 Tk 对象，Tk 代表窗口
root.title(' 窗口标题 ')    # 设置窗口标题
w = Label(root, text="Hello Tkinter!")    # 创建 Label 对象，第一个参数指定该 Label 放入 root
w.pack()                   # 调用 pack 进行布局
root.mainloop()            # 启动主窗口的消息循环
```

创建一个窗口（创建 Frame 的子类，子类就会自动创建 Tk 对象作为窗口）：

```
from tkinter import *      # 导入 tkinter 中的所有函数
class Application(Frame):        # 定义继承 Frame 的 Application 类
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.initWidgets()    # 调用 initWidgets() 方法初始化界面
    def initWidgets(self):
        w = Label(self)                # 创建 Label 对象，第一个参数指定该 Label 放入 root
        bm = PhotoImage(file = 'logo.png')            # 创建一个位图
        w.x = bm               # 必须用一个不会被释放的变量引用该图片，否则该图片会被回收
        w['image'] = bm        # 设置显示的图片是 bm
        w.pack()
        okButton = Button(self, text=" 确定 ",background = 'yellow')   # 创建 Button 对象，放入 root
        okButton.pack()
app = Application()                # 创建 Application 对象，对象默认的 master 属性是 Tk 对象（窗口）
app.master.title('Python 窗口 ')   # 通过 master 属性来设置窗口标题
app.mainloop()                     # 启动主窗口的消息循环
```

大部分 GUI 组件都支持的选项：

选项名（别名）	含义	单位	典型值
activebackground	指定组件处于激活状态时的背景色	color	'gray25' 或 '#ff4400'
activeforeground	指定组件处于激活状态时的前景色	color	'gray25' 或 '#ff4400'
anchor	内部信息如何显示。可选：N、NE、E、SE、S、SW、W、NW 或 CENTER。字母含义为 N（上北），S（下南），W（左西），E（右东）		CENTER
background(bg)	指定组件正常显示时的背景色	color	'gray25' 或 '#ff4400'
bitmap	指定显示的位图（Tk_GetBitmap 位图）。覆盖优先级为 image 覆盖 bitmap 覆盖 text。		
borderwidth	正常显示时的 3D 边框宽度（Tk_GetPixels 格式）	pixel	2
cursor	光标样式（Tk_GetCursors 格式）	cursor	gumby
command	关联的命令方法		
disabledforeground	禁用状态时的前景色	color	'gray25' 或 '#ff4400'
font	文本字体	font	'Helvetica' 或 ('Verdana', 8)
foreground(fg)	正常显示时的前景色	color	'gray' 或 '#ff4400'
highlightbackground	高亮状态下的背景色	color	'gray' 或 '#ff4400'
highlightcolor	高亮状态下的前景色	color	'gray' 或 '#ff4400'
highlightthickness	高亮状态下的周围宽度（Tk_GetPixels 格式）	pixel	2
height	高度（以 font 字体字符高度为单位，至少为 1）	integer	14
image	指定显示的图像（image 覆盖 bitmap 覆盖 text）	image	
justify	内部内容的对齐方式：LEFT（左对齐）、CENTER（居中对齐）或 RIGHT（右对齐）	constant	RIGHT
padx	内部水平方向上的空白（Tk_GctPixels 格式）	pixel	12
pady	内部垂直方向上的空白（Tk_GctPixels 格式）	pixel	12
relief	指定内部相对于外部的 3D 效果：RAISED、SUNKEN、FLAT、RIDGE、SOLID、GROOVE。	constant	GROOVE RAISED
selectbackground	选中状态下的背景色	color	'gray' 或 '#ff4400'
selectborderwidth	选中状态下的 3D 边框宽度（Tk_GetPixels 格式）	pixel	2
selectforeground	选中状态下的前景色	color	'gray' 或 '#ff4400'
state	当前状态：NORMAL（正常）、DISABLE（禁用）	constant	NORMAL
takefocus	键盘遍历（Tab 或 Shift+Tab）时是否接收焦点：1（接收焦点），0（不接收焦点）	boolean	1 或 YES
text	显示的文本	str	' 确定 '
textvariable	变量名（组件显示变量的值）	variable	bnText
underline	组件文本的第几个字符添加下画线，相当于为组件绑定了快捷键	integer	2
width	宽度（以 font 字体字符宽度为单位，至少为 1）	integer	14
wraplength	对于能支持字符换行的组件，指定每行显示的最大字符数，超过该数量，字符将自动换行	integer	20
xscrollcommand	用于将组件的水平滚动改变（包括内容滚动或宽度发生改变）与水平滚动条的 set 方法关联，从而让组件的水平滚动改变传递到水平滚动条	function	scroll.set
yscrollcommand	用于将组件的垂直滚动改变（包括内容滚动或高度发生改变）与垂直滚动条的 set 方法关联，从而让组件的垂直滚动改变传递到垂直滚动条	function	scroll.set

Pack 布局管理器

布局管理器负责管理各组件的大小和位置的。当用户调整窗口大小后，布局管理器会自动调整窗口中各组件的大小和位置。

Pack 布局是优先的布局方式，pack() 方法通常可支持如下选项：

- anchor：当可用空间大于组件所需求的大小时，该选项决定组件被放置在容器的位置。该选项支持 N（上北）、E（右东）、S（下南）、W（左西）、NW（左上西北）、NE（右上东北）、SW（左下西南）、SE（右下东南）、CENTER（默认值中）。
- expand：该 bool 值指定当父容器增大时，是否拉伸组件。
- fill：设置组件是否沿水平或垂直方向填充。该选项支持 NONE（不填充）、X（沿着 x 方向填充）、Y（沿着 y 方向填充）、BOTH（沿着两个方向填充）四个值。
- ipadx：指定组件在 x 方向（水平）上的内部留白（padding）。
- ipady：指定组件在 y 方向（水平）上的内部留白（padding）。
- padx：指定组件在 x 方向（水平）上与其他组件的间距。
- pady：指定组件在 y 方向（水平）上与其他组件的间距。
- side：设置组件的添加位置，可以设置为 TOP、BOTTOM、LEFT 或 RIGHT 这四个值的其中之一。

当程序界面比较复杂时，就需要使用多个容器（Frame）分开布局，然后再将 Frame 添加到窗口中。

```
from tkinter import *
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        fm1 = Frame(self.master)                                    # 创建第一个容器
        fm1.pack(side=LEFT, fill=BOTH, expand=YES)      # 该容器放在左边排列
        # 向 fm1 中添加 3 个按钮，按钮从顶部开始排列，只能在垂直（X）方向填充
        Button(fm1, text=' 第一个 ').pack(side=TOP, fill=X, expand=YES)
        Button(fm1, text=' 第二个 ').pack(side=TOP, fill=X, expand=YES)
        Button(fm1, text=' 第三个 ').pack(side=TOP,  fill=X, expand=YES)
        fm2 = Frame(self.master)                                    # 创建第二个容器
        fm2.pack(side=LEFT, padx=10, fill=BOTH, expand=YES)    # 该容器放在左边排列，会挨着 fm1
        # 向 fm2 中添加 3 个按钮，设置按钮从右边开始排列
        Button(fm2, text=' 第一个 ').pack(side=RIGHT, fill=Y, expand=YES)
        Button(fm2, text=' 第二个 ').pack(side=RIGHT, fill=Y, expand=YES)
        Button(fm2, text=' 第三个 ').pack(side=RIGHT, fill=Y, expand=YES)
        fm3 = Frame(self.master)                                              # 创建第三个容器
        fm3.pack(side=RIGHT, padx=10, fill=BOTH, expand=YES)  # 该容器放在右边排列，会挨着 fm2
        # 向 fm3 中添加 3 个按钮，设置按钮从底部开始排列，且按钮只能在垂直（Y）方向填充
        Button(fm3, text=' 第一个 ').pack(side=BOTTOM, fill=Y, expand=YES)
        Button(fm3, text=' 第二个 ').pack(side=BOTTOM, fill=Y, expand=YES)
        Button(fm3, text=' 第三个 ').pack(side=BOTTOM, fill=Y, expand=YES)
root = Tk()
root.title("Pack 布局 ")
display = App(root)
root.mainloop()
```

对于打算使用 Pack 布局的开发者来说，首先要做的是将程序界面进行分解，分解成水平排列的容器和垂直排列的容器（有时候甚至要容器嵌套容器），然后使用多个 Pack 布局的容器将它们组合在一起。

Grid 布局管理器

Grid 布局不仅简单易用，而且管理组件也非常方便。Grid 把组件空间分解成一个网格进行维护，即按照行、列的方式排列组件，组件位置由其所在的行号和列号决定，行号相同而列号不同的几个组件会被依次上下排列，列号相同而行号不同的几个组件会被依次左右排列。

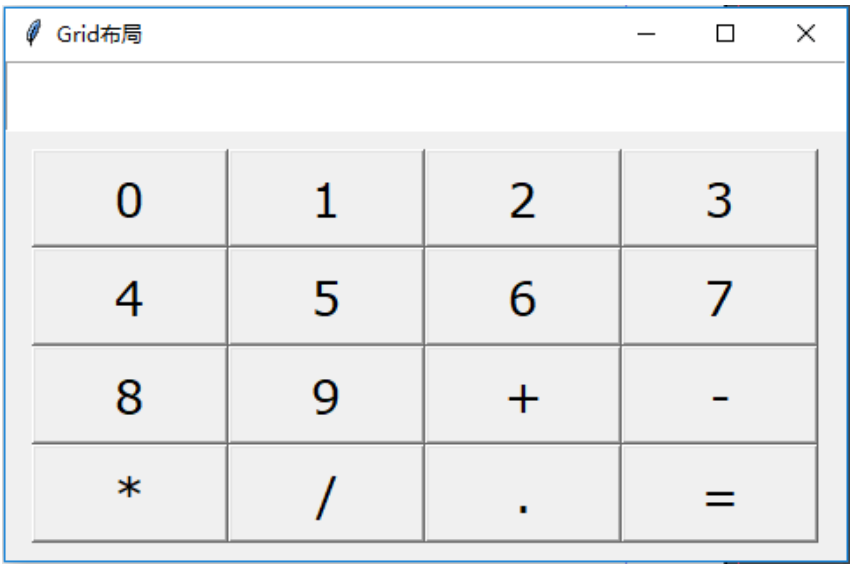
grid() 方法支持的 ipadx、ipady、padx、pady 与 pack() 方法相同，额外增加了如下选项：

- column：指定将组件放入哪列。第一列的索引为 0。
- columnspan：指定组件横跨多少列。
- row：指定组件放入哪行。第一行的索引为 0
- rowspan：指定组件横跨多少行。
- sticky：类似于 pack() 的 anchor 选项，支持 N（上北）、E（右东）、S（下南）、W（左西）、NW（左上西北）、NE（右上东北）、SW（左下西南）、SE（右下东南）、CENTER（默认值中）。

Grid 布局来实现一个计算器界面：

```
from tkinter import *
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        e = Entry(relief=SUNKEN, font=('Courier New', 24), width=25)          # 创建一个输入组件
        e.pack(side=TOP)                                                      # 对该输入组件使用 Pack 布局，放在容器顶部
        p = Frame(self.master)
        p.pack(side=TOP, padx=10, pady=10)
        # 定义字符串的元组，为按钮的字符
        names = ("0" , "1" , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9" , "+" , "-" , "*" , "/" , "." , "=")
        for i in range(len(names)):                # 遍历字符串元组
            b = Button(p, text=names[i], font=('Verdana', 20), width=6)          # 创建 Button，放入 p 组件
            b.grid(row=i // 4, column=i % 4)
root = Tk()
root.title("Grid 布局 ")
App(root)
root.mainloop()
```

界面效果：



Place 布局管理器

Place 布局是绝对布局，要求显式指定每个组件的绝对位置或相对于其他组件的位置。Tkinter 容器内的坐标系统的原点 (0,0) 在**左上角**，其中 X 轴向右延伸，Y 轴向下延伸。

place() 方法支持的选项：

- x：指定组件的 X 坐标，单位 pixel（像素）。x 为 0 代表位于最左边。
- y：指定组件的 Y 坐标，单位 pixel（像素）。y 为 0 代表位于最右边。
- relx：指定组件的 X 坐标（以父容器总宽度为单位 1，取值范围 0.0~1.0）。其中 0.0 代表位于窗口最左边，1.0 代表位于窗口最右边，0.5 代表位于窗口中间。
- rely：指定组件的 Y 坐标（以父容器总宽度为单位 1，取值范围 0.0~1.0）。其中 0.0 代表位于窗口最上边，1.0 代表位于窗口最下边，0.5 代表位于窗口中间。
- width：指定组件的宽度，单位 pixel（像素）。
- height：指定组件的高度，单位 pixel（像素）。
- relwidth：指定组件的宽度（以父容器总宽度为单位 1，取值范围 0.0~1.0）。其中 1.0 代表整个窗口宽度，0.5 代表窗口的一半宽度。
- relheight：指定组件的高度（以父容器总宽度为单位 1，取值范围 0.0~1.0）。其中 1.0 代表整个窗口高度，0.5 代表窗口的一半高度。
- bordermode：该属性支持“inside”或“outside” 属性值，用于指定当设置组件的宽度、高度时是否计算该组件的边框宽度。

```
from tkinter import *
import random
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        books = ('疯狂 Python 讲义', '疯狂 Swift 讲义', '疯狂 Kotlin 讲义', '疯狂 Java 讲义', '疯狂 Ruby 讲义')
        for i in range(len(books)):
            ct = [random.randrange(256) for x in range(3)]                    # 生成 3 个随机数
            bg_color = "#%02x%02x%02x" % tuple(ct)                            # 将 3 个随机数转成颜色格式
            grayness = int(round(0.299*ct[0] + 0.587*ct[1] + 0.114*ct[2]))
            lb = Label(root, text=books[i], fg = 'White' if grayness < 120 else 'Black', bg = bg_color)
            lb.place(x = 20, y = 36 + i*36, width=180, height=30)        # 使用 place() 设置该 Label 的大小和位置
root = Tk()
root.title("Place 布局 ")
root.geometry("250x250+30+30") # 设置窗口的大小和位置：width x height + x_offset + y_offset
App(root)
root.mainloop()
```

界面效果：

command 绑定事件处理方法

简单的事件处理可通过 command 选项来绑定，该选项绑定为一个函数或方法，当用户单击指定按钮时，通过该 command 选项绑定的函数或方法就会被触发。

```
from tkinter import *
import random
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.label = Label(self.master, width=30,font=('Courier', 20),bg='white')
        self.label.pack()
        bn = Button(self.master, text=' 单击我 ', command=self.change)
        bn.pack()
    def change(self):       # 定义事件处理方法
        self.label['text'] = ' 欢迎学习 Python'
        ct = [random.randrange(256) for x in range(3)]          # 生成 3 个随机数
        bg_color = "#%02x%02x%02x" % tuple(ct)                  # 将 3 个随机数转成颜色格式
        grayness = int(round(0.299*ct[0] + 0.587*ct[1] + 0.114*ct[2]))
        self.label['bg'] = bg_color
        self.label['fg'] = 'black' if grayness > 125 else 'white'
root = Tk()
root.title(" 简单事件处理 ")
App(root)
root.mainloop()
```

bind 绑定事件处理方法

为了弥补 command 绑定事件的不足（无法为具体事件（比如鼠标移动、按键事件）绑定事件处理方法；无法获取事件相关信息），所有 Widget 组件都提供了 bind() 方法，可以为“任意”事件绑定事件处理方法。

Tkinter 支持的各种鼠标、键盘事件如表 所示：

事件	简介
<Button-detail>	鼠标按键的单击事件，detail 指定单击的鼠标键。单击鼠标左键为 <Button-1>，单击鼠标中键为 <Button-2>，单击鼠标右键为 <Button-3>，单击向上滚动的滚轮为 <Button-4>，单击向下滚动的滚轮为 <Button-5>
<modifier-Motion>	鼠标在组件上的移动事件，modifier 指定按住的鼠标键。按住鼠标左键移动为 <B1-Motion>，锁住鼠杯中键移动为 <B2-Motion>，按住鼠标右键移动为 <B3-Motion>
<ButtonRelease-detail>	鼠标按键的释放事件，detail 指定被释放的鼠标键。鼠标左键被释放为 <ButtonRelease-1>，鼠标中键被释放为 <ButtonRelease-2>，鼠标右键被释放为 <ButtonRelease-3>
<Double-Button-detail> 或 <Double-detail>	用户双击某个鼠标键的事件，detail 指定双击的鼠标键。双击鼠标左键为 <Double-1>，双击鼠标中键为 <Double-2>，双击鼠标右键为 <Double-3>，双击向上滚动的滚轮为 <Double-4>，双击向下滚动的滚轮为 <Double-5>
<Enter>	鼠标进入组件的事件。注意，<Enter> 事件不是~~按下回车键~~事件
<Leave>	鼠标移出组件事件
<FocusIn>	组件及其包含的子组件获得焦点
<FocusOut>	组件及其包含的子组件失去焦点

<Return>	按下回车键的事件。可以为所有按键绑定事件处理方法。特殊键位名称包括 Cancel、BackSpace、Tab、Return（回车）、Shift_L（左 Shift，只写 Shift 代表任意 Shift）、Control_L（左 Ctrl，只写 Control 代表任意 Ctrl）、Alt_L（左 Alt，只写 Alt 代表任意 Alt）、Pause、Caps_Lock、Escape、Prior（Page Up）、Next（Page Down）、End、Home、Left、Up、Right、Down、Print、Insert、Delete、F1、F2、F3、F4、F5、F6、F7、F8、F9、F10、F11、F12、Num_Lock 和 Scroll_Lock
<Key>	键盘上任意键的单击事件，程序可通过 event 获取用户单击了哪个键
a	键盘上指定键被单击的事件。‘a’单击 a 键，‘b’单击 b 键（不要尖括号）……
<Shift-Up>	在 Shift 键被按下时按 Up 键。类似的还有 <Shift-Left>、<Shift-Down>、<Alt-Up>、<Control-Up> 等
<Configure>	组件大小、位置改变的事件。组件改变之后的大小、位置可通过 event 的 width、height、x、y 获取

```
from tkinter import *
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 信息显示 Label
        self.show = Label(self.master, width=40, height=1,bg='white', font=('Courier New', 20))
        self.show.pack()
        # 鼠标移动区域 Label
        lb = Label(self.master, width=40, height=3,bg='lightgreen', font=('Times', 20))
        lb.bind('<Motion>', self.motion)                # 为鼠标移动事件绑定事件处理方法
        lb.bind('<B1-Motion>', self.press_motion)   # 为按住左键时鼠标移动事件绑定事件处理方法
        lb.pack()
        # 按钮 Button
        bn = Button(self.master, text=' 单击我或双击我 ')
        bn.pack(fill=BOTH, expand=YES)
        bn.bind('<Button-1>', self.one)                 # 为左键单击事件绑定处理方法
        bn.bind('<Double-1>', self.double)              # 为左键双击事件绑定处理方法
    def motion(self, event):
        self.show['text'] = " 鼠标移动到 : (%s %s)" % (event.x, event.y)
        return
    def press_motion(self, event):
        self.show['text'] = " 按住鼠标的位置为 : (%s %s)" % (event.x, event.y)
        return
    def one(self, event):
        self.show['text'] = " 左键单击 :%s" % event.widget['text']
    def double(self, event):
        print(" 左键双击击 , 退出程序 :", event.widget['text'])
        import sys; sys.exit()
root = Tk()
root.title(' 鼠标事件 ')
App(root)
root.mainloop()
```

ttk 组件

为了 GUI 组件不美观的不足，Tkinter 后来引入了一个 ttk 组件作为补充（简单包装、美化），并使用功能更强大的 Combobox 取代了原来的 Listbox，新增了 LabeledScale（带标签的 Scale）、Notebook（多文档窗口）、Progressbar（进度条）、Treeview（树）等组件。

Variable 类

Tkinter 支持将很多 GUI 组件与变量进行双向绑定（程序改变变量值时，组件的显示内容或值会随之改变；组件的内容发生改变时，如用户输入，变量的值也会随之改变）。要使用双向绑定，只要为组件指定 variable（通常绑定组件的 value）、textvariable（通常绑定组件显示的文本）等属性即可。但双向绑定有一个限制，不允许将组件和普通变量进行绑定，只能 Variable 类的子类进行绑定。Variable 类的子类包括 ShringVar()（用于包装 str 值的变量）、IntVar()（用于包装整型值的变量）、DoubleVar()（用于包装浮点值的变量）和 BooleanVar()（用于包装 bool 值的变量）。

```
from tkinter import *      # 导入 tkinter
from tkinter import ttk    # 导入 ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.st = StringVar()
        ttk.Entry(self.master, width=24, font=('StSong', 20, 'bold'), foreground='red',
            textvariable=self.st       # 创建 Entry 组件，将其 textvariable 绑定到 self.st 变量
            ).pack(fill=BOTH, expand=YES)
        f = Frame(self.master)       # 创建 Frame 作为容器
        f.pack()
        ttk.Button(f, text=' 改变 ', command=self.change).pack(side=LEFT)  # 创建 " 改变 " 按钮
        ttk.Button(f, text=' 获取 ', command=self.get).pack(side=LEFT)       # 创建 " 获取 " 按钮
    def change(self):
        books = (' 疯狂 Python 讲义 ', ' 疯狂 Kotlin 讲义 ', ' 疯狂 Swift 讲义 ')
        import random
        self.st.set(books[random.randint(0, 2)])      # 改变 self.st 变量值，Entry 的内容随之改变
    def get(self):
        from tkinter import messagebox
        # 获取 self.st 变量的值，实际上就是获取与之绑定的 Entry 中的内容
        messagebox.showinfo(title=' 输入内容 ', message=self.st.get() )      # 显示 self.st 变量的值
root = Tk()
root.title("variable 测试 ")
App(root)
root.mainloop()
```

compound 选项

为按钮或 Label 等组件同时指定 text（文本）和 image（图片）两个选项时，通常 image 会覆盖 text。如希望组件能同时显示文本和图片，可通过 compound 选项进行控制。

compound 选项支持的属性值有：None（图片覆盖文字）、left（字符串，图片在左，文本在右）、right（字符串，图片在右，文本在左）、top（字符串，图片在上，文本在下）、bottom（字符串，图片在底，文本在上）、center（字符串，文本在图片上方）。

```
from tkinter import *       # 导入 tkinter
from tkinter import ttk     # 导入 ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        bm = PhotoImage(file = 'logo.png')              # 创建一个位图
        self.label = ttk.Label(self.master,  font=('StSong', 20, 'bold'), foreground='red',
            text=' 学编程 \n 的神器 ', image=bm )      # 创建一个 Label，同时指定 text 和 image
        self.label.bm = bm
        self.label['compound'] = None                   # 设置 Label 默认的 compound 为 None
        self.label.pack()
        f = ttk.Frame(self.master)                      # 创建 Frame 容器，用于装多个 Radiobutton
        f.pack(fill=BOTH, expand=YES)
        compounds = ('None', "LEFT", "RIGHT", "TOP", "BOTTOM", "CENTER")
        self.var = StringVar()          # 定义一个 StringVar 变量，用作绑定 Radiobutton 的变量
        self.var.set('None')
        for val in compounds:           # 使用循环创建多个 Radionbutton 组件
            rb = Radiobutton(f, text = val, padx = 20, variable = self.var, value = val,
                command = self.change_compound).pack(side=LEFT, anchor=CENTER)
    def change_compound(self):  # 实现 change_compound 方法，动态改变 Label 的 compound 选项
        self.label['compound'] = self.var.get().lower()
root = Tk()
root.title("compound 测试 ")
App(root)
root.mainloop()
```

Entry（单行）和 Text（多行）输入框控件

Entry 和 Text 组件都是可接收用户输入的输入框组件，区别是 Entry 是单行输入框组件，Text 是多行输入框组件，而且 Text 可以为不同的部分添加不同的格式，甚至响应事件。Entry 和 Text 都提供：

- get() 方法：获取文本框中的内容
- insert() 方法：改变文本框中的内容
- delete(self,first,last=None) 方法：删除从 first 到 last 之间的内容

关于 Entry 和 Text 支持的索引说明：由于 Entry 是单行文本框组件，因此它的索引很简单（如 (3,8) 表示第 4 个字符到第 8 个字符）；Text 是多行文本框组件，因此它的索引需要同时指定行号和列号（如 (2.2,3.6) 表示第 2 行第 3 个字符到第 3 行第 7 个字符）。

```
from tkinter import *
from tkinter import ttk
from tkinter import messagebox
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.entry = ttk.Entry(self.master, width=44, font=('StSong', 14), foreground='green')  # 创建 Entry 组件
        self.entry.pack(fill=BOTH, expand=YES)
       self.text = Text(self.master, width=44, height=4, font=('StSong', 14), foreground='gray')  # 创建 Text 组件
        self.text.pack(fill=BOTH, expand=YES)
        f = Frame(self.master).pack()
        ttk.Button(f, text=' 开始插入 ', command=self.insert_start).pack(side=LEFT)
        ttk.Button(f, text=' 编辑处插入 ', command=self.insert_edit).pack(side=LEFT)
        ttk.Button(f, text=' 结尾插入 ', command=self.insert_end).pack(side=LEFT)
        ttk.Button(f, text=' 获取 Entry', command=self.get_entry).pack(side=LEFT)
        ttk.Button(f, text=' 获取 Text', command=self.get_text).pack(side=LEFT)
    def insert_start(self):              # 在 Entry 和 Text 开始处插入内容
        self.entry.insert(0, 'Kotlin')
        self.text.insert(1.0, 'Kotlin')
    def insert_edit(self):               # 在 Entry 和 Text 的编辑处插入内容
        self.entry.insert(INSERT, 'Python')
        self.text.insert(INSERT, 'Python')
    def insert_end(self):                # 在 Entry 和 Text 的结尾处插入内容
        self.entry.insert(END, 'Swift')
        self.text.insert(END, 'Swift')
    def get_entry(self):
        messagebox.showinfo(title=' 输入内容 ', message=self.entry.get())
    def get_text(self):
        messagebox.showinfo(title=' 输入内容 ', message=self.text.get(1.0, END))
root = Tk()
root.title("Entry 测试 ")
App(root)
root.mainloop()
```

Text 实际上是一个功能强大的“富文本”编辑组件，不仅可以插入文本内容，还可以：

- **插入图片：**image_create(self, index, cnf={}, **kw)
- **设置插入文本的格式：**insert(self, index, chars, *args)
- **使用滚动条:** 将 Scrollbar 的 command 设为目标组件的 xview(水平滚动条) 或 yview(垂直滚动条)；将目标组件的 xscrollcommand 或 yscrollcommand 属性设为 Scrollbar 的 set 方法

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        text1 = Text(self.master, height=27, width=32)                  # 创建 text1
        text1.pack(side=LEFT, fill=BOTH, expand=YES)
        book = PhotoImage(file='logo.png');text1.bm = book                      # 创建图片 book
        text1.insert(END,'\n');text1.image_create(END, image=book)          # 结尾处插入图片 book
        text2 = Text(self.master, height=33, width=50)                  # 创建 text1
        text2.pack(side=LEFT,fill=BOTH, expand=YES)
        self.text = text2
        scroll = Scrollbar(self.master, command=text2.yview)            # 创建 text2 的纵向滚动条
        scroll.pack(side=RIGHT, fill=Y)
        text2.configure(yscrollcommand=scroll.set)                      # 关联 text2 纵向滚动条
        text2.tag_configure('title', font=(' 楷体 ', 20, 'bold'),foreground='red', justify=CENTER,
            spacing3=20)                        # 创建 title 的样式，spacing3 为段间距
        text2.tag_configure('detail', foreground='darkgray',font=(' 微软雅黑 ', 11, 'bold'),
            spacing2=10,spacing3=15)        # 创建 detail 的样式，spacing2 为行间距，spacing3 为段间距
        text2.insert(END,'\n');text2.insert(END,'C 语言中文网 \n', 'title')         # 插入文本（title 样式）
        star = PhotoImage(file='logo.png');text2.bm = star                      # 创建图片 star
        details = ('C 语言中文网成立于 2012 年初，目前已经运营了将近 5 年，我们致力于分享精品教程，
帮助对编程感兴趣的读者。\n' ,' 我们一直都在坚持的是：认认真真、一丝不苟、以工匠的精神来打磨每
一套教程，让读者感受到作者的用心，以及默默投入的时间，由衷地心动和点赞。\n',' 这样的教程是一
件作品，而不是呆板的文字！ \n')
        for de in details:     # 采用循环插入多条介绍信息
            text2.image_create(END, image=star);text2.insert(END, de, 'detail')
        url =['http://vip.biancheng.net/','http://c.biancheng.net/'];
        name =['VIP 会员 ', 'C 语言中文网 ']
        m=0
        for each in name:  # 为每个链接创建单独的配置
            text2.tag_configure(m, foreground='blue', underline=True, font=(' 微软雅黑 ', 13, 'bold'))
            text2.tag_bind(m, '<Enter>', self.show_arrow_cursor)
            text2.tag_bind(m, '<Leave>', self.show_common_cursor)
            # 使用 handlerAdaptor 包装 , 将当前链接参数传入事件处理函数
            text2.tag_bind(m, '<Button-1>', self.handlerAdaptor(self.click, x = url[m]))
            text2.insert(END, each + '\n', m)
            m += 1
    def show_arrow_cursor(self, event):self.text.config(cursor='arrow')       # 光标移上去时变成箭头
    def show_common_cursor(self, event):self.text.config(cursor='xterm')  # 光标移出去时恢复原样
    def click(self, event, x):import webbrowser;webbrowser.open(x)          # 使用默认浏览器打开链接
    def handlerAdaptor(self, fun,**kwds):return lambda event, fun=fun, kwds=kwds: fun(event,**kwds)
root = Tk()
root.title("Text 测试 ")
App(root)
root.mainloop()
```

Radiobutton（单选按钮）控件

Radiobutton（单选按钮）组件可以绑定一个方法或函数，当单选钮被选择时，该方法或函数会被触发。为了将多个 Radiobutton 编为一组，需要将多个 Radiobutton 绑定到同一个变量，当其中一个单选钮被选中时，该变量会随之改变；反过来，当该变量发生改变时，也会自动选中该变量值所对应的单选钮。

Radiobutton 既可以显示文本，也可以显示图片（指定 image 选项）。如果希望同时显示文字和图片，可通过 compound 选项进行控制（如果不指定 compound 选项，默认为 None，只显示图片）。

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        ttk.Label(self.master, text=' 选择您喜欢的教程 :').pack(fill=BOTH, expand=YES) # 创建 Label 组件
        self.intVar = IntVar()
        books = ('C 语言入门 ', 'Python 入门 ','C++ 入门 ',  'Java 入门 ')        # 定义名称元组
        images = ('C.png', 'P.png','C2.png',  'J.png')                            # 定义图片元组
        i = 1
        for book in books:              # 采用循环创建多个 Radiobutton
            bm = PhotoImage(file = 'images/' + images[i-1])
            b=ttk.Radiobutton(self.master,text = book,value=i,image=bm,
                compound = LEFT,              # 图片在文字左边
                variable = self.intVar,       # 将 Radiobutton 绑定到 self.intVar 变量
                command = self.change)        # 将选中事件绑定到 self.change 方法
            b.bm = bm
            b.pack(anchor=W)
            i += 1
        self.intVar.set(2)    # 设置 Radiobutton 绑定的变量为 2，则选中 value 为 2 的 Radiobutton
    def change(self):
        from tkinter import messagebox
        # 通过 Radiobutton 绑定变量获取选中的单选框
        messagebox.showinfo(title=None, message=self.intVar.get() )root = Tk()
root.title("Radiobutton 测试 ")
root.iconbitmap('images/fklogo.ico')        # 自定义窗口图标
App(root)
root.mainloop()
```

Checkbutton（复选按钮）控件

Checkbutton（复选按钮）与 Radiobutton 很相似（为选中事件绑定处理方法或函数，显示文字和图片，绑定变量），不同的是 Checkbutton 允许选择多项（每组 Radiobutton 只能选择一项），因此需要为每个 Checkbutton 都绑定一个变量。

Checkbutton 像一个开关，支持两个值：开关打开的值和开关关闭的值。在创建 Checkbutton 时可同时设置 onvalue（打开值，默认 1）和 offvalue（关闭值，默认 0）选项。

```
from tkinter import *
from tkinter import ttk
from tkinter import messagebox
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        ttk.Label(self.master, text=' 选择您喜欢的人物 :').pack(fill=BOTH, expand=YES)
        self.chars = []
        characters = (' 孙悟空 ', ' 猪八戒 ',' 唐僧 ', ' 牛魔王 ')              # 定义元组
        for ch in characters:           # 采用循环创建多个 Checkbutton
            intVar = IntVar(); self.chars.append(intVar)
            cb = ttk.Checkbutton(self.master,text = ch,
                variable = intVar,                  # 将 Checkbutton 绑定到 intVar 变量
                command = self.change)          # 将选中事件绑定到 self.change 方法
            cb.pack(anchor=W)
        ttk.Label(self.master, text=' 选择您喜欢的教程 :').pack(fill=BOTH, expand=YES)
        self.books = []
        books = ('C 语言入门 ', 'Python 入门 ','C++ 入门 ',  'Java 入门 ')        # 定义名称元组
        vals = ('C', 'Python', 'C++','Java')                                    # 定义值元组
        images = ('C.png', 'P.png','C2.png',  'J.png')                          # 定义图片元组
        i = 0
        for book in books:              # 采用循环创建多个 Checkbutton
            strVar = StringVar(); self.books.append(strVar)
            bm = PhotoImage(file = 'images/' + images[i])
            cb = ttk.Checkbutton(self.master,text = book,image=bm, compound = LEFT,
                variable = strVar,          # 将 Checkbutton 绑定到 strVar 变量
                onvalue = vals[i],          # onvalue 绑定到值
                offvalue = ' 无 ',
                command = self.books_change)          # 将选中事件绑定到 books_change 方法
            cb.bm=bm
            cb.pack(anchor=W)
            i += 1
    def change(self):
        new_li = [str(e.get()) for e in self.chars]       # 将 self.chars 列表转换成元素为 str 的列表
        st = ', '.join(new_li)                            # 将 new_li 列表连接成字符串
        messagebox.showinfo(title=None, message=st)
    def books_change(self):
        new_li = [e.get() for e in self.books]            # 将 self.books 列表转换成元素为 str 的列表
        st = ', '.join(new_li)                            # 将 new_li 列表连接成字符串
        messagebox.showinfo(title=None, message=st)
root = Tk()
root.title("Checkbutton 测试 ")
root.iconbitmap('images/python.ico')# 改变窗口图标
App(root)
root.mainloop()
```

Listbox（列表框）控件

Listbox（列表框）用户可通过列表框来选择一个列表项。Listbox 除支持大部分通用选项之外，还提供：

- selectmode 选项: 设置选择模式(字符串)。支持: browse(单选模式，支持按住鼠标键拖动来改变选择)、multiple（多边模式）、single（单边模式，必须通过鼠标键单击来改变选择）、extended（扩展的多边模式，必须通过 CtrL 或 Shift 键辅助实现多选）
- listvariable 选项：绑定变量，但不能控制 Listbox 选中哪些项，而是控制 Listbox 包含哪些项。
- insert(self, index, *elements) 方法：添加一个或多个选项。index 指定选项的插入位置，支持 END（结尾处）、ANCHOR（当前位置）和 ACTIVE（选中处）等特殊索引
- selection_set(self, first, last=None) 方法：选中 first 项或到 last（包含）的所有列表项（指定了 last）。
- selection_clear(self, first, last=None) 方法：取消选中 first 项或到 last（包含）的所有列表项（指定了 last）。
- delete(self, first, last=None) 方法：删除 first 项或到 last（包含）的所有列表项（指定了 last）。
- curselection() 方法：返回一个元组，该元组包含当前 Listbox 的所有选中项。
- bind() 方法：绑定事件处理函数或方法（不支持 command 选项绑定）。

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        f_model = Frame(self.master); f_model.pack()
        Label(f_model, text = ' 选择模式 :').pack(side=LEFT)
        self.modelVar = StringVar()          # 用于绑定 Radiobutton（模式选择）
        for m in ('multiple', 'browse', 'single', 'extended'):
            rb = ttk.Radiobutton(f_model, text=m, value=m,variable =self.modelVar,command=self.choose_mode)
            rb.pack(side=LEFT)
        self.modelVar.set('browse')
        f_listbox = Frame(self.master); f_listbox.pack(fill=Y, expand=YES)
        self.listboxVar = StringVar();          # 用于绑定 Listbox
        self.lb = Listbox(f_listbox, listvariable = self.listboxVar)        # 创建 Listbox 组件
        self.lb.pack(side=LEFT, fill=Y, expand=YES)
        for item in range(20):self.lb.insert(END, str(item))                # 可循环插入多个元素
        self.lb.insert(ANCHOR, 'Python', 'Kotlin', 'Swift', 'Ruby')         # 也直接插入多个元素
        scroll = Scrollbar(f_listbox, command=self.lb.yview)                # 设置 Listbox 的纵向滚动
        scroll.pack(side=RIGHT, fill=Y)
        self.lb.configure(yscrollcommand=scroll.set)                        # 关联 Listbox 的纵向滚动
        self.lb.bind("<Double-1>", self.click)                              # 为双击事件绑定事件处理方法
        f_button = Frame(self.master); f_button.pack()
        Button(f_button, text=" 选中前 10 项 ", command=self.select).pack(side=LEFT)
        Button(f_button, text=" 清除选中 1-3 项 ", command=self.clear_select).pack(side=LEFT)
        Button(f_button, text=" 删除 5-8 项 ", command=self.delete).pack(side=LEFT)
        Button(f_button, text=" 绑定变量设置：12,15", command=self.var_select).pack(side=LEFT)
    def choose_mode(self): self.lb['selectmode'] = self.modelVar.get()      # 修改选择模式
    def select(self):self.lb.selection_set(0, 9)                            # 选中指定项
    def clear_select(self):self.lb.selection_clear(1,3)                     # 取消选中指定项
    def delete(self):self.lb.delete(5, 8)                                   # 删除指定项
    def var_select(self):self.listboxVar.set(('12', '15'))                  # 修改绑定的变量
    def click(self, event):                                                 # 获取当前选中项
        from tkinter import messagebox
        messagebox.showinfo(title=None, message=str(self.lb.curselection()))
root = Tk()
root.title("Listbox 测试 ")
App(root)
root.mainloop()
```

Combobox（复合框）控件

Combobox（复合框）是 ttk 模块下 Listbox 的改进版，既提供了单行文本框让用户直接输入（就像 Entry 一样），也提供了下拉列表框供用户选择（就像 Listbox 一样）。Combobox 的用法更加简单：

- values 选项：设置多个选项。
- state 选项：设置为‘readonly’表示文本框不允许编辑，只能通过下拉列表框的列表项来改变。
- textvariable 选项：绑定变量，可通过该变量来获取或修改 Combobox 组件的值。
- postcommand 选项：指定事件处理函数或方法，当用户单击 Combobox 的下拉箭头时，程序就会触发 postcomrnand 选项指定的事件处理函数或方法。

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.strVar = StringVar()
        # 创建 Combobox 组件
        self.cb = ttk.Combobox(self.master,
            value=['Python', 'Ruby', 'Kotlin', 'Swift'],    # 设置选项值
            textvariable=self.strVar,                       # 绑定到 self.strVar 变量
            postcommand=self.choose)                        # 单击下拉箭头时触发 choose 方法
        self.cb.pack(side=TOP)
        # self.cb['values'] = ['Python', 'Ruby', 'Kotlin', 'Swift']          # 也可以后设置选项值
        f = Frame(self.master)
        f.pack()
        self.isreadonly = IntVar()
        # 创建 Checkbutton
        Checkbutton(f, text = ' 是否只读 :',
            variable=self.isreadonly,           # 绑定到 self.isreadonly 变量
            command=self.change                 # 选择是时触发 change 方法
            ).pack(side=LEFT)
        # 创建 Button，单击该按钮激发 setvalue 方法
        Button(f, text = ' 绑定变量设置 ',command=self.setvalue).pack(side=LEFT)
    def choose(self):       # 获取 Combbox 的当前值
        from tkinter import messagebox
        messagebox.showinfo(title=None, message=str(self.cb.get()))
    def change(self):       # 设置 Combbox 只读方式
        self.cb['state'] = 'readonly' if self.isreadonly.get() else 'enable'
    def setvalue(self):     # 设置 Combbox 值
        self.strVar.set(' 我爱 Python')
root = Tk()
root.title("Combobox 测试 ")
App(root)
root.mainloop()
```

Spinbox 控件

Spinbox 控件是一个带有两个小箭头的文本框，用户既可以通过两个小箭头上下调整该组件内的值，也可以直接在文本框内输入内容作为该组件的值。

- from 选项：选项最小值。因 from 是关键字，实际使用时写成 from_。
- to 选项：选项最大值。
- increment 选项：选项步长值。
- values 选项：指定选项列表，选项的值可以是 list 或 tuple 。
- textvariable 选项：绑定变量，通过该变量来获取或修改 Spinbox 的值。
- command 选项：指定事件处理函数或方法（单击向上、向下箭头时触发）。

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 通过指定 from_、to、increament 选项创建 Spinbox
        ttk.Label(self.master, text=' 通过 from、to、increment 设置 Spinbox 值 ').pack()
        sb1 = Spinbox(self.master, from_=20,to=100,increment=5)
        sb1.pack(fill=X, expand=YES)
        # 通过指定 values 选项创建 Spinbox
        ttk.Label(self.master, text=' 通过 values 设置 Spinbox 值 ').pack()
        spinboxValue=('Python', 'Swift', 'Kotlin', 'Ruby')
        self.sb2 = Spinbox(self.master,values=spinboxValue)
        self.sb2.pack(fill=X, expand=YES)
        # 绑定变量和事件
        ttk.Label(self.master,text=' 绑定变量和事件 ').pack()
        self.intVar = IntVar()
        self.sb3 = Spinbox(self.master,values=list(range(20, 100, 4)),
            textvariable = self.intVar,          # 绑定 self.intVar 变量
            command = self.press)                # 绑定 self.press 方法
        self.sb3.pack(fill=X, expand=YES)
        self.intVar.set(33)                      # 通过 self.intVar 变量改变 Spinbox 的值
    def press(self):
        print(' 通过 get() 方法获得值：',self.sb3.get())        # 通过 get() 方法获得 Spinbox 的值
        print(' 通过绑定变量获得值：',self.intVar.get())        # 通过绑定变量获得 Spinbox 的值
root = Tk()
root.title("Spinbox 测试 ")
App(root)
root.mainloop()
```

Scale 控件和 LabeledScale 控件

Scale 组件代表一个滑动条，可以为该滑动条设置最小值和最大值，也可以设置滑动条每次调节的步长。

- from、to、resolution：分别设置最小值、最大值和滑动步长。
- label：设置标签内容。
- length、width、troughcolor：分别设置轨道的长度、宽度和背景色。
- sliderlength、sliderrelief：分别设置滑块长度和立体样式。
- showvalue：设置是否显示当前值。
- orient：设置方向。支持 VERTICAL（垂直）和 HORIZONTAL（水平）。
- digits：设置有效数字至少要有几位。
- variable：用于绑定变量。
- command：用于绑定事件处理，函数或方法。

可通过三种方式获得 Scale 的值：

- 通过事件处理方法的参数来获取。
- 通过 Scale 组件提供的 get() 方法来获取。
- 通过 Scale 组件绑定的变量来获取。

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.doubleVar = DoubleVar()          # 定义 Scale 的绑定变量
        self.scale = Scale(self.master,label = ' 示范 Sacle',orient = HORIZONTAL, from_=-100,to=100,re
solution=5,showvalue=YES, length=400,width = 30,troughcolor='lightblue', sliderlength=20,sliderrelief
=SUNKEN, command = self.change,variable = self.doubleVar)    # 创建一个 Frame 作为容器
        self.scale.pack()
        f = Frame(self.master);f.pack(fill=X, expand=YES, padx=10)
        Label(f, text=' 是否显示值 :').pack(side=LEFT)
        self.showVar = IntVar(); self.showVar.set(1)          # 定义 Radiobutton 的绑定变量并设置为 1
        i = 0
        for s in (' 不显示 ', ' 显示 '):  # 创建两个 Radiobutton 控制 Scale 是否显示值
            Radiobutton(f, text=s, value=i,variable=self.showVar, command=self.switch_show).pack(side=LEFT)
            i += 1
        f = Frame(self.master);f.pack(fill=X, expand=YES, padx=10)
        Label(f, text=' 方向 :').pack(side=LEFT)
        self.orientVar = IntVar(); self.orientVar.set(0)          # 定义 Radiobutton 的绑定变量并设置为 0
        i = 0
        for s in (' 水平 ', ' 垂直 '):     # 创建两个 Radiobutton 控制 Scale 的方向
            Radiobutton(f, text=s, value=i,variable=self.orientVar, command=self.switch_orient).pack(side=LEFT)
            i += 1
    def switch_show(self):    self.scale['showvalue'] = self.showVar.get()
    def switch_orient(self):    self.scale['orient'] = VERTICAL if self.orientVar.get() else HORIZONTAL
    def change(self, value):    print(value, self.scale.get(), self.doubleVar.get())
root = Tk()
root.title("Scale 测试 ")
App(root)
root.mainloop()
```

ttk.LabeledScale 是平台化的滑动条，允许设置的选项很少，只能设置 from、to 和 compound 等有限的几个选项，而且总是生成一个水平滑动条（不能变成垂直的），compound 选项控制滑动条的数值标签是显示在滑动条的上方，还是滑动条的下方。

LabelFrame 容器

Labelframe 是 Frame 容器的改进版，允许为容器添加一个标签，该标签既可以是普通的文字标签，也可以将任意 GUI 组件作为标签。Labelframe 通过如下选项对标签进行定制：

- labelwidget：设置可以将任意 GUI 组件作为标签。
- labelanchor：设置标签的位置。支持 'e'、's'、'w'、'n'、'es'、'ws'、'en'、'wn'、'ne'、'nw'、'se'、'sw' 。

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建外层 Labelframe 容器
        self.outf = ttk.Labelframe(self.master, text='Labelframe 示例 ',padding=20)
        self.outf.pack(fill=BOTH, expand=YES, padx=10, pady=10)
        # 创建内层 Labelframe 容器
        self.inf = ttk.Labelframe(self.outf, padding=20)
        self.inf.pack(fill=BOTH, expand=YES, padx=10, pady=10)
        # 创建内层 Labelframe 容器 labelwidget 用图片
        bm = PhotoImage(file='images/p.png')
        lb = Label(self.inf, image=bm)
        lb.bm = bm
        self.inf['labelwidget'] = lb     # 将 Labelframe 的标题设为显示图片的 Label
        # 定义代表 Labelframe 的标题位置的 12 个常量
        self.books = ['e', 's', 'w', 'n', 'es', 'ws', 'en', 'wn', 'ne', 'nw', 'se', 'sw']
        self.intVar = IntVar()          # 定义绑定变量，用于绑定到 Radiobutton
        i = 0
        for book in self.books:# 使用循环创建多个 Radiobutton，并放入 Labelframe 中
            Radiobutton(self.inf, text= book,value=i,command=self.change,variable=self.intVar)
                    .pack(side=LEFT)
            i += 1
        self.intVar.set(9)
    def change(self):
        # 通过 labelanchor 选项改变 Labelframe 的标题的位置
        self.inf['labelanchor'] = self.books[self.intVar.get()]
root = Tk()
root.title("Labelframe 测试 ")
App(root)
root.mainloop()
```

Panedwindow 容器

Panedwindow 是一个非常有特色的容器，自带布局管理功能（不需要使用 Pack、Grid 或 Place 布局），通过 orient 选项指定水平或垂直方向，让容器中的各组件按水平或垂直方向排列。

通过在 Panedwindow 容器中添加多个子组件，并为每个子组件划分一个区域，用户可用鼠标移动各区域的分隔线来改变各子组件的大小（如果没有显式指定大小，子细件总是自动占满整个区域）。

Panedwindow 操作子组件的方法有：

- add(self, child, **kw)：添加一个子组件。
- insert(self, pos, child,**kw)：在 pos 位置插入一个子组件。
- remove(self, child)：删除一个子组件，该子组件所在区域也被删除。

Panedwindow 组件可以嵌套的，以实现功能更丰富的界面。

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        # 创建 Style
        style = ttk.Style()
        style.configure("fkit.TPanedwindow",background='darkgray', relief=RAISED)
        # 创建 Panedwindow 组件（水平排列），通过 style 属性配置分隔线
        pwindow = ttk.Panedwindow(self.master,orient=HORIZONTAL, style="fkit.TPanedwindow")
        pwindow.pack(fill=BOTH, expand=YES)
        # 左边 Label 组件
        left = ttk.Label(pwindow, text=" 左边标签 ", background='pink')
        pwindow.add(left)
        # 右边 Panedwindow 组件（垂直排列）
        rightwindow = PanedWindow(pwindow, orient=VERTICAL)
        pwindow.add(rightwindow)
        # 右边 Panedwindow 组件添加上 Label 组件
        top = Label(rightwindow, text=" 右上标签 ", background='lightgreen')
        rightwindow.add(top)
        # 右边 Panedwindow 组件添加下 Label 组件
        bottom = Label(rightwindow, text=" 右下标签 ", background='lightblue')
        rightwindow.add(bottom)
root = Tk()
root.title("Panedwindow 测试 ")
App(root)
root.mainloop()
```

OptionMenu 控件

OptionMenu 组件用于构建一个带菜单的按钮，该菜单可以在按钮的四个方向上（由 direction 选项控制）展开。使用 OptionMenu 比较简单，直接调用构造函数即可：

__init__ (self, master, variable, value, *values, **kwargs)

- variable：指定按钮菜单的绑定变量。
- value：指定默认的菜单选择项。
- values：菜单项的值。
- kwargs： OptionMenu 配置选项
- direction ：菜单的展开方向。

```
from tkinter import *
from tkinter import ttk
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.sv = StringVar()          # 定义 OptionMenu 控件的绑定变量
        self.om = ttk.OptionMenu(root, self.sv, 'Python',                    # 初始选中值
            'Kotlin', 'Ruby', 'Swift', 'Java', 'Python', 'JavaScript', 'Erlang',      # 菜单项值
            command = self.print_option                                      # 绑定事件处理方法
            )          # 创建一个 OptionMenu 控件
        self.om.pack()
        lf = ttk.Labelframe(self.master, padding=20, text=' 请选择菜单方向 ') # 创建 Labelframe 容器
        lf.pack(fill=BOTH, expand=YES, padx=10, pady=10)
        self.directions = ['below', 'above', 'left', 'right', 'flush']          # OptionMenu 菜单位置的常量
        self.intVar = IntVar()                  # 定义 Radiobutton 控件的绑定变量
        i = 0
        for direct in self.directions:          # 循环创建多个 Radiobutton，并放入 Labelframe 中
            Radiobutton(lf, text= direct, value=i, command=self.change,variable=self.intVar).pack(side=LEFT)
            i += 1
        self.intVar.set(0)    # 设置位置
    def print_option(self, val):      # 通过两种方式来获取 OptionMenu 选中的菜单项的值
        print(self.sv.get(), val)
    def change(self):                 # 通过 direction 选项改变 OptionMenu 上菜单的展开方向
        self.om['direction'] = self.directions[self.intVar.get()]
root = Tk()
root.title("OptionMenu 测试 ")
App(root)
root.mainloop()
```

Tkinter 对话框（SimpleDialog 类和 Dialog 类）

对话框是图形界面编程中很常用的组件，通常用于向用户生成某种提示信息，或者请求用户输入某些简单的信息。对话框有点类似于顶级窗口，对于对话框有两点需要注意：

- 对话框通常依赖其他窗口，在创建对话框时需要指定 master 属性（对话框的属主窗口）。
- 对话框有非模式（non-modal）和模式（modal）两种，模式对话框总是位于它依赖的窗口之上；在模式对话框被关闭之前，它依赖的窗口无法获得焦点。SimpleDialog 和 Dialog 都是模式的。

simpledialog 和 dialog 模块下分别提供了 SimpleDialog 类和 Dialog 类，都可作为普通对话框使用：

创建 simpledialog.SimpleDialog 对话框的选项：

- title：指定该对话框的标题。
- text：指定该对话框的内容。
- button：指定该对话框下方的几个按钮。
- default：指定该对话框中默认第几个按钮得到焦点。
- cancel：指定当用户通过对话框右上角的 X 按钮关闭对话框时，该对话框的返回值。

创建 dialog.Dialog 对话框，使用 master 指定对话框的属主窗口，dict 来指定选项：

- title：指定该对话框的标题。
- text：指定该对话框的内容。
- strings：指定该对话框下方的几个按钮。
- default：指定该对话框中默认第几个按钮得到焦点。
- bitmap：指定该对话框上的图标。（内置的 10 个位图：“error”、“gray75”、“gray50”、“gray25”、“gray12”、“howglass”、“info”、“questhead”、“question”、“warning”）。

```
from tkinter import *
from tkinter import ttk
from tkinter import simpledialog  # 导入 simpledialog
from tkinter import dialog        # 导入 dialog
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.msg = 'C 语言中文网成立于 2012 年初，目前已经运营了将近 5 年，我们致力于分享精品教程，
帮助对编程感兴趣的读者。'
        ttk.Button(self.master, text=' 打开 SimpleDialog', command=self.open_simpledialog)
            .pack(side=LEFT, ipadx=5, ipady=5, padx= 10)          # 打开 simpledialog 对话框按钮
        ttk.Button(self.master, text=' 打开 Dialog', command=self.open_dialog)
            .pack(side=LEFT, ipadx=5, ipady=5, padx = 10)         # 打开 dialog 对话框按钮
    def open_simpledialog(self):  # 使用 simpledialog.SimpleDialog 创建对话框
        d = simpledialog.SimpleDialog(self.master, title='SimpleDialog 测试 ', text=self.msg,
            buttons=[" 是 ", " 否 ", " 取消 "], cancel=3, default=0)
        print(d.go())         # go() 返回用户单击按钮的值，右上角的 X 按钮值为 3
    def open_dialog(self):          # 使用 dialog.Dialog 创建对话框
        d = dialog.Dialog(self.master, {'title': 'Dialog 测试 ', 'text':self.msg,
            'bitmap': 'question','default': 0,'strings': (' 确定 ',' 取消 ',' 退出 ')})
        print(d.num)          # d.num 返回用户单击按钮的值
root = Tk()
root.title(" 对话框测试 ")
App(root)
root.mainloop()
```

Tkinter 自定义对话框（Toplevel 类）

自定义的对话框（包括定制模式和非模式行为）通过继承 Toplevel 来实现。通过这种方式来实现自定义对话框，有两点要注意：

- 继承 Toplevel 来实现自定义对话框同样需要为对话框指定 master。
- 程序可调用 Toplevel 的 grab_set() 方法让该对话框变成模式对话框，否则就是非模式对话框。

```
from tkinter import *
from tkinter import ttk
from tkinter import messagebox
class MyDialog(Toplevel):          # 自定义对话框类，继承 Toplevel
    def __init__(self, parent, title = None, modal=True):     # 定义构造方法
        Toplevel.__init__(self, parent); self.transient(parent)
        if title: self.title(title)          # 设置标题
        self.parent = parent; self.result = None
        frame = Frame(self)                                  # 创建对话框的主体内容
        self.initial_focus = self.init_widgets(frame)   # 调用 init_widgets 方法来初始化对话框界面
        frame.pack(padx=5, pady=5)
        self.init_buttons()                                  # 调用 init_buttons 方法初始化对话框下方的按钮
        if modal: self.grab_set()                            # 根据 modal 选项设置是否为模式对话框
        if not self.initial_focus: self.initial_focus = self
        self.protocol("WM_DELETE_WINDOW", self.cancel_click)  # 为协议定义处理方法
        self.geometry("+%d+%d" % (parent.winfo_rootx()+50, parent.winfo_rooty()+50)) # 设置对话框位置
        self.initial_focus.focus_set(); self.wait_window(self) # 让对话框获取焦点
    def init_widgets(self, master): # 通过该方法来创建自定义对话框的内容
        Label(master, text=' 用户名 ', font=12,width=10).grid(row=1, column=0)        # 创建“用户名”label
        self.name_entry = Entry(master, font=16); self.name_entry.grid(row=1, column=1)    # 创建 Entry
        Label(master, text=' 密  码 ', font=12,width=10).grid(row=2, column=0)        # 创建“密码”label
        self.pass_entry = Entry(master, font=16); self.pass_entry.grid(row=2, column=1)      # 创建 Entry
    def init_buttons(self): # 通过该方法来创建对话框下方的按钮框
        f = Frame(self)
        w = Button(f, text=" 确定 ", width=10, command=self.ok_click, default=ACTIVE)
        w.pack(side=LEFT, padx=5, pady=5)          # 创建 " 确定 " 按钮 , 绑定 self.ok_click 处理方法
        w = Button(f, text=" 取消 ", width=10, command=self.cancel_click)
        w.pack(side=LEFT, padx=5, pady=5)          # 创建 " 取消 " 按钮 , 绑定 self.cancel_click 处理方法
        self.bind("<Return>", self.ok_click); self.bind("<Escape>", self.cancel_click)
        f.pack()
    def validate(self):return True  # 可重写该方法 , 用于对用户输入的数据进行校验
    def process_input(self):         # 该方法可处理用户输入的数据
        user_name = self.name_entry.get(); user_pass = self.pass_entry.get()
        messagebox.showinfo(message=' 用户输入的用户名 : %s, 密码 : %s' % (user_name , user_pass))
    def ok_click(self, event=None):          # 确定
        if not self.validate():          # 如果不能通过校验，让用户重新输入
            self.initial_focus.focus_set(); return
        self.withdraw(); self.update_idletasks(); self.process_input() # 获取输入数据 , 并显示
        self.parent.focus_set(); self.destroy()          # 将焦点返回给父窗口，销毁自己
    def cancel_click(self, event=None):    # 取消
        self.parent.focus_set(); self.destroy()          # 将焦点返回给父窗口，销毁自己
class App:
    def __init__(self, master):
        self.master = master; self.initWidgets()
    def initWidgets(self):
        ttk.Button(self.master, text=' 模式对话框 ', command=self.open_modal)
            .pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 打开模式对话框按钮
        ttk.Button(self.master, text=' 非模式对话框 ', command=self.open_none_modal)
            .pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 打开非模式对话框按钮
    def open_modal(self): d = MyDialog(self.master, title=' 模式对话框 ') # 默认是模式对话框
    def open_none_modal(self): d = MyDialog(self.master, title=' 非模式对话框 ', modal=False)
root = Tk(); root.title(" 自定义对话框测试 "); App(root); root.mainloop()
```

Tkinter 输入对话框

simpledialog 模块下还有如下便捷的工具函数，可以更方便地生成各种输入对话框：

- askinteger：生成一个让用户输入整数的对话框。
- askfloat：生成一个让用户输入浮点数的对话框。
- askstring：生成一个让用户输入字符串的对话框。

```
from tkinter import *; from tkinter import ttk; from tkinter import simpledialog
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        ttk.Button(self.master, text=' 输入整数对话框 ', command=self.open_integer
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 open_integer 方法
        ttk.Button(self.master, text=' 输入浮点数对话框 ', command=self.open_float
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 open_integer 方法
        ttk.Button(self.master, text=' 输入字符串对话框 ', command=self.open_string
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 open_integer 方法
    def open_integer(self):          # 调用 askinteger 函数生成一个让用户输入整数的对话框
        print(simpledialog.askinteger(" 猜糖果 ", " 你猜我手上有几个糖果 :",
            initialvalue=3, minvalue=1, maxvalue=10))
    def open_float(self):            # 调用 askfloat 函数生成一个让用户输入浮点数的对话框
        print(simpledialog.askfloat(" 猜体重 ", " 你猜我我体重多少公斤 :",
            initialvalue=27.3, minvalue=10, maxvalue=50))
    def open_string(self):           # 调用 askstring 函数生成一个让用户输入字符串的对话框
        print(simpledialog.askstring(" 猜名字 ", " 你猜我叫什么名字 :",
            initialvalue='Charlie'))
root = Tk(); root.title(" 输入对话框测试 "); App(root); root.mainloop()
```

Tkinter 颜色选择对话框（colorchooser 模块）

colorchooser 模块提供了用于生成颜色选择对话框的 askcolor() 函数，该函数可指定如下选项：

- parent：指定该对话框的属主窗口。
- title：指定该对话框的标题。
- color：指定该对话框初始选择的颜色。

```
from tkinter import *; from tkinter import ttk; from tkinter import colorchooser        # 导入 colorchooser
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        ttk.Button(self.master, text=' 选择颜色 ', command=self.choose_color # 绑定 choose_color 方法
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 创建按钮，并为之绑定事件处理函数
    def choose_color(self):          # 调用 askcolor 函数获取选中的颜色
        print(colorchooser.askcolor(parent=self.master, title=' 选择画笔颜色 ', color = 'blue')) # 初始颜色
root = Tk(); root.title(" 颜色对话框测试 "); App(root); root.mainloop()
```

Tkinter 文件对话框（filedialog 模块）

filedialog 模块下提供了各种用于生成文件对话框的工具函数，有些返回用户所选择文件的路径，有些直接返回用户所选择文件的输入 / 输出流：

- askopenfile()：打开单个文件对话框，返回所选择文件的文件流，可通过它来读取文件内容。
- askopenfiles()：打开多个文件对话框，返回多个所选择文件的文件流列表，通过它们来读取文件内容。
- askopenfilename()：打开单个文件对话框，返回所选择文件的文件路径。
- askopenfilenames()：打开多个文件对话框，返回多个所选择文件的文件路径组成的元组。
- asksaveasfile()：保存文件对话框，返回所选择文件的文件输出流，可通过它写入数据。
- asksaveasfilename()：保存文件对话框，返回所选择文件的文件路径。
- askdirectory()：打开目录对话框。

这些工具函数支持如下选项：

- defaultextension：指定默认扩展名。用户未输入扩展名时，系统会默认添加指定的扩展名。
- filetypes：指定在文件对话框中能查看的文件类型。可指定多个文件类型（“*”表示所有文件）。
- initialdir：指定初始打开的目录。
- initialfile：指定所选择的文件。
- parent：指定该对话框的属主窗口。
- title：指定对话框的标题。
- multiple：指定是否允许多选。
- mustexist：（对于打开目录对话框）指定是否只允许打开已存在的目录。

```
from tkinter import *; from tkinter import ttk; from tkinter import filedialog
class App:
    filetype=[(" 文本文件 ", "*.txt"), ('Python 源文件 ', '*.py')]
    def __init__(self, master):
        self.master = master; self.initWidgets()
    def initWidgets(self):
        ttk.Button(self.master, text=' 打开单个文件 ', command=self.open_file
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 open_file 方法
        ttk.Button(self.master, text=' 打开多个文件 ', command=self.open_files
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 open_files 方法
        ttk.Button(self.master, text=' 获取单个打开文件的文件名 ', command=self.open_filename
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 open_filename 方法
        ttk.Button(self.master, text=' 获取多个打开文件的文件名 ', command=self.open_filenames
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 open_filenames 方法
        ttk.Button(self.master, text=' 获取保存文件 ', command=self.save_file
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 save_file 方法
        ttk.Button(self.master, text=' 获取保存文件的文件名 ', command=self.save_filename
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 save_filename 方法
        ttk.Button(self.master, text=' 打开路径 ', command=self.open_dir
            ).pack(side=LEFT, ipadx=5, ipady=5, padx= 10)  # 绑定 open_dir 方法
    def open_file(self):              # 调用 askopenfile 方法获取单个打开的文件
        print(filedialog.askopenfile(title=' 打开单个文件 ',initialdir='g:/',filetypes=self.filetype))
    def open_files(self):             # 调用 askopenfile 方法获取多个打开的文件
        print(filedialog.askopenfiles(title=' 打开多个文件 ',initialdir='g:/', filetypes=self.filetype))
    def open_filename(self):          # 调用 askopenfilename 方法获取单个文件的文件名
        print(filedialog.askopenfilename(title=' 打开单个文件 ',initialdir='g:/', filetypes=self.filetype))
    def open_filenames(self):         # 调用 askopenfilenames 方法获取多个文件的文件名
        print(filedialog.askopenfilenames(title=' 打开多个文件 ',initialdir='g:/', filetypes=self.filetype))
    def save_file(self):              # 调用 asksaveasfile 方法保存文件
        print(filedialog.asksaveasfile(title=' 保存文件 ',initialdir='g:/',, filetypes=self.filetype))
    def save_filename(self):          # 调用 asksaveasfilename 方法获取保存文件的文件名
        print(filedialog.asksaveasfilename(title=' 保存文件 ',initialdir='g:/', filetypes=self.filetype))
    def open_dir(self):print(filedialog.askdirectory(title=' 打开目录 ',initialdir='g:/'))    # 打开目录
root = Tk(); root.title(" 文件对话框测试 "); App(root); root.mainloop()
```

Tkinter 消息框（messagebox 模块）

messagebox 模块下提供了大量工具函数来生成各种消息框。在默认情况下，调用 messagebox 的工具函数时只要设置提示区的字符串即可，图标区的图标、按钮区的按钮都有默认设置。如果有必要，可通过如下两个选项来定制图标和按钮：

- icon：定制图标（字符串）。支持 error、info、question、warning 这几个选项值。
- type：定制按钮（字符串）。支持 abortretryignore（取消、重试、忽略）、ok（确定）、okcancel（确定、取消）、retrycancel（重试、取消）、yesno（是、否）、yesnocancel（是、否、取消）这些选项值。

```
from tkinter import *; from tkinter import ttk; from tkinter import messagebox as msgbox
class App:
    def __init__(self, master):
        self.master = master; self.initWidgets()
    def initWidgets(self):
        topF = Frame(self.master); topF.pack(fill=BOTH)
        lf1 = ttk.Labelframe(topF, text=' 请选择图标类型 ')
        lf1.pack(side=LEFT, fill=BOTH, expand=YES, padx=10, pady=5); i = 0
        self.icons = [None, "error", "info", "question", "warning"]; self.iconVar = IntVar()
        for icon in self.icons:          # 使用循环创建多个 Radiobutton，并放入 Labelframe 中
            Radiobutton(lf1, text = icon if icon is not None else ' 默认 ', value=i, variable=self.iconVar
                ).pack(side=TOP, anchor=W); i += 1
        lf2 = ttk.Labelframe(topF, text=' 请选择按钮类型 ')
        lf2.pack(side=LEFT,fill=BOTH, expand=YES, padx=10, pady=5); i = 0
          self.types = [None, "abortretryignore", "ok", "okcancel", "retrycancel", "yesno", "yesnocancel"];
self.typeVar = IntVar()
        for tp in self.types:            # 使用循环创建多个 Radiobutton，并放入 Labelframe 中
            Radiobutton(lf2, text= tp if tp is not None else ' 默认 ', value=i, variable=self.typeVar
                ).pack(side=TOP, anchor=W); i += 1
        self.iconVar.set(0);self.typeVar.set(0)
        bottomF = Frame(self.master); bottomF.pack(fill=BOTH)
        btn1 = ttk.Button(bottomF, text="showinfo", command=self.showinfo_clicked)
        btn1.pack(side=LEFT, fill=X, ipadx=5, ipady=5, pady=5, padx=5)
        btn2 = ttk.Button(bottomF, text="showwarning", command=self.showwarning_clicked)
        btn2.pack(side=LEFT, fill=X, ipadx=5, ipady=5, pady=5, padx=5)
        btn3 = ttk.Button(bottomF, text="showerror", command=self.showerror_clicked)
        btn3.pack(side=LEFT, fill=X, ipadx=5, ipady=5, pady=5, padx=5)
        btn4 = ttk.Button(bottomF, text="askquestion", command=self.askquestion_clicked)
        btn4.pack(side=LEFT, fill=X, ipadx=5, ipady=5, pady=5, padx=5)
        btn5 = ttk.Button(bottomF, text="askokcancel", command=self.askokcancel_clicked)
        btn5.pack(side=LEFT, fill=X, ipadx=5, ipady=5, pady=5, padx=5)
        btn6 = ttk.Button(bottomF, text="askyesno", command=self.askyesno_clicked)
        btn6.pack(side=LEFT, fill=X, ipadx=5, ipady=5, pady=5, padx=5)
        btn7 = ttk.Button(bottomF, text="askyesnocancel", command=self.askyesnocancel_clicked)
        btn7.pack(side=LEFT, fill=X, ipadx=5, ipady=5, pady=5, padx=5)
        btn8 = ttk.Button(bottomF, text="askretrycancel", command=self.askretrycancel_clicked)
        btn8.pack(side=LEFT, fill=X, ipadx=5, ipady=5, pady=5, padx=5)
    def showinfo_clicked(self): print(msgbox.showinfo("Info", "showinfo 测试 .",
            icon=self.icons[self.iconVar.get()], type=self.types[self.typeVar.get()]))
    def showwarning_clicked(self): print(msgbox.showwarning("Warning", "showwarning 测试 .",
            icon=self.icons[self.iconVar.get()], type=self.types[self.typeVar.get()]))
    def showerror_clicked(self): print(msgbox.showerror("Error", "showerror 测试 .",
            icon=self.icons[self.iconVar.get()], type=self.types[self.typeVar.get()]))
    def askquestion_clicked(self): print(msgbox.askquestion("Question", "askquestion 测试 .",
            icon=self.icons[self.iconVar.get()], type=self.types[self.typeVar.get()]))
    def askokcancel_clicked(self): print(msgbox.askokcancel("OkCancel", "askokcancel 测试 .",
            icon=self.icons[self.iconVar.get()], type=self.types[self.typeVar.get()]))
    def askyesno_clicked(self): print(msgbox.askyesno("YesNo", "askyesno 测试 .",
            icon=self.icons[self.iconVar.get()], type=self.types[self.typeVar.get()]))
    def askyesnocancel_clicked(self): print(msgbox.askyesnocancel("YesNoCancel", "askyesnocancel 测试 .",
            icon=self.icons[self.iconVar.get()], type=self.types[self.typeVar.get()]))
    def askretrycancel_clicked(self): print(msgbox.askretrycancel("RetryCancel", "askretrycancel 测试 .",
            icon=self.icons[self.iconVar.get()], type=self.types[self.typeVar.get()]))
root = Tk(); root.title(" 消息框测试 "); App(root); root.mainloop()
```

Tkinter Menu 菜单（窗口菜单和右键菜单）

Tkinter 为菜单提供了 Menu 类，该类可以搞定所有菜单相关内容（菜单，菜单条，右键菜单）。调用构造方法创建菜单后，可添加各种菜单项：add_command（菜单项）、add_checkbutton（复选框菜单项）、add_radiobutton（单选钮菜单项）、add_separator（菜单分隔条）。添加菜单项支持以下选项：

- label：指定菜单项的文本。
- command：指定为菜单项绑定的事件处理方法。
- image：指定菜单项的图标。
- compound：指定在菜单项中图标位于文字的哪个方位。

窗口菜单

```
from tkinter import *
from tkinter import ttk
from tkinter import messagebox as msgbox
class App:
    def __init__(self, master):
        self.master = master
        self.init_menu()
    def init_menu(self):    # 创建菜单
        self.master.filenew_icon = PhotoImage(file='images/filenew.png')     #“新建”图标
        self.master.fileopen_icon = PhotoImage(file='images/fileopen.png')  #“打开”图标
        menubar = Menu(self.master); self.master['menu'] = menubar          # 创建并添加菜单条
        file_menu = Menu(menubar, tearoff=0)        # menubar 中创建 file_menu 菜单
        menubar.add_cascade(label=' 文件 ', menu=file_menu)          # 用 add_cascade 添加 file_menu
        lang_menu = Menu(menubar, tearoff=0)     # menubar 中创建 lang_menu 菜单
        menubar.add_cascade(label=' 选择语言 ', menu=lang_menu)   # 用 add_cascade 添加 lang_menu
        file_menu.add_command(label=" 新建 ", image=self.master.filenew_icon, compound=LEFT,
            command = None)        # 用 add_command 为 file_menu 添加菜单项
        file_menu.add_command(label=" 打开 ", image=self.master.fileopen_icon, compound=LEFT,
            command = None)        # 用 add_command 为 file_menu 添加菜单项
        file_menu.add_separator() # 用 add_command 为 file_menu 添加分隔条
        sub_menu = Menu(file_menu, tearoff=0)     # 为 file_menu 创建子菜单
        file_menu.add_cascade(label=' 选择性别 ', menu=sub_menu)   # 用 add_cascade 添加 sub_menu
        self.genderVar = IntVar()    # 定义用于接收“性别”的变量
        for i, im in enumerate([' 男 ', ' 女 ', ' 保密 ']):  # 循环为 sub_menu 子菜单添加菜单项
            # 用 add_radiobutton 为 sub_menu 子菜单添加单选菜单项，绑定同一个变量，说明它们是一组
            sub_menu.add_radiobutton(label=im, command=self.choose_gender, variable=self.genderVar, value=i)
        self.langVars = [StringVar(), StringVar(), StringVar(), StringVar()]
        for i, im in enumerate(('Python', 'Kotlin','Swift', 'Java')):        # 循环为 lang_menu 菜单添加菜单项
            # 使用 add_add_checkbutton 方法为 lang_menu 菜单添加多选菜单项
            lang_menu.add_checkbutton(label=im, command=self.choose_lang, onvalue=im, variable=self.langVars[i])
    def choose_gender(self):
        msgbox.showinfo(message=(' 选择的性别为 : %s' % self.genderVar.get()))
    def choose_lang(self):
        rt_list = [e.get() for e in self.langVars]
        msgbox.showinfo(message=(' 选择的语言为 : %s' % ','.join(rt_list)))
root = Tk(); root.title(" 菜单测试 "); root.geometry('400x200')
root.resizable(width=False, height=False)          # 禁止改变窗口大小
App(root)
root.mainloop()
```

功能更全面的菜单示例（添加一个工具条）：

```
from tkinter import *; from tkinter import ttk; from collections import OrderedDict
class App:
    def __init__(self, master):
        self.master = master; self.initWidgets()
    def initWidgets(self):
        self.init_icons()      # 初始化菜单、工具条用到的图标
        self.init_menu()       # 调用 init_menu 初始化菜单
        self.init_toolbar()    # 调用 init_toolbar 初始化工具条
        leftframe = ttk.Frame(self.master, width=40); leftframe.pack(side=LEFT, fill=Y)          # 左边容器
        lb = Listbox(leftframe, font=('Courier New', 20)); lb.pack(fill=Y, expand=YES)      # 左边窗口放 Listbox
        for s in ('Python', 'Ruby', 'Swift', 'Kotlin', 'Java'): lb.insert(END, s)
        mainframe = ttk.Frame(self.master); mainframe.pack(side=LEFT, fill=BOTH)            # 右边容器
        text = Text(mainframe, width=40, font=('Courier New', 16)); text.pack(side=LEFT, fill=BOTH)
        scroll = ttk.Scrollbar(mainframe,command=text.yview); scroll.pack(side=LEFT,fill=Y)
        text.configure(yscrollcommand=scroll.set)   # 设置滚动条与 text 组件关联
    def init_menu(self):    # 创建菜单
        ' 初始化菜单的方法 '
        menus = (' 文件 ', ' 编辑 ', ' 帮助 ')    # 定义菜单条所包含的 3 个菜单
        # 定义菜单数据，每项对应一个菜单项，后面元组第一项菜单图标，第二项菜单事件处理函数
        items = (OrderedDict([(' 新建 ', (self.master.filenew_icon, None)),(' 打开 ', (self.master.fileopen_
icon, None)),(' 保存 ', (self.master.save_icon, None)),(' 另存为 ...', (self.master.saveas_icon, None)),('–
1', (None, None)),(' 退 出 ', (self.master.signout_icon, None)),]),OrderedDict([(' 撤 销 ',(None, None)),('
重做 ',(None, None)),('–1',(None, None)),(' 剪切 ',(None, None)),(' 复制 ',(None, None)),(' 粘贴 ',(None,
None)),(' 删除 ',(None, None)),(' 选择 ',(None, None)),('–2',(None, None)),(' 更多 ', OrderedDict([(' 显示
数据 ',(None, None)),(' 显示统计 ',(None, None)),(' 显示图表 ',(None, None))])]),OrderedDict([(' 帮助主
题 ',(None, None)),('–1',(None, None)),(' 关于 ', (None, None))]))
        menubar = Menu(self.master);self.master['menu'] = menubar          # 创建并配置菜单条
        for i, m_title in enumerate(menus):  # 遍历 menus 元组
            m = Menu(menubar, tearoff=0); menubar.add_cascade(label=m_title, menu=m) # 创建并添加菜单
            tm = items[i]      # 将当前正在处理的菜单数据赋值给 tm
            for label in tm:    # 遍历 OrderedDict, 默认只遍历它的 key
                if isinstance(tm[label], OrderedDict):    # 如果 value 又是 OrderedDict，说明是二级菜单
                    sm = Menu(m, tearoff=0); m.add_cascade(label=label, menu=sm)  # 创建并添加子菜单
                    sub_dict = tm[label]
                    for sub_label in sub_dict:    # 再次遍历子菜单对应的 OrderedDict，默认只遍历它的 key
                        if sub_label.startswith('–'): sm.add_separator()      # 添加分隔条
                        else: sm.add_command(label=sub_label,image=sub_dict[sub_label][0],
                              command=sub_dict[sub_label][1], compound=LEFT) # 添加菜单项
                elif label.startswith('–'): m.add_separator()       # 添加分隔条
                else: m.add_command(label=label,image=tm[label][0],
                      command=tm[label][1], compound=LEFT)          # 添加菜单项
    def init_icons(self):    # 生成所有需要的图标
        self.master.filenew_icon = PhotoImage(file='images/filenew.png')
        self.master.fileopen_icon = PhotoImage(file='images/fileopen.png')
        self.master.save_icon = PhotoImage(file='images/save.png')
        self.master.saveas_icon = PhotoImage(file='images/saveas.png')
        self.master.signout_icon = PhotoImage(file='images/signout.png')
    def init_toolbar(self):  # 生成工具条
        toolframe = Frame(self.master, height=20, bg='lightgray'); toolframe.pack(fill=X) # 添加工具条
        frame = ttk.Frame(toolframe); frame.pack(side=LEFT)         # 添加工具条按钮容器
        for i, e in enumerate(dir(self.master)):         # 遍历 self.master，根据图标创建工具栏按钮
            if e.endswith('_icon'):      # 只处理属性名以 _icon 结尾的属性（这些属性都是图标）
                ttk.Button(frame, width=20, image=getattr(self.master, e), command=None
                    ).grid(row=0, column=i, padx=1, pady=1, sticky=E)
root = Tk(); root.title(" 菜单测试 "); root.resizable(width=False, height=True); App(root); root.mainloop()
```

● 右键菜单

实现右键菜单很简单，程序只要先创建菜单，然后为目标组件的右键单击事件绑定处理函数，当用户单击鼠标右键时，调用菜单的 post() 方法即可在指定位置弹出右键菜单。

```
from tkinter import *
from tkinter import ttk
from collections import OrderedDict
class App:
    def __init__(self, master):
        self.master = master
        self.initWidgets()
    def initWidgets(self):
        self.text = Text(self.master, height=12, width=60, foreground='darkgray', font=(' 微软雅黑 ', 12),
            spacing2=8, spacing3=12) ; self.text.pack()
        self.text.insert(END, 'C 语言中文网成立于 2012 年初，目前已经运营了将近 5 年，我们致力于分享精品教程，帮助对编程感兴趣的读者 \n')
        self.text.bind('<Button-3>',self.popup)        # 为 text 组件的右键单击事件绑定处理方法
        self.popup_menu = Menu(self.master,tearoff = 0)    # 创建 Menu 对象，准备作为右键菜单
        self.my_items = (OrderedDict([(' 超  大 ', 16), (' 大 ',14), (' 中 ',12), (' 小 ',10), (' 超  小 ',8)]),
OrderedDict([(' 红色 ','red'), (' 绿色 ','green'), (' 蓝色 ', 'blue')]))
        i = 0
        for k in [' 字体大小 ',' 颜色 ']:
            m = Menu(self.popup_menu, tearoff = 0)
            self.popup_menu.add_cascade(label=k ,menu = m)        # 添加子菜单
            for im in self.my_items[i]:            # 遍历 OrderedDict 的 key（默认就是遍历 key）
                m.add_command(label=im, command=self.handlerAdaptor(self.choose, x=im))
            i += 1
    def popup(self, event): self.popup_menu.post(event.x_root,event.y_root)        # 指定位置显示菜单
    def choose(self, x):
        if x in self.my_items[0].keys():        # 如果用户选择修改字体大小的子菜单项，改变字体大小
            self.text['font'] = (' 微软雅黑 ', self.my_items[0][x])
        if x in self.my_items[1].keys():        # 如果用户选择修改颜色的子菜单项，改变颜色
            self.text['foreground'] = self.my_items[1][x]
    def handlerAdaptor(self, fun,**kwds): return lambda fun=fun, kwds=kwds: fun(**kwds)
root = Tk()
root.title(" 右键菜单测试 ")
App(root)
root.mainloop()
```

tkinter Canvas 画布

Canvas 组件来实现绘图。既可在 Canvas 中绘制直线、矩形、椭圆等各种几何图形，也可绘制图片、文字、UI 组件（如 Button）等。Canvas 允许重新改变这些图形项（Tkinter 将程序绘制的所有东西统称为 item）的属性，比如改变其坐标、外观等。

```
from tkinter import *
root = Tk()        # 创建窗口
cv = Canvas(root, background='white')   # 创建并添加 Canvas
cv.pack(fill=BOTH, expand=YES)
cv.create_rectangle(30, 30, 200, 200,outline='red',stipple = 'question', fill="red",width=5)   # 绘制矩形
cv.create_oval(240, 30, 330, 200,outline='yellow', fill='pink', width=4)       # 绘制圆形
root.mainloop()
```

Canvas 的坐标系统是绘图的基础，点 (0,0) 位于 左上角，X 轴水平向右延伸，Y 轴垂直向下延伸。

函数	描述
create_line()	绘制直线，需要指定两个点的坐标：直线的起点和终点。
create_rectangle()	绘制矩形，需要指定两个点的坐标：矩形左上角点和右下角点的坐标。
create_oval()	绘制椭圆（包括圆），需要指定两个点的坐标：左上角点和右下角点的坐标来确定一个矩形，绘制该矩形的内切椭圆。
create_arc()	绘制弧，需要指定左上角和右下角两个点的坐标，默认绘制从原点（0）开始，逆时针旋转 90° 的一段弧。可通过 start 改变起始角度，可通过 extent 改变转过的角度。
create_bitmap()	绘制位图，只要指定一个坐标点，用于指定目标元素的绘制位置。
create_image()	绘制图片，只要指定一个坐标点，用于指定目标元素的绘制位置。
create_polygon()	绘制多边形，需要指定多个点的坐标来作为多边形的多个定点。
create_text()	绘制文字，只要指定一个坐标点，用于指定目标元素的绘制位置。
create_window()	绘制组件，只要指定一个坐标点，用于指定目标元素的绘制位置。

在绘制这些图形时可指定如下**选项**：

选项	描述
fill	指定填充颜色（默认不填充）。
outline	指定边框颜色。
width	指定边框宽度（默认为 1）。
dash	指定边框使用虚线。支持整数（虚线中线段的长度）和元组（如 (5,2,3) 表示虚线中线段长度 5，间隔 2，虚线长度 3）。
stipple	使用位图平铺进行填充。可与 fill 选项结合使用（fill 指定位图的颜色）。
style	指定绘制弧的样式（仅对 create_arc 方法起作用）。支持 PIESLICE（扇形）、CHORD（弓形）、ARC（仅绘制弧）。
start	指定绘制弧的起始角度（仅对 create_arc 方法起作用）。
extent	指定绘制弧的角度（仅对 create_arc 方法起作用）。
arrow	指定绘制直线时两端是否有箭头。支持 NONE（两端无箭头）、FIRST（开始端有箭头）、LAST（结束端有箭头）、BOTH（两端都有箭头）。
arrowshape	指定箭头形状。字符串（如 "20 20 10" 表示填充长度、箭头长度、箭头宽度）。
joinstyle	指定直接连接点的风格（仅对绘制直线和多向形有效）。支持 METTER、ROUND、BEVEL 。
anchor	指定绘制文字、GUI 组件的位置（仅对 create_text、create_window 方法有效）。
justify	指定文字的对齐方式（仅对 create_text 方法有效）。支持 CENTER、LEFT、RIGHT 。

以下示范绘制不同的图形，分别使用不同的边框、不同的填充效果：

```
from tkinter import *
root = Tk(); root.title(' 绘制图形项 ')        # 创建窗口，设置标题
cv = Canvas(root, background='white', width=830, height=830)    # 创建并添加 Canvas
cv.pack(fill=BOTH, expand=YES)
columnFont = (' 微软雅黑 ', 18); titleFont = (' 微软雅黑 ', 20, 'bold')
for i, st in enumerate([' 默认 ', ' 指定边宽 ', ' 指定填充 ', ' 边框颜色 ', ' 位图填充 ']):  # 循环绘制文字
    cv.create_text((130 + i * 140, 20),text = st, font = columnFont, fill='gray', anchor = W, justify = LEFT)
cv.create_text(10, 60, text = ' 绘制矩形 ', font = titleFont, fill='magenta', anchor = W, justify = LEFT)
options = [(None, None, None, None, None, None), (4, None, None, None, BOTH, (20, 40, 10)), (4,
'pink', None, None, FIRST, (40, 40, 10)), (4, 'pink', 'blue', None LAST, (60, 50, 10)),(4, 'pink', 'blue',
'error', None, None)]     # 定义列表（边框宽度、填充色、边框颜色、位图填充、箭头风格，箭头形状）
for i, op in enumerate(options):  # 采用循环绘制 5 个矩形
    cv.create_rectangle(130 + i * 140, 50, 240 + i * 140, 120,
            width = op[0], fill = op[1], outline = op[2],  stipple = op[3])
cv.create_text(10, 160, text = ' 绘制椭圆 ', font = titleFont, fill='magenta', anchor = W, justify = LEFT)
for i, op in enumerate(options):  # 采用循环绘制 5 个椭圆
    cv.create_oval(130 + i * 140, 150, 240 + i * 140, 220,
            width = op[0],  fill = op[1], outline = op[2], stipple = op[3])
cv.create_text(10, 260, text = ' 绘制多边形 ', font = titleFont, fill='magenta', anchor = W, justify = LEFT)
for i, op in enumerate(options):  # 采用循环绘制 5 个多边形
    cv.create_polygon(130 + i * 140, 320, 185 + i * 140, 250, 240 + i * 140, 320,
            width = op[0], fill = '' if not op[1] else op[1], outline = 'black' if not op[2] else op[2], stipple = op[3])
cv.create_text(10, 360, text = ' 绘制扇形 ', font = titleFont, fill='magenta', anchor = W, justify = LEFT)
for i, op in enumerate(options):  # 采用循环绘制 5 个扇形
    cv.create_arc(130 + i * 140, 350, 240 + i * 140, 420,
            width = op[0],  fill = op[1], outline = op[2], stipple = op[3])
cv.create_text(10, 460, text = ' 绘制弓形 ', font = titleFont, fill='magenta', anchor = W, justify = LEFT)
for i, op in enumerate(options):  # 采用循环绘制 5 个弓形
    cv.create_arc(130 + i * 140, 450, 240 + i * 140, 520, start = 30, extent = 60, style = CHORD,
            width = op[0], fill = op[1],  outline = op[2], stipple = op[3])
cv.create_text(10, 560, text = ' 仅绘弧 ', font = titleFont, fill='magenta', anchor = W, justify = LEFT)
for i, op in enumerate(options):  # 采用循环绘制 5 个弧
    cv.create_arc(130 + i * 140, 550, 240 + i * 140, 620,start = 30, extent = 60,  style = ARC,
            width = op[0],  fill = op[1],  outline = op[2],  stipple = op[3])
cv.create_text(10, 660, text = ' 绘制直线 ', font = titleFont, fill='magenta', anchor = W, justify = LEFT)
for i, op in enumerate(options):  # 采用循环绘制 5 个直线
    cv.create_line(130 + i * 140, 650, 240 + i * 140, 720,
            width = op[0], fill = op[2], stipple = op[3], arrow = op[4], arrowshape = op[5])
cv.create_text(10, 760, text = ' 绘制位图、图片、组件 ', font = titleFont, fill='magenta', anchor = W, justify = LEFT)
# 定义包括 create_bitmap, create_image, create_window 三个方法的数组
funcs = [Canvas.create_bitmap, Canvas.create_image, Canvas.create_window]
# 为上面 3 个方法定义选项
items = [{'bitmap' : 'questhead'}, {'image':PhotoImage(file='images/fklogo.gif')},
    {'window':Button(cv,text = ' 单击我 ', padx=10, pady=5, command = lambda :print(' 按钮单击 ')),'anchor': W}]
for i, func in enumerate(funcs):  func(cv, 450 + i * 140, 780, **items[i])
root.mainloop()
```

- 操作图形项

在 Canvas 中通过 create_xxx 方法绘制图形项之后，这些图形项并不是完全静态的图形，每个图形项都是一个独立的对象，程序完全可以动态地修改、删除这些图形项。

Canvas 以“堆叠”的形式来管理这些图形项，先绘制的图形项位于“堆叠”的下面，后绘制的图形项位于“堆叠”的上面。为了修改、删除图形项，需要先获得图形项的引用，获得这些图形项的引用有两种方式：

- **通过图形项的 id**（create_xxx 方法的返回值）。一般来说，create_xxx() 会依次返回 1、2、3 等整数作为图形项的 id。
- **通过图形项的 tag**（标签）。create_xxx() 方法绘图时可传入一个 tags 选项添加一个或多个 tag（标签）。还可以调用方法为图形项添加 tag、删除 tag，这些 tag 也相当于该图形项的标识，程序完全可以根据 tag 来获取图形项。

在获取图形项之后，就可以对图形项进行修改和删除图形项。

	函数	描述
操作图形项标签	addtag_aboove(self, newtag, tagOrId)	为 tagOrId 对应图形项的上一个图形项添加新 tag。
	addtag_all(self, newtag)	为所有图形项添加新 tag。
	addtag_below(self, newtag, tagOrId)	为 tagOrId 对应图形项的下一个图形项添加新 tag。
	addtag_closest(self, newtag, x, y)	为和 x、y 点最接近的图形项添加新 tag。
	addtag_enclosed(self, newtag, x1, y1, x2, y2)	为指定矩形区域内最上面的图形项添加新 tag。其中 x1、y1 为矩形区域的左上角坐标；x2、y2 为矩形区域的右下角坐标。
	addtag_overlapping(self, newtag, x1, y1, x2, y2)	为与指定矩形区域重叠的最上面的图形项添加 tag。
	addtag_withtag(self, newtag, tagOrId)	为 tagOrId 对应图形项添加新 tag。
	dtag(self, *args)	删除指定图形项的 tag。
	gettags(self, *args)	获取指定图形项的所有 tag。
操作图形项	find_all(self)	返回全部图形项。
	find_above(self, tagOrId)	返回 tagOrId 对应图形项的上一个图形项。
	find_below(self, tagOrId)	返回 tagOrId 对应图形项的下一个图形项。
	find_closest(self, x, y)	返回和 x 、y 点最接近的图形项。
	find_enclosed(self, x1, y1, x2, y2)	返回位于指定矩形区域内最上面的图形项。
	find_overlapping(self, x1, y1, x2, y2)	返回与指定矩形区域重叠的最上面的图形项。
	find_withtag(self, tagOrId)	返回 tagOrId 对应的全部图形项。
	tag_lower(self, *args) 或 lower	将 args 的第一个参数对应的图形项移到“堆叠”的最下面。也可额外指定一个参数，代表移动到指定图形项的下面。
	tag_raise(self, *args) 或 lift	将 args 的第一个参数对应的图形项移到“堆叠”的最上面。也可额外指定一个参数，代表移动到指定图形项的上面。
	itemcget(self, tagOrId, option)	获取 tagOrId 对应图形项的 option 选项值。
	itemconfig(self, tagOrId, cnf=None, **kw) 或 itemconfigure	为 tagOrId 对应图形项配置选项。
	coords(self, *args)	重设图形项的大小和位置。
	move(self, *args)	移动图形项，但不能改变大小。简单来说，就是在图形项的 x、y 基础上加上新的 mx、my 参数。
	scale(self, *args)	缩放图形项。该方法的 args 参数要传入 4 个值，其中前两个值指定缩放中心；后两个值指定 x、y 方向的缩放比。
	delete(self, *args)	删除指定 id 或 tag 对应的全部图形项。
	dchars(self, *args)	删除文字图形项中间的部分文字。
	tag_bind(self,tag,event,fun, add=false)	为指定图形项绑定事件处理函数或方法。 tag 为图形项，event 为事件，fun 为处理函数， add 为 true 时，添加事件，add 为 false 时，替代原事件

操作图形项标签

```
from tkinter import *
root = Tk()                    # 创建窗口
root.title(' 操作标签 ')       # 设置窗口标题
# 创建并添加 Canvas
cv = Canvas(root, background='white', width=620, height=250)
cv.pack(fill=BOTH, expand=YES)
# 绘制矩形框（id=1）并指定标签（'t1', 't2', 't3', 'tag4'）
rt = cv.create_rectangle(40, 40, 300, 220, outline='blue', width=2,  tag = ('t1', 't2', 't3', 'tag4'))
# 绘制椭圆（id=2）并指定标签（'g1', 'g2', 'g3', 'tag4'）
oval = cv.create_oval(350, 50, 580, 200, fill='yellow', width=0, tag = ('g1', 'g2', 'g3', 'tag4'))
print(rt)                          # 访问图形项的 id：1
print(oval)                        # 访问图形项的 id：2
print(cv.find_withtag('tag4'))     # 根据 tag(tag4) 获得所有 id：(1, 2)
print(cv.gettags(rt))              # 获取图形项 (rt) 的 tag：  ('t1', 't2', 't3', 'tag4')
print(cv.gettags(2))               # 获取图形项 (id=2) 的 tag：  ('g1', 'g2', 'g3', 'tag4')
cv.dtag(1, 't1')                   # 删除图形项 (id=1) 的 tag(t1)
print(cv.gettags(1))                       # ('tag4', 't2', 't3')
print(cv.gettags(2))                       # ('g1', 'g2', 'g3', 'tag4')
cv.dtag(oval, 'g1')                # 删除图形项 (oval) 的 tag(g1)
print(cv.gettags(1))                       # ('tag4', 't2', 't3')
print(cv.gettags(2))                       # ('tag4', 'g2', 'g3')
cv.addtag_all('t5')                # 为所有图形项添加 tag(t5)
print(cv.gettags(1))                       # ('tag4', 't2', 't3', 't5')
print(cv.gettags(2))                       # ('tag4', 'g2', 'g3', 't5')
cv.addtag_withtag('t6', 'g2')      # 为图形项 (tag=g2) 添加 tag（t6）
print(cv.gettags(1))                       # ('tag4', 't2', 't3', 't5')
print(cv.gettags(2))                       # ('tag4', 'g2', 'g3', 't5', 't6')
cv.addtag_above('t7', 't2')        # 为图形项 (tag=t2) 上面的图形项添加 tag(t7)
print(cv.gettags(1))                       # ('tag4', 't2', 't3', 't5')
print(cv.gettags(2))                       # ('tag4', 'g2', 'g3', 't5', 't6', 't7')
cv.addtag_below('t8', 'g2')        # 为图形项 (tag=g2) 下面的图形项添加 tag(t8)
print(cv.gettags(1))                       # ('tag4', 't2', 't3', 't5', 't8')
print(cv.gettags(2))                       # ('tag4', 'g2', 'g3', 't5', 't6', 't7')
cv.addtag_closest('t9', 360, 90)   # 为最接近指定点 (360、90) 的图形项添加 tag(t9)
print(cv.gettags(1))                       # ('tag4', 't2', 't3', 't5', 't8')
print(cv.gettags(2))                       # ('tag4', 'g2', 'g3', 't5', 't6', 't7', 't9')
cv.addtag_closest('t10', 30, 30, 600, 240)     # 为位于指定区域 (30, 30, 600, 240) 内最上面的图形项添加 tag(t10)
print(cv.gettags(1))                               # ('tag4', 't2', 't3', 't5', 't8')
print(cv.gettags(2))                               # ('tag4', 'g2', 'g3', 't5', 't6', 't7', 't9', 't10')
cv.addtag_overlapping('t11', 250, 30, 400, 240)    # 为与指定区域 (250, 30, 400, 240) 内重合的最上面的图形项添加 tag(t11)
print(cv.gettags(1))                               # ('tag4', 't2', 't3', 't5', 't8', 't11')
print(cv.gettags(2))                               # ('tag4', 'g2', 'g3', 't5', 't6', 't7', 't9', 't10', 't11')
root.mainloop()
```

操作图形项

```
from tkinter import *; from tkinter import colorchooser; import threading
root = Tk();root.title(' 操作图形项 ')          # 创建窗口并添加 Canvas
cv = Canvas(root, background='white', width=400, height=350);cv.pack(fill=BOTH, expand=YES)
current = None             # current 变量用于保存当前选中的图形项
current_outline = None     # current_outline 变量用于保存当前选中的图形项的边框颜色
current_width = None       # current_width 变量用于保存当前选中的图形项的边框宽度
firstx = firsty = None     # 记录鼠标拖动的第一个点的 x、y 坐标
prev_select = None         # 记录前一次绘制的、代表选择区的虚线框
def show_current():        # 该函数用于高亮显示选中图形项（边框颜色会 red、yellow 之间切换）
    if current is not None:            # 如果当前选中项不为 None
        if cv.itemcget(current, 'outline') == 'red':cv.itemconfig(current, width=2,outline='yellow')
        else: cv.itemconfig(current, width=2,outline='red')
    global t
    t = threading.Timer(0.2, show_current);t.start()  # 通过定时器指定 0.2 秒之后执行 show_current 函数
```

```
def change_fill():          # 用户颜色选择框选择颜色，初始颜色为椭圆当前的填充色（fill 选项值）
    fill_color = colorchooser.askcolor(parent=root, title=' 选择填充颜色 ',color =  cv.itemcget(oval, 'fill'))
    if fill_color is not None: cv.itemconfig(oval, fill=fill_color[1])
def change_outline():    # 用户颜色选择框选择颜色，初始颜色为椭圆当前的边框色（outline 选项值）
    outline_color = colorchooser.askcolor(parent=root, title=' 选择边框颜色 ', color = cv.itemcget(oval, 'outline'))
    if outline_color is not None: cv.itemconfig(oval, outline=outline_color[1], width=4)
def select_handler(ct):
    global current, current_outline, current_width
    if ct is not None and len(ct) > 0:          # 如果 ct 元组包含了选中项
        ct = ct[0]
        if current is not None:         # 如果 current 对应的图形项不为空
            cv.itemconfig(current, outline=current_outline, width = current_width)       # 恢复边框
        current_outline = cv.itemcget(ct, 'outline');current_width = cv.itemcget(ct, 'width') # 获取边框信息
        current = ct            # 使用 current 保存当前选中项
def click_handler(event):
    ct = cv.find_closest(event.x, event.y)  # 获取当前选中的图形项
    select_handler(ct)                      # 调用 select _handler 处理选中图形项
def click_select():
    cv.unbind('<B1-Motion>');cv.unbind('<ButtonRelease-1>')     # 取消“框选”绑定的事件处理函数
    cv.bind('<Button-1>', click_handler)              # 为“点选”绑定鼠标点击事件处理函数
def drag_handler(event):
    global firstx, firsty, prev_select
    if firstx is None and firsty is None:firstx, firsty = event.x, event.y   # 开始拖动时记录鼠标初始位置
    leftx,lefty = min(firstx,event.x),min(firsty,event.y)；rightx,righty=max(firstx,event.x),max(firsty,event.y)
    if prev_select is not None:cv.delete(prev_select)                  # 删除上一次绘制的虚线选择框
    prev_select = cv.create_rectangle(leftx, lefty, rightx, righty, dash=2)     # 重新绘制虚线选择框
def release_handler(event):
    global firstx, firsty
    if prev_select is not None: cv.delete(prev_select)
    if firstx is not None and firsty is not None:
        leftx,lefty=min(firstx,event.x),min(firsty,event.y); rightx,righty=max(firstx,event.x),max(firsty,event.y)
        firstx = firsty = None
        ct = cv.find_enclosed(leftx, lefty, rightx, righty)         # 获取当前选中的图形项
        select_handler(ct)                                          # 调用 select _handler 处理选中图形项
def rect_select():
    cv.unbind('<Button-1>')         # 取消为“点选”绑定的事件处理函数
    cv.bind('<B1-Motion>', drag_handler)                            # 为“框选”绑定鼠标拖动事件处理函数
    cv.bind('<ButtonRelease-1>', release_handler)                   # 为“框选”绑定鼠标释放的事件处理函数
t = threading.Timer(0.2, show_current);t.start()   # 通过定时器指定 0.2 秒之后执行 show_current 函数
rect = cv.create_rectangle(30, 30, 250, 200,fill='magenta', width='0')        # 创建矩形
oval = cv.create_oval(180, 50, 380, 180, fill='yellow', width='0')            # 创建椭圆
circle = cv.create_oval(120, 150, 300, 330, fill='pink', width='0')           # 创建圆
bottomF = Frame(root); bottomF.pack(fill=X,expand=True)          #
liftbn=Button(bottomF,text=' 向上 ',command=lambda:cv.tag_raise(oval,cv.find_above(oval)))
liftbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)         # 向上按钮，将椭圆移动到它上面的 item 之上
lowerbn = Button(bottomF, text=' 向下 ', command=lambda : cv.tag_lower(oval, cv.find_below(oval)))
lowerbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)    # 向下按钮，将椭圆移动到它下面的 item 之下
fillbn = Button(bottomF, text=' 改变填充色 ', command=change_fill)
fillbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)         # 改变填充色按钮，触发 change_fill 函数
outlinebn = Button(bottomF, text=' 改变边框色 ', command=change_outline)
outlinebn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)  # 改变边框色按钮，触发 change_outline 函数
movebn = Button(bottomF, text=' 向下移动 ', command=lambda : cv.move(oval, 15, 10))
movebn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)    # 向下移动按钮，调用 move 方法移动图形项
coordsbn = Button(bottomF, text=' 位置复位 ', command=lambda : cv.coords(oval, 180, 50, 380, 180))
coordsbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)  # 位置复位按钮，重设图形项的大小和位置
zoomoutbn = Button(bottomF, text=' 缩小 ',command=lambda : cv.scale(oval, 180, 50, 0.8, 0.8))
zoomoutbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)          # 缩小按钮，缩小椭圆
zoominbn = Button(bottomF, text=' 放大 ',command=lambda : cv.scale(oval, 180, 50, 1.2, 1.2))
zoominbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)           # 缩小按钮，缩小椭圆
clickbn = Button(bottomF, text=' 点选图形项 ',command=click_select)
clickbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)      # 点选图形项按钮，触发 click_select 函数
rectbn = Button(bottomF, text=' 框选图形项 ',command=rect_select)
rectbn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)       # 框选图形项按钮 , 触发 rect_select 函数
deletebn = Button(bottomF, text=' 删除 ',command=lambda : cv.delete(oval))
deletebn.pack(side=LEFT, ipadx=10, ipady=5, padx=3)   # 删除按钮，删除图形项
root.mainloop()
```

一个功能相对完善的绘图程序。用户可以绘制直线、矩形、椭圆、多边形，鼠标左键单击来选中图形，鼠标右键拖动来移动图形。

```
from tkinter import *; from tkinter import ttk; from tkinter import colorchooser; import threading
class App:
    def __init__(self, master):
        self.master = master
        self.width = IntVar(); self.width.set(1)          # 保存设置初始的边框宽度
        self.outline = 'black'; self.fill = None          # 保存设置初始的边框颜色和填充颜色
        self.prevx = self.prevy = -10                     # 记录拖动时前一个点的 x、y 坐标
        self.firstx = self.firsty = -10                   # 记录拖动开始的第一个点的 x、y 坐标
        self.mv_prevx = self.mv_prevy = -10               # 记录拖动右键来移动图形时前一个点的 x、y 坐标
        self.item_type = 0; self.points = []              # item_type 保存图形类型，self.points 保存多边形的点
        self.init_widgets();
        self.temp_item = None; self.temp_items = []
        self.choose_item = None                           # 初始化选中的图形项
    def init_widgets(self):           # 创建界面组件
        self.cv = Canvas(root, background='white'); self.cv.pack(fill=BOTH, expand=True)
        self.cv.bind('<B1-Motion>', self.drag_handler)                  # 为鼠标左键拖动事件绑定处理函数
        self.cv.bind('<ButtonRelease-1>', self.release_handler)         # 为鼠标左键释放事件绑定处理函数
        self.cv.bind('<Double-1>', self.double_handler)                 # 为鼠标左键双击事件绑定处理函数
        f = ttk.Frame(self.master); f.pack(fill=X)
        self.bns = []
        for i, lb in enumerate((' 直线 ', ' 矩形 ', ' 椭圆 ', ' 多边形 ', ' 铅笔 ')):          # 创建多个按钮用于绘制图形
            bn = Button(f, text=lb, command=lambda i=i: self.choose_type(i))
            bn.pack(side=LEFT, ipadx=8,ipady=5, padx=5); self.bns.append(bn)
        self.bns[self.item_type]['relief'] = SUNKEN               # 默认选中直线
        ttk.Button(f, text=' 边框颜色 ', command=self.choose_outline).pack(side=LEFT, ipadx=8,ipady=5, padx=5)
        ttk.Button(f, text=' 填充颜色 ', command=self.choose_fill).pack(side=LEFT, ipadx=8,ipady=5, padx=5)
        om = ttk.OptionMenu(f, self.width, '1', '0','1','2','3','4','5','6','7','8', command = None)
        om.pack(side=LEFT, ipadx=8,ipady=5, padx=5)
    def choose_type(self, i):
        for b in self.bns: b['relief'] = RAISED           # 将所有按钮恢复默认状态
        self.bns[i]['relief'] = SUNKEN                    # 将当前按钮设置选中样式
        self.item_type = i                                # 设置要绘制的图形
    def choose_outline(self):       # 处理选择边框颜色的方法
        select_color = colorchooser.askcolor(parent=self.master,title=" 请选择边框颜色 ", color=self.outline)
        if select_color is not None: self.outline = select_color[1]
    def choose_fill(self):          # 处理选择填充颜色的方法
        select_color = colorchooser.askcolor(parent=self.master, title=" 请选择填充颜色 ", color=self.fill)
        if select_color is not None: self.fill = select_color[1]
        else: self.fill = None
    def drag_handler(self, event):
        if self.item_type == 0:       # 如果是绘制直线
            # 如果第一个点不存在（self.firstx 和 self.firsty 都小于 0）
            if self.firstx < -1 and self.firsty < -1: self.firstx, self.firsty = event.x, event.y
            if self.temp_item is not None: self.cv.delete(self.temp_item)   # 删除上一次绘制的虚线图形
            self.temp_item = self.cv.create_line(self.firstx, self.firsty, event.x, event.y, dash=2) # 重绘虚线
        if self.item_type == 1 or self.item_type == 2:          # 如果是绘制矩形或椭圆
            # 如果第一个点不存在（self.firstx 和 self.firsty 都小于 0）
            if self.firstx < -1 and self.firsty < -1: self.firstx, self.firsty = event.x, event.y
            if self.temp_item is not None: self.cv.delete(self.temp_item)   # 删除上一次绘制的虚线图形
            leftx, lefty = min(self.firstx, event.x), min(self.firsty, event.y)
            rightx, righty = max(self.firstx, event.x), max(self.firsty, event.y)
            self.temp_item = self.cv.create_rectangle(leftx, lefty, rightx, righty, dash=2) # 重绘虚线选择框
        if self.item_type == 3:
            self.draw_polygon = True
            # 如果第一个点不存在（self.firstx 和 self.firsty 都小于 0）
            if self.firstx < -1 and self.firsty < -1: self.firstx, self.firsty = event.x, event.y
            if self.temp_item is not None: self.cv.delete(self.temp_item) # 删除上一次绘制的虚线图形
            self.temp_item = self.cv.create_line(self.firstx, self.firsty, event.x, event.y, dash=2) # 重绘虚线
        if self.item_type == 4:
            if self.prevx > 0 and self.prevy > 0:# 如果前一个点存在（self.prevx 和 self.prevy 都大于 0）
                self.cv.create_line(self.prevx, self.prevy, event.x, event.y, fill=self.outline, width=self.width.get())
            self.prevx, self.prevy = event.x, event.y
```

```
    def item_bind(self, t):
        self.cv.tag_bind(t, '<B3-Motion>', self.move)                    # 为鼠标右键拖动事件绑定处理函数
        self.cv.tag_bind(t, '<ButtonRelease-3>', self.move_end)          # 为鼠标右键释放事件绑定处理函数
    def release_handler(self, event):
        if self.temp_item is not None:          # 删除临时绘制的虚线图形项
            if self.item_type != 3: self.cv.delete(self.temp_item)       # 如果不是绘制多边形
            else: self.temp_items.append(self.temp_item)      #如果绘制多边形，保存之前的虚线以便后面删除
        self.temp_item = None
        if self.item_type == 0:         # 如果是绘制直线
            if self.firstx > 0 and self.firsty > 0: #如果第一个点存在（self.firstx 和 self.firsty 都大于 0）绘制实际的直线
                t = self.cv.create_line(self.firstx, self.firsty, event.x, event.y, fill=self.outline, width=self.width.get())
                self.cv.tag_bind(t, '<Button-1>',lambda event=event, t=t: self.choose_item_handler(event,t))
                self.item_bind(t)       # 为鼠标左键单击事件绑定处理函数，用于选择被单击的图形项
        if self.item_type == 1 or self.item_type == 2:          # 如果是绘制矩形或椭圆
            if self.firstx > 0 and self.firsty > 0:          # 如果第一个点存在（self.firstx 和 self.firsty 都大于 0）
                leftx, lefty = min(self.firstx, event.x), min(self.firsty, event.y)
                rightx, righty = max(self.firstx, event.x), max(self.firsty, event.y)
                if self.item_type == 1: # 绘制实际的矩形
                    t = self.cv.create_rectangle(leftx, lefty, rightx, righty,outline=self.outline, fill=self.fill, width=self.width.get())
                if self.item_type == 2: # 绘制实际的椭圆
                    t = self.cv.create_oval(leftx, lefty, rightx, righty, outline=self.outline, fill=self.fill, width=self.width.get())
                self.cv.tag_bind(t, '<Button-1>', lambda event=event, t=t: self.choose_item_handler(event,t))
                self.item_bind(t)       # 为鼠标左键单击事件绑定处理函数，用于选择被单击的图形项
        if self.item_type != 3:
            self.prevx = self.prevy = -10;  self.firstx = self.firsty = -10
        elif(self.draw_polygon):      # 如果正在绘制多边形
            self.points.append((self.firstx, self.firsty))          # 将第一个点添加到列表中
            self.firstx, self.firsty = event.x, event.y
    def double_handler(self, event):
        if self.item_type == 3:         # 只处理绘制多边形的情形
            t = self.cv.create_polygon(*self.points, outline=self.outline, fill="" if self.fill is None else self.fill,
                width=self.width.get())
            self.cv.tag_bind(t, '<Button-1>', lambda event=event, t=t: self.choose_item_handler(event,t))
            self.item_bind(t) # 为鼠标左键单击事件绑定处理函数，用于选择被单击的图形项
            self.points.clear()         # 清空所有保存的点数据
            self.firstx = self.firsty = -10         # 将 self.firstx = self.firsty 设置为 -10，停止绘制
            for it in self.temp_items: self.cv.delete(it) # 删除所有临时的虚线框
            self.temp_items.clear()
            self.draw_polygon = False
    def choose_item_handler(self, event, t):         # 根据传入的参数 t 来选中对应的图形项
        self.choose_item = t          # 使用 self.choose_item 保存当前选中项
    def move(self, event):          # 定义移动图形项的方法
        if self.choose_item is not None:      # 如果被选中图形项不为空，才可以执行移动
            if self.mv_prevx > 0 and self.mv_prevy > 0:
                # 如果前一个点存在（self.mv_prevx 和 self.mv_prevy 都大于 0），移动选中的图形项
                self.cv.move(self.choose_item, event.x - self.mv_prevx, event.y - self.mv_prevy)
            self.mv_prevx, self.mv_prevy = event.x, event.y
    def move_end(self, event):    # 结束移动的方法
        self.mv_prevx = self.mv_prevy = -10
    def delete_item(self, event):
        # 如果被选中的 item 不为空，删除被选中的图形项
        if self.choose_item is not None: self.cv.delete(self.choose_item)
root = Tk()
root.title(" 绘图工具 ")
root.iconbitmap('images/python.ico')
root.geometry('800x680')
app = App(root)
root.bind('<Delete>', app.delete_item)
root.mainloop()
```

- 绘制动画

```
from tkinter import *
from tkinter import messagebox
import threading
import random
GAME_WIDTH = 500; GAME_HEIGHT = 680
BOARD_X = 230; BOARD_Y = 600; BOARD_WIDTH = 80
BALL_RADIUS = 9
class App:
    def __init__(self, master):
        self.master = master
        self.ball_index = 0             # 记录小球动画的第几帧
        self.is_lose = False            # 记录游戏是否失败的旗标
        self.curx = 260; self.cury = 30; self.boardx = BOARD_X      # 初始化记录小球位置的变量
        self.init_widgets()
        self.vx = random.randint(3, 6); self.vy = random.randint(5, 10)         # x、y 方向的速度
        self.t = threading.Timer(0.1, self.moveball)   # 通过定时器指定 0.1 秒之后执行 moveball 函数
        self.t.start()
    def init_widgets(self): # 创建界面组件
        self.cv = Canvas(root, background='white', width=GAME_WIDTH, height=GAME_HEIGHT)
        self.cv.pack()
        self.cv.focus_set() # 让画布得到焦点，从而可以响应按键事件
        self.cv.bms = []
        for i in range(8): self.cv.bms.append(PhotoImage(file='images/ball_' + str(i+1) + '.gif')) # 初始化小球的动画帧
        self.ball = self.cv.create_image(self.curx, self.cury, image=self.cv.bms[self.ball_index]) # 绘制小球
        self.board = self.cv.create_rectangle(BOARD_X, BOARD_Y,
            BOARD_X + BOARD_WIDTH, BOARD_Y + 20, width=0, fill='lightblue')
        self.cv.bind('<Left>', self.move_left)          # 为向左箭头按键绑定事件，挡板左移
        self.cv.bind('<Right>', self.move_right)        # 为向右箭头按键绑定事件，挡板右移
    def move_left(self, event):
        if self.boardx <= 0: return
        self.boardx -= 5
        self.cv.coords(self.board, self.boardx, BOARD_Y, self.boardx + BOARD_WIDTH, BOARD_Y + 20)
    def move_right(self, event):
        if self.boardx + BOARD_WIDTH >= GAME_WIDTH: return
        self.boardx += 5
        self.cv.coords(self.board, self.boardx, BOARD_Y, self.boardx + BOARD_WIDTH, BOARD_Y + 20)
    def moveball(self):
        self.curx += self.vx; self.cury += self.vy
        if self.curx + BALL_RADIUS >= GAME_WIDTH: self.vx = -self.vx    # 小球到了右边墙壁，转向
        if self.curx - BALL_RADIUS <= 0: self.vx = -self.vx             # 小球到了左边墙壁，转向
        if self.cury - BALL_RADIUS <= 0: self.vy = -self.vy             # 小球到了上边墙壁，转向
        if self.cury + BALL_RADIUS >= BOARD_Y:                          # 小球到了挡板处
            if self.boardx <= self.curx <= (self.boardx + BOARD_WIDTH):     # 如果在挡板范围内
                self.vy = -self.vy
            else:
                messagebox.showinfo(title=' 失败 ', message=' 您已经输了 ')
                self.is_lose = True
        self.cv.coords(self.ball, self.curx, self.cury)
        self.ball_index += 1
        self.cv.itemconfig(self.ball, image=self.cv.bms[self.ball_index % 8])
        if not self.is_lose:  # 如果游戏还未失败，让定时器继续执行
            self.t = threading.Timer(0.1, self.moveball)   # 通过定时器指定 0.1 秒之后执行 moveball 函数
            self.t.start()
root = Tk(); root.title(" 弹球游戏 ")
root.geometry('%dx%d' % (GAME_WIDTH, GAME_HEIGHT))
root.resizable(width=False, height=False)          # 禁止改变窗口大小
App(root)
root.mainloop()
```

Python 文件操作（I/O）

Python 提供了丰富的文件 I/O 支持，既提供了 pathlib 和 os.path 来操作各种路径，也提供了全局的 open() 函数来打开文件（在打开文件之后，既可读取文件的内容，也可向文件输出内容）。

os 模块包含了大量进行文件 I/O 的函数来读取、写入文件，可以根据需要选择不同的方式来读写文件。

tempfile 模块来创建临时文件和临时目录，tempfile 模块的高级 API 会自动管理临时文件的创建和删除；当程序不再使用临时文件和临时目录时，程序会自动删除临时文件和临时目录。

文件的操作可大致分为以下 2 类：

- **删除、修改权限**：作用于文件本身，属于系统级操作。系统级操作功能单一，比较容易实现，可以借助 Python 中的转用模块（os、sys 等），并调用模块中的指定函数来实现。
- **写入、读取**：是文件最常用的操作，作用于文件的内容，属于应用级操作。应用级操作可以分为 3 步：
  1. **打开文件**：使用 open() 函数，该函数会返回一个文件对象；
  2. **对已打开文件做读 / 写操作**：读取文件内容可使用 read()、readline() 以及 readlines() 函数；向文件中写入内容，可以使用 write() 函数。
  3. **关闭文件**：完成对文件的读 / 写操作之后，最后需要关闭文件，可以使用 close() 函数。

打开指定文件（open() 函数）

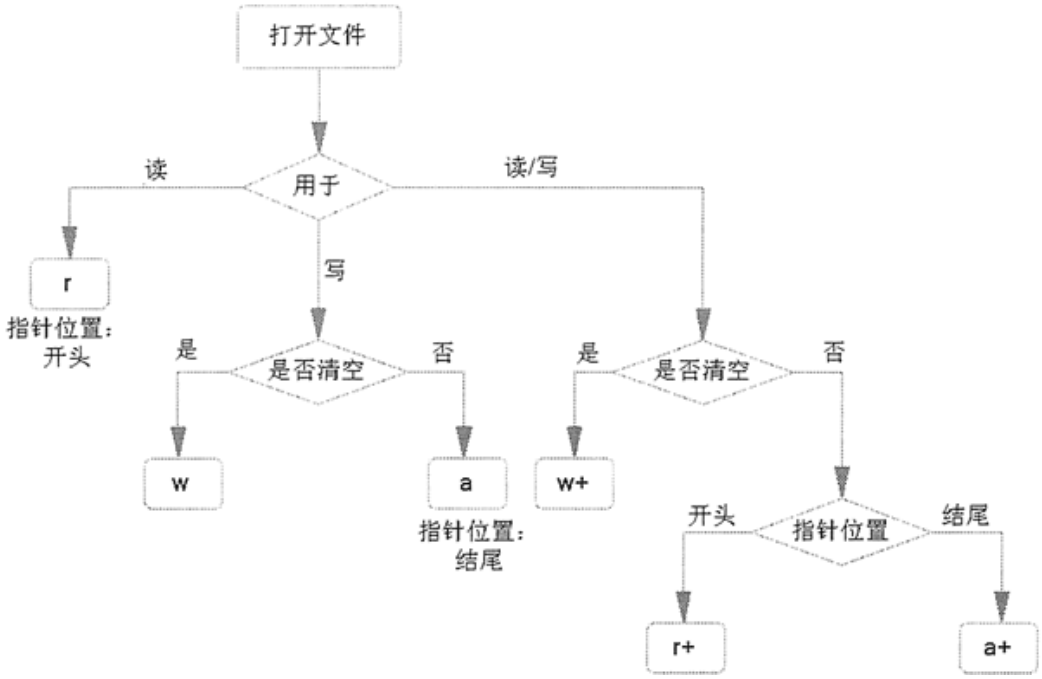
如果想要操作文件，首先需要创建或者打开指定的文件，并创建一个文件对象，open() 函数用于创建或打开指定文件，该函数的语法格式如下：

```
file = open(file_name [, mode[, buffering]])
```

- file：表示要创建的文件对象。
- file_mame：要创建或打开文件的文件名称（要用单引号或双引号括起来）。如果文件和当前执行的代码文件位于同一目录，则直接写文件名即可；否则，此参数需要指定打开文件所在的完整路径。
- mode：可选参数，用于指定文件的打开模式（默认为只读）。文件打开模式，直接决定了后续可以对文件做哪些操作。可选的打开模式下表 1 。
- buffing：可选参数，用于指定对文件做读写操作时，是否使用缓冲区。

模式	意义	注意事项
r	只读模式打开文件，读文件内容的指针会放在文件的开头。	操作的文件必须存在。
rb	以二进制格式、采用只读模式打开文件，读文件内容的指针位于文件的开头，一般用于非文本文件，如图片文件、音频文件等。	
r+	打开文件后，既可以从头读取文件内容，也可以从开头向文件中写入新的内容，写入的新内容会覆盖文件中等长度的原有内容。	
rb+	以二进制格式、采用读写模式打开文件，读写文件的指针会放在文件的开头，通常针对非文本文件（如音频文件）。	
w	以只写模式打开文件，若该文件存在，打开时会清空文件中原有的内容。	若文件存在，会清空其原有内容（覆盖文件）；反之，则创建新文件。
wb	以二进制格式、只写模式打开文件，一般用于非文本文件（如音频文件）	
w+	打开文件后，会对原有内容进行清空，并对该文件有读写权限。	
wb+	以二进制格式、读写模式打开文件，一般用于非文本文件	
a	以追加模式打开一个文件，对文件只有写入权限，如果文件已经存在，文件指针将放在文件的末尾（即新写入内容会位于已有内容之后）；反之，则会创建新文件。	
ab	以二进制格式打开文件，并采用追加模式，对文件只有写权限。如果该文件已存在，文件指针位于文件末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	
a+	以读写模式打开文件；如果文件存在，文件指针放在文件的末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	
ab+	以二进制模式打开文件，并采用追加模式，对文件具有读写权限，如果文件存在，则文件指针位于文件的末尾（新写入文件会位于已有内容之后）；反之，则创建新文件。	

将以上几个容易混淆的文件打开模式的功能，做了很好的对比：



成功打开文件之后，可以调用文件对象本身拥有的属性获取当前文件的部分信息，如：file.closed（判断文件是否已经关闭），file.mode（返回被打开文件的访问模式），file.name（返回文件的名称）。

open() 是否需要缓冲区

一般建议打开缓冲。在打开缓冲之后，当程序执行输出时，程序会先将数据输出到缓冲区中，而不用等待外设同步输出，当程序把所有数据都输出到缓冲区中之后，程序就可以去干其他事情了，留着缓冲区慢慢同步到外设即可；反过来，当程序执行输入时，程序会先等外设将数据读入缓冲区中，而不用等待外设同步输入。

使用 open() 函数时，如果其第三个参数是 0（或 False），那么该函数打开的文件就是不带缓冲的；如果其第三个参数是 1（或 True），则该函数打开的文件就是带缓冲的，此时程序执行 I/O 将具有更好的性能。如果其第三个参数是**大于 1 的整数**，则该整数用于指定缓冲区的大小（单位是字节）；如果其第三个参数为**任何负数**，则代表使用默认的缓冲区大小。

```
f = open('open_test.py')          # 以默认方式打开文件
print(f.encoding)                 # 访问文件的编码方式：cp936
print(f.mode)                     # 访问文件的访问模式：r
print(f.closed)                   # 访问文件是否已经关闭：False
print(f.name)                     # 访问文件对象打开的文件名：open_test.py
```

文本文件和二进制文件有什么区别

从数据存储的角度上分析，实际上二进制文件和文本文件没有区别，内容都是以二进制形式保存在磁盘中。

文本文件通常保存的是肉眼可见的字符，采用的是 ASCII、UTF-8、GBK 等字符编码，文本编辑器可以识别出这些编码格式，并将编码值转换成字符展示出来。所以，用文本编辑器打开，能够顺利看懂文件的内容。

二进制文件通常用来保存视频、图片、音频等不可阅读的内容，文本编辑器无法识别这些文件的编码格式，只能按照字符编码格式胡乱解析，所以，用文本编辑器打开，看到的是一堆乱码，根本看不懂。

使用 open() 函数以文本格式打开文件和以二进制格式打开文件，唯一的区别是对文件中换行符的处理不同。

在 Windows 系统中，文件中用 "\r\n" 作为行末标识符（即换行符），当以文本格式读取文件时，会将 "\r\n" 转换成 "\n"；反之，以文本格式将数据写入文件时，会将 "\n" 转换成 "\r\n"。这种隐式转换换行符的行为，对用文本格式打开文本文件是没有问题的，但如果用文本格式打开二进制文件，就有可能改变文本中的数据（将 \r\n 隐式转换为 \n）。而**在 Unix/Linux 系统中**，默认的文件换行符就是 \n，因此在 Unix/Linux 系统中文本格式和二进制格式并无本质的区别。

总的来说，为了保险起见，对于 Windows 平台最好用 b 打开二进制文件；对于 Unix/Linux 平台，打开二进制文件，可以用 b，也可以不用。

按字节（字符）读取文件（read()）

read() 函数可以按字节或字符读取文件内容，使用 open() 函数打开文件时，如果使用了 b 模式，则每次读取一个字节；反之，则每次读取一个字符。read() 函数的基本语法格式如下：

file.read([size])

- file：打开的文件对象；
- size：可选参数，用于指定要读取的字符个数（默认一次性读取所有内容）。

```
f = open("a.txt", 'r', True)          # 只读方式发开 a.txt
while True:
    ch = f.read(1)            # 每次读取一个字符
    if not ch:  break         # 如果没有读到数据，跳出循环
    print(ch, end='')         # 输出 ch
print(f.read())    # 直接读取全部文件
f.close()          # 关闭文件
```

当使用 open() 函数打开文本文件时，默认会使用当前操作系统的字符集（比如 Windows 平台，默认使用 GBK 字符集），若 open() 函数使用的字符集与文件的字符集不匹配，read() 函数抛出 UnicodeDecodeError 异常。有两种解决方式：

- 使用**二进制模式**读取，然后用 bytes 的 decode() 方法恢复成字符串。

```
f = open("a.txt", 'rb', True)          # 指定使用二进制方式读取文件内容，a.txt 以 utf-8 编码存储
print(f.read().decode('utf-8'))     # 读取全部文件，并调用 bytes 的 decode 将字节内容恢复成字符串
f.close()          # 关闭文件
```

- 利用 codecs **模块的** open() **函数**来打开文件，该函数在打开文件时允许指定字符集。

```
import codecs
f = codecs.open("a.txt", 'r', 'utf-8', buffering=True)          # 指定使用 utf-8 字符集读取文件内容
print(f.read())    # 读取全部文件
f.close()          # 关闭文件
```

按行读取文件（readline() 和 readlines()）

文件对象提供了 readline() 和 readlines() 两个函数来逐行读取文件

- readline() **函数**用于读取一行，包含最后的换行符“\n”。基本语法格式为：

  file.readline([size])

  - file：打开的文件对象；
  - size：可选参数，指定读取每一行时，一次最多读取的字符数（默认为整行）。

```
f = open("a.txt", 'r', True)
while True:
    line = f.readline()       # 每次读取一行
    if not line: break        # 如果没有读到数据，跳出循环
    print(line)               # 输出 line
f.close()          # 关闭文件
```

- readlines() **函数**用于读取文件内的所有行。基本语法格式为：

  file.readlines()

```
f = open("a.txt", 'r', True)
lines = f.readlines()
for line in lines:  print(line)
f.close()          # 关闭文件
```

向文件中写入数据（write() 和 writelines()）

- write() 函数可以向文件中写入指定内容。语法格式如下：

file.write(string)

  - file：已经打开的文件对象（如打开模式错误，会抛出 io.UnsupportedOperation 错误）；
  - string：要写入文件的字符串（或字节串，仅适用写入二进制文件中）。

```
f = open("a.txt", 'w')                  # 可以以 r+、w、w+、a 或 a+ 的模式打开文件
f.write(" 写入一行新数据 ")              # w 打开会先清空原文件中的内容； a 打开保留原文件中的内容
f.close()          # 写入文件完成后，一定要用 close() 关闭文件，否则写入的内容不会保存到文件中。
```

- writelines() 函数可以实现将字符串列表写入文件中。

```
f = open('a.txt', 'r')
n = open('b.txt','w+')
n.writelines(f.readlines())
n.close()
f.close()
```

使用 writelines() 函数向文件中写入多行数据时，不会自动给各行添加换行符。

关闭文件（close()）

close() 函数用来关闭使用 open() 函数打开的文件。语法格式如下：

file.close()

文件在打开并操作完成之后，应及时关闭，否则会给程序带来很多无法预知的错误。

文件指针操作（seek() 和 tell()）

文件指针用于标明文件读写的起始位置。使用 open() 函数打开文件并读取文件中的内容时，文件指针总是位于文件开始位置。通过移动文件指针的位置，再借助 read() 和 write() 函数，就可以轻松实现，读取文件中指定位置的数据（或者向文件中的指定位置写入数据）。

tell() 函数用于判断文件指针当前所处的位置。语法格式如下：

file.tell()

```
f = open("a.txt",'r')
print(f.tell())      # 0
print(f.read(3))     # htt
print(f.tell())      # 3
```

seek() 函数用于移动文件指针到文件的指定位置。语法格式如下：

file.seek(offset[, whence])

- file：表示文件对象；
- whence：可选参数，文件指针要放置的位置，0（文件头，默认值）、1（当前位置）、2（文件尾）。
- offset：相对于 whence 位置文件指针的偏移量，正数表示向后偏移，负数表示向前偏移。

```
f = open("a.txt",'rb')       # 文件是 26 个字母（后跟 \r、\n 和文件结束符），共 29 个字符
print(f.tell())      # 0
f.seek(5)            # 从文件开头，向后移动到 5 个字符的位置
print(f.tell())      # 5
f.seek(5, 1)         # 从当前位置，向后移动到 5 个字符的位置
print(f.tell())      # 10
f.seek(–3, 2)        # 从文件结尾，向前移动到 1 个字符的位置
print(f.tell())      # 25
```

fileinput 模块（逐行读取多个文件）

fileinput 模块提供了 input 函数，可以把多个输入流合并在一起，该函数的语法格式如下：

```
fileinput.input（files="filename1, filename2, ...",
                inplace=False, backup='', bufsize=0, mode='r', openhook=None）
```

此函数会返回一个 FileInput 对象，其中，各个参数的含义如下：

- files：多个文件的路径列表；
- inplace：用于指定是否将标准输出的结果写回到文件（默认为 False）。
- backup：用于指定备份文件的扩展名；
- bufsize：指定缓冲区的大小（默认为 0）；
- mode：打开文件的格式（默认为 r：只读格式）；
- openhook：控制文件的打开方式（例如编码格式等）。

使用 input 函数创建了 FileInput 对象之后，即可通过 for 循环来遍历文件的每一行。此外，fileinput 还提供了很多全局函数来判断正在读取的文件信息，如表所示：

函数	描述
fileinput.filename()	返回正在读取的文件的文件名。
fileinput.fileno()	返回当前文件的文件描述符（file descriptor），该文件描述符是一个整数。
fileinput.lineno()	返回当前读取的行号。
fileinput.filelineno()	返回当前读取的行在其文件中的行号。
fileinput.isfirstline()	返回当前读取的行在其文件中是否为第一行。
fileinput.isstdin()	返回最后一行是否从 sys.stdin 读取。程序可以使用“-”代表从 sys.stdin 读取。
fileinput.nextfile()	关闭当前文件，开始读取下一个文件。
fileinput.close()	关闭 FileInput 对象。

```
import fileinput
for line in fileinput.input(files=('a.txt', 'b.txt')):       # 一次读取多个文件
    print(fileinput.filename(), fileinput.filelineno(), line)      # 输出文件名，当前行在当前文件中的行号
fileinput.close()                                            # 关闭文件流
```

linecache 模块（随机读取文件指定行）

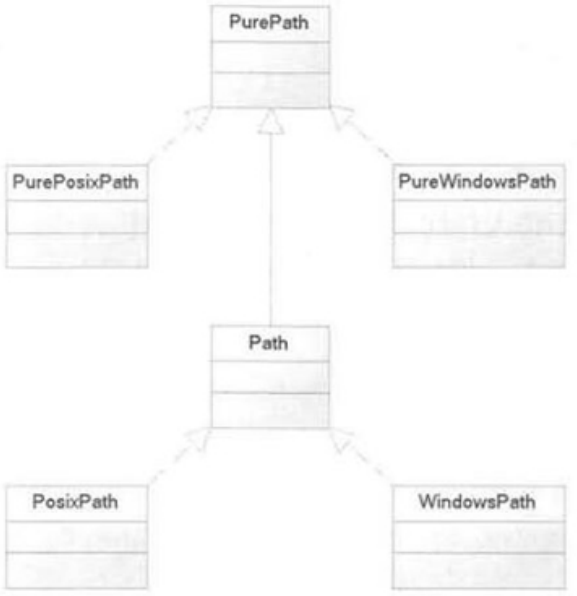
linecache 模块允许从 Python 源文件中随机读取指定行，并在内部使用缓存优化存储。由于该模块主要被设计成读取 Python 源文件，因此它会用 UTF-8 字符集来读取文本文件。实际上，使用 linecache 模块也可以读取其他文件，只要该文件使用了 UTF-8 字符集存储。linecache 模块包含以下常用函数：

- linecache.getline(filename, lineno, module_globals=None)：读取指定模块中指定文件的指定行。其中 filename 指定文件名，lineno 指定行号。
- linecache.clearcache()：清空缓存。
- linecache.checkcache(filename=None)：检查缓存是否有效。如果没有指定 filename 参数，则默认检查所有缓存的数据。

```
import linecache
import random
print(linecache.getline(random.__file__, 3))        # 读取 random 模块的源文件的第 3 行
print(linecache.getline('linecache_test.py', 3))    # 读取本程序的第 3 行
print(linecache.getline('utf_text.txt', 2))         # 读取普通文件的第 2 行
```

pathlib 模块

pathlib 模块提供了一组面向对象的类，这些类可代表各种操作系统上的路径，程序可通过这些类操作路径。pathlib 模块下的类如图所示。



PurePath 代表并不访问实际文件系统的“纯路径”。只是负责对路径字符串执行操作，至于该字符串是否对应实际的路径，它并不关心。PurePath 有两个子类：PurePosixPath（UNIX 或 Mac OS X 风格的路径）和 PureWindowsPath（Windows 风格的路径）。

> UNIX 风格的路径和 Windows 风格路径的主要区别在于根路径和路径分隔符，UNIX 风格路径的根路径是斜杠（/），而 Windows 风格路径的根路径是盘符（c:）；UNIX 风格的路径的分隔符是斜杠（/），而 Windows 风格路径的分隔符是反斜杠（\）。

Path 代表访问实际文件系统的“真正路径”。可用于判断对应的文件是否存在、是否为文件、是否为目录等。Path 同样有两个子类，即 PosixPath 和 WindowsPath。

- PurePath 类

如果在 **UNIX 或 Mac OS X 系统**上使用 PurePath 创建对象，实际返回 **PurePosixPath 对象**；

如果在 **Windows 系统**上使用 PurePath 创建对象，实际返回 **PureWindowsPath 对象**。

在创建 PurePath 对象时，可**不传入参数**（默认为当前路径），也可传入**单个路径字符串**或**多个路径字符串**（将会拼接成一个字符串，若包含多个根路径时，只有最后一个根路径有效）。

PurePath 对象支持**比较运算符**，可进行相等比较和大小比较。

PurePath 对象支持**斜杠运算符（/）**，用于将多个路径连接起来。

PurePath 对象支持 **str() 函数**，将 PurePath 对象恢复成字符串对象。

PurePath 类属性和方法	功能描述
PurePath.parts	返回路径字符串中所包含的各部分。
PurePath.drive	返回路径字符串中的驱动器盘符。
PurePath.root	返回路径字符串中的根路径。
PurePath.anchor	返回路径字符串中的盘符和根路径。
PurePath.parents	返回当前路径的全部父路径。
PurPath.parent	返回当前路径的上一级路径，相当于 parents[0] 的返回值。
PurePath.name	返回当前路径中的文件名。
PurePath.suffixes	返回当前路径中的文件所有后缀名。
PurePath.suffix	返回当前路径中的文件后缀名。相当于 suffixes 返回列表的最后一个元素。
PurePath.stem	返回当前路径中的主文件名。
PurePath.as_posix()	将当前路径转换成 UNIX 风格的路径。
PurePath.as_uri()	将当前路径转换成 URI。只有绝对路径才能转换，否则将会引发 ValueError。
PurePath.is_absolute()	判断当前路径是否为绝对路径。
PurePath.joinpath(*other)	将多个路径连接在一起，作用类似于前面介绍的斜杠运算符。
PurePath.match(pattern)	判断当前路径是否匹配指定通配符。
PurePath.relative_to(*other)	获取当前路径中去除基准路径之后的结果。
PurePath.with_name(name)	将当前路径中的文件名替换成新文件名。若没有文件名，则会引发 ValueError。
PurePath.with_suffix(suffix)	将当前路径中的文件后缀名替换成新的后缀名。若没有后缀名，则添加。

```
from pathlib import *
# 访问 drive 属性（驱动器盘符）
print(PureWindowsPath('c:/Program Files/').drive) # c:
print(PureWindowsPath('/Program Files/').drive) # ''
print(PurePosixPath('/etc').drive) # ''
# 访问 root 属性（根路径）
print(PureWindowsPath('c:/Program Files/').root) # \
print(PureWindowsPath('c:Program Files/').root) # ''
print(PurePosixPath('/etc').root) # /
# 访问 anchor 属性（盘符和根路径）
print(PureWindowsPath('c:/Program Files/').anchor) # c:\
print(PureWindowsPath('c:Program Files/').anchor) # c:
print(PurePosixPath('/etc').anchor) # /
pp0 = PurePath('c:/path1/path2/path3/filename.ext1.ext2.ext3')   # 创建绝对路径
pp1 = PurePath('path1','path2','path3','filename.ext1.ext2.ext3')   # 创建相对路径
print(' 绝对路径：',pp0)                    # 绝对路径： c:\path1\path2\path3\filename.ext1.ext2.ext3
print('Unix 风格：',pp0.as_posix())         #Unix 风格： c:/path1/path2/path3/filename.ext1.ext2.ext3
print(' 相对路径：',pp1)                    # 相对路径： path1\path2\path3\filename.ext1.ext2.ext3
print('Unix 风格：',pp1.as_posix())         #Unix 风格： path1/path2/path3/filename.ext1.ext2.ext3
print(pp0.parent)          # 访问 parent 属性：c:\path1\path2\path3
print(pp0.name)            # 访问 name 属性： filename.ext1.ext2.ext3
print(pp0.suffix)          # 访问 suffix 属性：.ext3
print(pp0.stem)            # 访问 suffix 属性：filename.ext1.ext2
# 访问 parents 属性
print(pp0.parents[0])      # c:\path1\path2\path3
print(pp0.parents[1])      # c:\path1\path2
print(pp0.parents[2])      # c:\path1
print(pp0.parents[3])      # c:\
# 访问 suffixes 属性
print(pp0.suffixes[0])     # .ext1
print(pp0.suffixes[1])     # .ext2
print(pp0.suffixes[2])     # .ext3
# 转换成 uri
print(pp0.as_uri())        # 绝对路径转换正确：file:///c:/path1/path2/path3/filename.ext1.ext2.ext3
#print(pp1.as_uri())       # 相对路径引发异常：ValueError
print(PurePath('a/b.py').match('*.py'))              # 判断路径是否匹配指定模式：True
print(PurePath('/a/b/c.py').match('b/*.py'))         # 判断路径是否匹配指定模式：True
print(PurePath('/a/b/c.py').match('a/*.py'))         # 判断路径是否匹配指定模式：False
# relative_to 方法（去除基准路径）
print(pp0.relative_to('c:/'))                  # path1\path2\path3\filename.ext1.ext2.ext3
print(pp0.relative_to('c:/path1'))             # path2\path3\filename.ext1.ext2.ext3
print(pp0.relative_to('c:/path1/path2'))       # path3\filename.ext1.ext2.ext3
# 文件名替换
print(pp0.with_name('filename.py'))                  # 替换文件名：c:\path1\path2\path3\filename.py
#print(PureWindowsPath('c:/').with_name('filename.py')) # 无文件名时报错：ValueError
print(PurePath('readme.txt').with_suffix('.py'))     # 替换后缀名：readme.py
print(PurePath('readme').with_suffix('.txt'))        # 替换后缀名：readme.txt
```

- Path 类

Path 是 PurePath 的子类，支持 PurePath 的各种操作、属性和方法，还可真正访问底层的文件系统，包括判断 Path 对应的路径是否存在，获取 Path 对应路径的各种属性（如是否只读、是文件还是文件夹等），甚至可以对文件进行读写。

Path 也提供了两个子类，PosixPath（UNIX 风格的路径）和 WindowsPath（Windows 风格的路径）。

Path 类属性和方法	功能描述
Path.is_xxx()	判断该 Path 对应的路径是否为 xxx。
Path.exists()	判断该 Path 对应的目录是否存在。
Path.iterdir()	返回 Path 对应目录下的所有子目录和文件。
Path.glob()	获取 Path 对应目录及其子目录下匹配指定模式的所有文件。
Path.read_bytes()	读取该 Path 对应文件的字节数据（二进制数据）。
Path.read_text(encoding=None, errors=None)	读取该 Path 对应文件的文本数据。
Path.write_bytes(data)	输出字节数据（二进制数据）。
Path.write_text(data, encoding=None, errors=None)	输出文本数据。

```
from pathlib import *
p = Path('.')                    # 获取当前目录
for x in p.iterdir(): print(x)   # 遍历当前目录下所有文件和子目录
p = Path('../')                  # 获取上一级目录
for x in p.glob('**/*.py'): print(x)  # 获取目录及其所有子目录下的 .py 文件
p = Path('e:\python')            # 获取指定目录（e:\python）
for x in p.glob('**/a.py'):print(x)  # 获取目录及其所有子目录下的 a.py 文件
```

```
from pathlib import *
p = Path('path_test.txt')
result = p.write_text(''' 有一个美丽的新世界
它在远方等我
那里有天真的孩子
还有姑娘的酒窝 ''', encoding='GBK')     # 以 GBK 字符集输出文本内容
print(' 输出字符数：',result)             # 输出字符数：33
content = p.read_text(encoding='GBK')  # 指定以 GBK 字符集读取文本内容
print(content)                         # 输出读取的文本内容
bb = p.read_bytes()                    # 读取字节内容
print(bb)
```

os.path 模块

os.path 模块提供了一些目录的操作方法。如：exists（判断该目录是否存在）、getctime（获取目录创建时间）、getmtime（获取目录最后一次修改时间）、getatime（获取目录最后一次访问时间）、getsize（获取文件的大小）。

```
import os
import time
print(os.path.abspath("abc.txt"))          # 获取绝对路径：G:\publish\codes\12\12.2\abc.txt
print(os.path.commonprefix(['/usr/lib', '/usr/local/lib']))    # 获取共同前缀：/usr/l
print(os.path.commonpath(['/usr/lib', '/usr/local/lib']))      # 获取共同路径：\usr
print(os.path.dirname('abc/xyz/README.txt'))                   # 获取目录：abc/xyz
print(os.path.exists('abc/xyz/README.txt'))                    # 判断指定目录是否存在：False
print(time.ctime(os.path.getatime('os.path_test.py')))         # 获取最近一次访问时间
print(time.ctime(os.path.getmtime('os.path_test.py')))         # 获取最后一次修改时间
print(time.ctime(os.path.getctime('os.path_test.py')))         # 获取创建时间
print(os.path.getsize('os.path_test.py'))                      # 获取文件大小
print(os.path.isfile('os.path_test.py'))                       # 判断是否为文件：True
print(os.path.isdir('os.path_test.py'))                        # 判断是否为目录：False
print(os.path.samefile('os.path_test.py', './os.path_test.py')) # 判断是否为同一个文件：True
```

fnmatch 模块

fnmatch 模块于用文件名的匹配，匹配支持如下通配符：

- *：可匹配任意个任意字符。
- ？：可匹配一个任意字符。
- [字符序列]：可匹配中括号里字符序列中的任意字符，支持中画线（如[a-c]表示 a、b 和 c 中任意一个）。
- [! 字符序列]：可匹配不在中括号里字符序列中的任意字符。

提供了如下函数：

- fnmatch.fnmatch(filename, pattern)：判断文件名是否匹配指定 pattern（不区分大小写）。

```
from pathlib import *
import fnmatch
for file in Path('.').iterdir():          # 遍历当前目录下所有文件和子目录
    if fnmatch.fnmatch(file, '*_test.PY'):          # 访问所有以 _test.py 结尾的文件
        print(file)
```

- fnmatch.fnmatchcase(filename, pattern)：判断文件名是否匹配指定 pattern（区分大小写）。
- fnmatch.filter(names, pattern)：对 names 列表进行过滤，返回 names 列表中匹配 pattern 的文件名。

```
names = ['a.py', 'b.py', 'c.py', 'd.py']
sub = fnmatch.filter(names, '[ac].py')      # 对 names 列表进行过滤
print(sub)                                  # ['a.py', 'c.py']
```

- fnmatch.translate(pattern)：将一个 UNIX shell 风格的 pattern 转换为正则表达式 pattern。

```
print(fnmatch.translate('?.py'))            # (?s:.\.py)\Z
print(fnmatch.translate('[ac].py'))         # (?s:[ac]\.py)\Z
print(fnmatch.translate('[a-c].py'))        # (?s:[a-c]\.py)\Z
```

tempfile 模块

tempfile 模块专门用于创建临时文件和临时目录，可以在 UNIX 平台和 Windows 平台上运行良好。

模块函数	功能描述
tempfile.TemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)	创建临时文件。该函数返回一个类文件对象，也就是支持文件 I/O。
tempfile.NamedTemporaryFile(mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None, delete=True)	创建临时文件。该函数的功能与上一个函数的功能大致相同，只是它生成的临时文件在文件系统中有文件名。
tempfile.SpooledTemporaryFile(max_size=0, mode='w+b', buffering=None, encoding=None, newline=None, suffix=None, prefix=None, dir=None)	创建临时文件。与 TemporaryFile 函数相比，当程序向该临时文件输出数据时，会先输出到内存中，直到超过 max_size 才会真正输出到物理磁盘中。
tempfile.TemporaryDirectory(suffix=None, prefix=None, dir=None)	生成临时目录。
tempfile.gettempdir()	获取系统的临时目录。
tempfile.gettempdirb()	同 gettempdir()，返回字节串。
tempfile.gettempprefix()	返回用于生成临时文件的前缀名。
tempfile.gettempprefixb(	同 gettempprefix()，返回字节串。

```
import tempfile
# 第一种方式：手动创建临时文件，关闭该临时文件时，该文件会被自动删除
fp = tempfile.TemporaryFile()
print(fp.name)
fp.write(' 两情若是久长时，'.encode('utf-8'))
fp.write(' 又岂在朝朝暮暮。'.encode('utf-8'))
fp.seek(0)                          # 将文件指针移到开始处，准备读取文件
print(fp.read().decode('utf-8'))    # 输出刚才写入的内容
fp.close()                          # 关闭文件，该文件将会被自动删除
# 第二种方式使用 with 语句创建临时文件，with 会自动关闭临时文件
with tempfile.TemporaryFile() as fp:
    fp.write(b'I Love Python!')     # 写入内容
    fp.seek(0)                      # 将文件指针移到开始处，准备读取文件
    print(fp.read())                # 读取文件内容：b'I Love Python!'
with tempfile.TemporaryDirectory() as tmpdirname:       # 通过 with 语句创建临时目录
    print(' 创建临时目录 ', tmpdirname)
```

os 模块

os 模块提供了大量操作文件和目录的函数

	函数	描述
与目录相关的函数	os.getcwd()	获取当前目录。
	os.chdir(path)：	改变当前目录。
	os.fchdir(fd)	通过文件描述改变当前目录。
	os.chroot(path)	改变当前进程的根目录。
	os.listdir(path)	返回 path 对应目录下的所有文件和子目录。
	os.mkdir(path[, mode])	创建 path 对应的目录。mode 指定目录的权限。
	os.makedirs(path[, mode])	递归创建目录。
	os.rmdir(path)	删除 path 对应的空目录。
	os.removedirs(path)	递归删除目录。
	os.rename(src, dst)	重命名文件或目录，将 src 重名为 dst。
	os.renames(old, new)	对文件或目录进行递归重命名。
与权限相关的函数	os.access(path, mode)	检查 path 对应的文件或目录是否具有指定权限。mode 可选以下一个或多个值：os.F_OK（是否存在）、os.R_OK（是否可读）、os.W_OK（是否可写）、os.X_OK（是否可执行）。
	os.chrnod(path, mode)	更改权限。mode 代表要改变的权限，可选以下一个或多个值：stat.S_IXOTH（其他用户有执行权限）、stat.S_IWOTH（其他用户有写权限）、stat.S_TROTH（其他用户有读权限）、stat.S_IRWXO（其他用户有全部权限）、stat.S_IXGRP（组用户有执行权限）、stat.S_IWGRP（组用户有写权限）、stat.S_IRGRP（组用户有读权限）、stat.S_IRWXG（组用户有全部权限）、stat.S_IXUSR（所有者有执行权限）、stat.S_IWUSR（所有者有写权限）、stat.S_IRUSR（所有者有读权限）、stat.S_IRWXU（所有者有全部权限）、stat.S_IREAD（Windows 将该文件设为只读的）、stat.S_IWRITE（Windows 将该文件设为可写的）。
	os.chown(path, uid, gid)	更改文件的所有者（UNIX 下有小）。uid 为用户 id，gid 为组 id。
	os.fchmod(fd, mode)	改变一个文件的访问权限。mode 同 chrnod
	os.fchown(fd, uid, gid)	改变文件的所有者。
与文件访问相关的函数	os.open(file, flags[, mode])	打开一个文件，并且设置打开选项，返回文件描述符。flags 可选以下一个或多个值：os.O_RDONLY（只读的方式打开）、os.O_WRONLY（只写的方式打开）、os.O_RDWR（读写的方式打开）、os.O_NONBLOCK（打开时不阻塞）、os.O_APPEND（追加的方式打开）、os.O_CREAT（创建并打开一个新文件）、os.O_TRUNC（打开一个文件并截断它的长度为 0（必须有写权限））、os.O_EXCL（创建文件时，如果指定的文件存在，则返回错误）、os.O_SHLOCK（自动获取共享锁）、os.O_EXLOCK（自动获取独立锁）、os.O_DIRECT（消除或减少缓存效果）、os.O_FSYNC（同步写入）、os.O_NOFOLLOW（不追踪软链接）。
	os.read(fd, n)	从文件描述符 fd 中读取最多 n 个字节，返回读到的字符串。
	os.write(fd, str)	将字节串写入文件描述符 fd 中，返回实际写入的字节串长度。
	os.close(fd)	关闭文件描述符 fd。
	os.lseek(fd, pos, how)	从 how 开始移动文件指针。
	os.fdopen(fd[, mode[, bufsize]])	通过文件描述符 fd 打开文件，并返回对应的文件对象。
	os.closerange(fd_low, fd_high)	关闭从 fd_low（包含）到 fd_high（不包含）范围的所有文件描述符。
	os.dup(fd)	复制文件描述符。
	os.dup2(fd,fd2)	将一个文件描述符 fd 复制到另一个文件描述符 fd2 中。
	os.ftruncate(fd, length)	将 fd 对应的文件截断到 length 长度（不应该超过文件大小）。
	os.remove(path)	删除 path 对应的文件。
	os.link(src, dst)	创建从 src 到 dst 的硬链接。
	os.symlink(src, dst)	创建从 src 到 dst 的符号链接，对应于 Windows 的快捷方式。

os 模块函数的用法：

```
import os

# 与目录相关的演示
print(os.getcwd())          # 获取当前目录：G:\publish\codes\12.7
os.chdir('../12.6')         # 改变当前目录
print(os.getcwd())          # 再次获取当前目录：G:\publish\codes\12.6
os.mkdir('my_dir', 0o755)                   # 直接在当前目录下创建目录
os.makedirs("abc/xyz/wawa", 0o755)      # 递归创建目录
# 0o777 代表所有者可读 / 可写 / 可执行、组用户可读 / 可写 / 可执行、其他用户可读 / 可写 / 可执行
os.rmdir('my_dir')                          # 直接删除当前目录下的子目录
os.removedirs("abc/xyz/wawa")           # 递归删除子目录
os.rename('my_dir', 'your_dir')                     # 直接重命名当前目录下的子目录
os.renames("abc/xyz/wawa", 'foo/bar/haha')      # 递归重命名子目录

# 与权限相关的演示
ret = os.access('.', os.F_OK|os.R_OK|os.W_OK|os.X_OK)          # 判断当前目录的权限
print("os.F_OK|os.R_OK|os.W_OK|os.X_OK – 返回值 :", ret)
ret = os.access('os.access_test.py', os.F_OK|os.R_OK|os.W_OK)   # 判断 os.access_test.py 文件的权限
print("os.F_OK|os.R_OK|os.W_OK – 返回值 :", ret)
os.chmod('os.chmod_test.py', stat.S_IREAD)      # 将 os.chmod_test.py 文件改为只读
ret = os.access('os.chmod_test.py', os.W_OK)    # 判断是否可写
print("os.W_OK – 返回值 :", ret)

# 与文件访问相关的演示
f = os.open('abc.txt', os.O_RDWR|os.O_CREAT)          # 以读写、创建方式打开文件
# 写入文件内容
len1 = os.write(f, ' 水晶潭底银鱼跃，\n'.encode('utf–8'))
len2 = os.write(f, ' 清徐风中碧竿横。\n'.encode('utf–8'))
os.lseek(f, 0, os.SEEK_SET)     # 将文件指针移动到开始处
data = os.read(f, len1 + len2)      # 读取文件内容
print(data)                         # 打印读取到字节串
print(data.decode('utf–8'))         # 将字节串恢复成字符串
os.close(f)
os.symlink('os.link_test.py', 'tt')   # 为 os.link_test.py 文件创建快捷方式
os.link('os.link_test.py', 'dst')     # 为 os.link_test.py 文件创建硬连接（Windows 上就是复制文件）
```

Python 数据库编程

支持 SQL 标准的数据库（如 SQListe、MySQL、SQL Server 等）在 Python 中都有对应的客户端模块，并且所有数据库的绝大多数功能基本上都是相同的，都遵守 Python 制订的 DB API 协议（DB API 是一个规范，定义了一系列必须的对象和数据库存取方式，以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口）。Python 中，任何支持 2.0 版本 DB API 的数据库模块都必须定义 3 个描述模块特性的全局变量。

1. apilevel：用于显示数据库模块的 API 版本号（1.0 或 2.0）。如果这个变量不存在，则可能该数据库模块暂时不支持 2.0 版本的 API。
2. threadsafety：指定数据库模块的线程安全等级（0 ~ 3）。其中 3 代表该模块完全是线程安全的；1 表示该模块具有部分线程安全性，线程可以共享该模块，但不能共享连接；0 表示线程完全不能共享该模块。
3. paramstyle：指定当 SQL 语句需要参数时，可以使用哪种风格的参数。该变量可能返回如下变量值：
   - format：表示在 SQL 语句中使用 Python 标准的格式化字符串代表参数。例如，在程序中需要参数的地方使用 %s，接下来程序即可为这些参数指定参数值。
   - pyformat：表示在 SQL 语句中使用扩展的格式代码代表参数。比如使用 %(name)，这样即可使用包含 key 为 name 的字典为该参数指定参数值。
   - qmark：表示在 SQL 语句中使用问号（?）代表参数。在 SQL 语句中有几个参数，全部用问号代替。
   - numeric：表示在 SQL 语句中使用数字占位符（:N）代表参数。例如：:1 代表一个参数，:2 也表示一个参数，这些数字相当于参数名，因此它们不一定需要连续。
   - named：表示在 SQL 语句中使用命名占位符（:name）代表参数。例如 :name 代表一个参数，:age 也表示一个参数。

- **数据库 API 的核心类**

遵守 DB API 2.0 协议的数据库模块会提供一个 connect() 函数，用于连接数据库，并返回数据库连接对象。

数据库连接对象的方法和属性	功能描述
cursor()	打开游标。
commit()	提交事务。
rollback()	回滚事务。
close()	关闭数据库连接。
in_transaction	判断当前是否处于事务中。

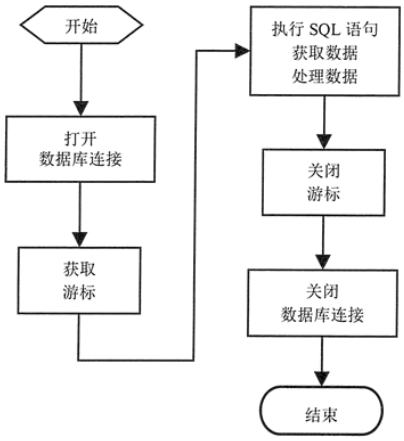
游标对象是 Python DB API 的核心对象，用于执行各种 SQL 语句（包括 DDL、DML、select 查询语句等）。

数据库连接对象的方法和属性	功能描述
execute(sql[, parameters])	执行 SQL 语句。parameters 用于为 SQL 语句中的参数指定值。
executemany(sql, seq_of_parameters)	重复执行 SQL 语句。可以通过 seq_of_parameters 序列为 SQL 语句中的参数指定值，该序列有多少个元素，SQL 语句被执行多少次。
executescript(sql_script)	这不是 DB API 2.0 的标准方法。可以直接执行包含多条 SQL 语句。
fetchone()	获取查询结果集的下一行。如果没有下一行，则返回 None。
fetchmany(size=cursor.arraysize)	返回查询结果集的下 N 行组成的列表。如果没有更多的数据行，则返回空列表。
fetchall()	返回查询结果集的全部行组成的列表。
close()	关闭游标。
rowcount	该只读属性返回受 SQL 语句影响的行数。对于 executemany() 方法，该方法所修改的记录条数也可通过该属性获取。
lastrowid	该只读属性可获取最后修改行的 rowid。
arraysize	用于设置或获取 fetchmany() 默认获取的记录条数，该属性默认为 1。有些数据库模块没有该属性。
description	获取最后一次查询返回的所有列的信息。
connection	返回创建游标的数据库连接对象。有些数据库模块没有该属性。

- **操作数据库的基本流程**

Python 的 DB API 2.0 由一个 connect() 开始，涉及数据库连接和游标两个核心 API。它们的分工如下：

- **数据库连接**：用于获取游标、控制事务。
- **游标**：执行各种 SQL 语句。



Python DB API 2.0 操作数据库的基本流程如下：

1. 调用 connect() 方法打开数据库连接，该方法返回数据库连接对象。
2. 通过数据库连接对象打开游标。
3. 使用游标执行 SQL 语句（包括 DDL、DML、select 查询语句等）。如果执行的是查询语句，则处理查询数据。
4. 关闭游标。
5. 关闭数据库连接。

Python 数据库编程（SQLite）

SQLite 不是一个客户端 / 服务器结构的数据库引擎，而是一种嵌入式数据库，它的数据库其实就是一个文件。SQLite 将整个数据库（包括定义、表、索引以及数据本身）作为一个单独的、可跨平台使用的文件存储在主机中。

Python 内置了 SQLite 3，在使用 SQLite 时，不需要安装任何模块，使用 import 语句引入即可使用。

创建数据表

使用 connect() 函数可以打开或创建一个数据库

```
import sqlite3    # 导入访问 SQLite 的模块
conn = sqlite3.connect('first.db') # ①、打开或创建数据库（创建内存中的数据库可使用 :memory）
c = conn.cursor()                  # ②、获取游标
c.execute('''create table user_tb(
    _id integer primary key autoincrement,
    name text,
    pass text,
    gender text)''')               # ③、执行 DDL 语句创建数据表
c.execute('''create table order_tb(
    _id integer primary key autoincrement,
    item_name text,
    item_price real,
    item_number real,
    user_id inteter,
    foreign key(user_id) references user_tb(_id) )''')        # 执行 DDL 语句创建数据表
c.close()                          # ④、关闭游标
conn.close()                       # ⑤、关闭连接
```

SQLite 内部只支持 NULL、INTEGER、REAL（浮点数）、TEXT（文本）和 BLOB（大二进制对象）这 5 种数据类型，但实际上 SQLite 完全可以接受 varchar(n)、char(n)、decimal(p, s) 等数据类型，只不过 SQLite 会在运算或保存时将它们转换为上面 5 种数据类型中相应的类型。

SQLite 允许把各种类型的数据保存到任何类型的字段中，可以不用关心声明该字段所使用的数据类型。（例如，可以把字符串类型的值存入 INTEGER 类型的字段中，也可以把数值类型的值存入布尔类型的字段中……）但有一种情况例外，被定义为“INTEGER PRIMARY KEY”的字段只能存储 64 位整数，当使用这种字段保存除整数以外的其他类型的数据时，SQLite 会产生错误。因此在编写建表语句时可以省略各数据列后面的类型声明。

安装 SQLite Expert 工具

SQLite Expert 是用来查看和管理 SQLite 数据库的工具。安装 SQLite Expert 工具的步骤如下：

1. 登录 http://www.sqliteexpert.com/download.html 站点来下载 SQLite Expert，该工具提供了两个版本：免费的个人版和收费的商业版。此处选择免费的个人版。将页面滚动到下方，找到“SQLite Expert Personal 5.x”，然后单击下方的链接（64 位操作系统选择 64bit 版，32 位操作系统选择 32bit 版）。
2. 下载完成后，单击该文件开始安装（安装过程和安装普通的 Windows 软件完全相同）。
3. 安装完成后，启动 SQLite Expert 工具。
4. 单击工具栏中的“Open Database”按钮，找到前面程序所创建的 first.db 文件，然后单击“打开”按钮，SQLite Expert 将会打开 first.db 文件所代表的数据库。
5. 打开 first.db 数据库之后，可以在 SQLite Expert 工具中看到该数据库包含两个数据表。随便选中一个数据表，就可以在右边看到该数据表的详细信息，包括数据列（Columns）、主键（Primary Key）、索引（Indexs）、外键（Foreign Keys）、唯一约束（Unique Constraints）等。

执行 SQL 语句

- **execute() 方法**　可以执行 DML（数据库操纵语言）的 insert、update、delete 语句，即可以对数据库执行插入、修改和删除数据操作。

```
import sqlite3    # 导入访问 SQLite 的模块
conn = sqlite3.connect('first.db')          # ①、打开或创建数据库
c = conn.cursor()                           # ②、获取游标
# ③、调用执行 insert 语句插入数据
c.execute('insert into user_tb values(null, ?, ?, ?)',  (' 孙悟空 ', '123456', 'male'))
c.execute('insert into order_tb values(null, ?, ?, ?, ?)', (' 鼠标 ', '34.2', '3', 1))
conn.commit()
c.close()                   # ④、关闭游标
conn.close()                # ⑤、关闭连接
```

- **executemany() 方法**　可以多次执行同一条 SQL 语句。

```
import sqlite3    # 导入访问 SQLite 的模块
conn = sqlite3.connect('first.db')          # ①、打开或创建数据库
c = conn.cursor()                           # ②、获取游标
# ③、调用 executemany() 方法把同一条 SQL 语句执行多次
c.executemany('insert into user_tb values(null, ?, ?, ?)', (('sun','123456','male'),
   ('bai','123456','female'), ('zhu','123456','male'), ('niu','123456','male'), ('tang','123456','male')))
# executemany() 方法也可以重复执行 update 语句或 delete 语句
c.executemany('update user_tb set name=? where _id=?',
   ((' 小孙 ', 2), (' 小白 ', 3), (' 小猪 ', 4), (' 小牛 ', 5), (' 小唐 ', 6)))
conn.commit()
c.close()                   # ④、关闭游标
conn.close()                # ⑤、关闭连接
```

- **executescript() 方法**　可以执行一段 SQL 脚本。。

```
import sqlite3    # 导入访问 SQLite 的模块
conn = sqlite3.connect('first.db')          # ①、打开或创建数据库
c = conn.cursor()                           # ②、获取游标
# ③、调用 executescript() 方法执行一段 SQL 脚本
c.executescript("""
   insert into user_tb values(null, ' 武松 ', '3444', 'male');
   insert into user_tb values(null, ' 林冲 ', '44444', 'male');
   create table item_tb(_id integer primary key autoincrement, name, price);
   """)
conn.commit()
c.close()                   # ④、关闭游标
conn.close()                # ⑤、关闭连接
```

SQLite 数据库模块还为数据库**连接对象**提供了如下 3 个方法：

- execute(sql[, parameters])：执行一条 SQL 语句。
- executemany(sql[, parameters])：根据序列重复执行 SQL 语句。
- executescript(sql_script)：执行 SQL 脚本。

获取数据（fetchone()、fetchmany() 和 fetchall()）

select 语句执行完成后可以得到查询结果。可通过游标的以下方法来获取查询结果。

- fetchone()：用于获取一条记录；
- fetchmany(n)：用于获取 n 条记录；
- fetchall()：用于获取全部记录。

```
import sqlite3    # 导入访问 SQLite 的模块
conn = sqlite3.connect('first.db')          # ①、打开或创建数据库
c = conn.cursor()                           # ②、获取游标
# ③、调用执行 select 语句查询数据
c.execute('select * from user_tb where _id > ?', (2,))      # 查询所有 _id 大于 2 的记录
print(' 查询返回的记录数 :', c.rowcount)
for col in (c.description):  print(col[0], end='\t')     # 通过游标的 description 属性获取列信息
print('\n----------------------------------')
while True:
    row = c.fetchone()     # 获取一行记录，每行数据都是一个元组
    if not row : break     # 如果抓取的 row 为 None，退出循环
    print(row)
    print(row[1] + '-->' + row[2])
c.close()                  # ④、关闭游标
conn.close()               # ⑤、关闭连接
```

也可以使用 fetchmany(n) 或 fetchall() 方法一次获取多条记录。

```
import sqlite3    # 导入访问 SQLite 的模块
conn = sqlite3.connect('first.db')          # ①、打开或创建数据库
c = conn.cursor()                           # ②、获取游标
# ③、调用执行 select 语句查询数据
c.execute('select * from user_tb where _id > ?', (2,))      # 查询所有 _id 大于 2 的记录
print(' 查询返回的记录数 :', c.rowcount)
for col in (c.description):  print(col[0], end='\t')     # 通过游标的 description 属性获取列信息
print('\n----------------------------------')
while True:
    rows = c.fetchmany(3)          # 每次抓取 3 条记录，该方法返回一个由 3 条记录组成的列表
    if not rows :  break           # 如果抓取的 rows 为 None，退出循环
    for r in rows:  print(r)       # 再次使用循环遍历获取的列表
c.close()                  # ④、关闭游标
conn.close()               # ⑤、关闭连接
```

一般来说，在程序中应该尽量避免使用 fetchall() 来获取查询返回的全部记录。这是因为程序可能并不清楚实际查询会返回多少条记录，如果查询返回的记录数量太多，那么调用 fetchall() 一次获取全部记录可能会导致内存开销过大，情况严重时可能导致系统崩溃。

注册自定义函数

- **自定义函数　create_function(name, num_params, func)**

create_function 方法可注册一个自定义函数，在 SQL 语句中就可以使用该函数了。该方法包含 3 个参数：

- name：指定注册的自定义函数的名字。
- num_params：指定自定义函数所需参数的个数。
- func：指定自定义函数对应的函数。

- **自定义聚集函数** create_aggregate(name, num_params, aggregate_class)

  标准的 SQL 提供了 5 个标准的聚集函数：sum()（统计总和）、avg()（统计平均值）、count()（统计记录条数）、max()（统计最大值）、min()（统计最小值）。如果需要在 SQL 语句中使用与其他业务相关的聚集函数，就可以用 create_aggregate 方法注册一个自定义聚集函数。该方法包含 3 个参数：

  - name：指定自定义聚集函数的名字。
  - num_params：指定聚集函数所需的参数。
  - aggregate_class：指定聚集函数的实现类。该类必须实现 step(self, params...) 和 finalize(self) 方法，其中 step() 方法对于查询所返回的每条记录各执行一次；finalize(self) 方法只在最后执行一次，该方法的返回值将作为聚集函数最后的返回值。

- **自定义比较函数** create_collation(name, callable)

  标准的 SQL 使用 order by 子句对查询结果进行排序，如果需要按业务相关规则进行排序，可用 create_collation(name, callable) 方法注册一个自定义比较函数，该方法包含 2 个参数：

  - name：指定自定义比较函数的名字。
  - callable：指定自定义比较函数对应的函数。该函数包含两个参数，并对这两个参数进行大小比较，如果该方法返回正整数，系统认为第一个参数更大；如果返回负整数，系统认为第二个参数更大；如果返回 0，系统认为两个参数相等。（callable 函数的参数以 Python（bytes）字节串的形式传入，因此系统默认会以 UTF-8 字符集将字符串编码成字节串后传入 callable 函数）。

```
import sqlite3    # 导入访问 SQLite 的模块
def reverse_ext(st):      # 定义一个普通函数，准备注册为 SQL 中的自定义函数
   return '[' + st[::-1] + ']'          # 对字符串反转，前后加方括号
class MinLen:             # 定义一个普通类，准备注册为 SQL 中的自定义聚集函数
   def __init__(self): self.min_len = None
   def step(self, value):
      if self.min_len is None :     # 如果 self.min_len 还未赋值，直接将当前 value 赋值给 self.min_lin
         self.min_len = value; return
      # 找到一个长度更短的 value，用 value 代替 self.min_len
      if len(self.min_len) > len(value): self.min_len = value
   def finalize(self):  return self.min_len
def my_collate(st1, st2):          # 定义一个普通函数，准备注册为 SQL 中的自定义聚合函数
   # 去掉字符串第一个、最后一个字符后比较大小
   if st1[1: -1] == st2[1: -1]: return 0
   elif st1[1: -1] > st2[1: -1]: return 1
   else:  return -1
conn = sqlite3.connect('first.db')                  # 打开或创建数据库
conn.create_function('enc', 1, reverse_ext)         # 调用 create_function 注册自定义函数：enc
conn.create_aggregate('min_len', 1, MinLen)         # 调用 create_aggregate 注册自定义聚集函数：min_len
conn.create_collation('sub_cmp', my_collate)        # 调用 create_collation 注册自定义比较函数：sub_cmp
c = conn.cursor()                                   # 获取游标
# 在 SQL 语句中使用 enc 自定义函数
c.execute('insert into user_tb values(null, ?, enc(?), ?)', (' 贾宝玉 ', '123456', 'male'))
# 在 SQL 语句中使用 min_len 自定义聚集函数
c.execute('select min_len(pass) from user_tb')
# 在 SQL 语句中使用 sub_cmp 自定义的比较函数
c.execute('select * from user_tb order by pass collate sub_cmp')
for row in c:  print(row)  # 采用 for 循环遍历游标
conn.commit()
c.close()                  # ④、关闭游标
conn.close()               # ⑤、关闭连接
```

Python 数据库编程（MySQL）

MySQL 是一款开源的数据库软件，由于其免费特性得到了全世界用户的喜爱，是目前使用人数最多的数据库。

安装 MySQL 数据库

安装 MySQL 数据库与安装普通程序并没有太大的区别，关键是在配置 MySQL 数据库时需要注意选择支持中文的编码集。在 Windows 平台上下载和安装 MySQL 数据库的步骤：

1. 登录 http://dev.mysql.com/downloads/mysql/ 站点，建议下载 MySQL 数据库社区版（Community）的最新版本。可根据自己所用的 Windows 平台，选择下载相应的 MSI Installer 安装文件。。
2. 下载完成后，双击开始安装 MySQL 数据库。
3. 勾选安装界面下方的“I accept the license terms”复选框，单击“Next”按钮。然后勾选“Custom”单选钮，单击“Next”按钮。然后选择安装 MySQL 所需的组件（**这里需要勾选 Connector/Python，即 Python 连接 MySQL 的模块**），并选择 MySQL 数据库及数据文件的安装路径。
4. 单击“Next”按钮，MySQL Installer 会检查系统环境是否满足安装 MySQL 数据库的要求。如果满足要求，则可以直接单击“Next”按钮开始安装；如果不符合条件，请根据 MySQL 提示先安装相应的系统组件，然后再重新安装 MySQL 数据库。
5. 单击安装结束对话框下方的“Next”按钮，开始配置 MySQL 数据库。选中“Standalone MySQL Server/Classic MySQL Replication”单选钮，即可进行更详细的配置。
6. 两次单击“Next”按钮，允许用户设置 MySQL 的 root 账户密码，也允许添加更多的用户。单击“Next”按钮，将依次出现一系列对话框，这些对话框对配置影响不大，直接单击“Next”按钮，直至 MySQL 数据库配置成功。

Python pip（管理模块工具）

Python 自带了一个 pip 工具用来查看、管理所安装的各种模块。

- **查看已安装的模块：pip show packagename**
  如：pip show mysql-connector-python 查看 mysql-connector-python 模块
- **卸载已安装的模块：pip uninstall packagename**
  如：pip uninstall mysql-connector-python 卸载 mysql-connector-python 模块
- **安装模块：pip install packagename[== 版本号]**
  如：pip install mysql-connectar-python 安装 mysql-connector-python 模块
  pip install mysql-connectar-python == 8.0.17 安装 mysql-connector-python 模块 8.0.17 版

检查 MySQL 全局属性并创建数据库

在使用 mysql-connector-python 模块操作 MySQL 数据库之前，同样先检查一下该模块的全局属性。

```
>>> import mysql.connector                # 导入 mysql.connector 包
>>> mysql.connector.apilevel              # 检查 apilevel（API 版本号）
'2.0'
>>> mysql.connector.threadsafety          # 检查 threadsafety（线程安全等级）
1
>>> mysql.connector.paramstyle            # 检查 paramstyle（参数风格）
'pyformat'
>>>
```

通过“开始”菜单中的“MySQL → MySQL Server 8.0->MySQL 8.0 Command Line Client-Unicode”启动 MySQL 的命令行客户端，输入在 root 账户设置的密码，即可进入 MySQL 的命令行客户端，然后输入如下命令来创建 python 数据库：

```
create database python;
```

MySQL 数据库执行 DDL 语句

Python 连接 MySQL 数据库时可指定远程服务器 IP 地址（默认 localhost）和端口（默认 3306）。

```
import mysql.connector                                          # 导入访问 MySQL 的模块
conn = mysql.connector.connect(user='root', password='root', host='localhost', port='3306',
    database='python', use_unicode=True)                        # ①、连接数据库
c = conn.cursor()                                               # ②、获取游标
c.execute('''create table user_tb( user_id int primary key auto_increment, name varchar(255),
    pass varchar(255), gender varchar(255))''')                 # ③、执行 DDL 语句创建数据表
c.execute('''create table order_tb( order_id integer primary key auto_increment,
    item_name varchar(255), item_price double, item_number double, user_id int,
    foreign key(user_id) references user_tb(user_id) )''')      # 执行 DDL 语句创建数据表
c.close()                                                       # ④、关闭游标
conn.close()                                                    # ⑤、关闭连接
```

MySQL 数据库执行 DML 语句

MySQL 数据库模块同样可以使用游标的 execute() 方法执行 DML 的 insert、update、delete 语句，对数据库进行插入、修改和删除数据操作。。

```
import mysql.connector                                          # 导入访问 MySQL 的模块
conn = mysql.connector.connect(user='root', password='root', host='localhost', port='3306',
    database='python', use_unicode=True)                        # ①、连接数据库
c = conn.cursor()                                               # ②、获取游标
# 调用 execute 执行 insert 语句插入数据
c.execute('insert into user_tb values(null, %s, %s, %s)', (' 孙悟空 ', '123456', 'male'))
c.execute('insert into order_tb values(null, %s, %s, %s, %s)', (' 鼠标 ', '34.2', '3', 1))
# 调用 executemany() 方法把同一条 SQL 语句（插入数据）执行多次
c.executemany('insert into user_tb values(null, %s, %s, %s)',  (('sun', '123456', 'male'),
    ('bai', '123456', 'female'), ('zhu', '123456', 'male'), ('niu', '123456', 'male'), ('tang', '123456', 'male')))
# 调用 executemany() 方法把同一条 SQL 语句（更新数据）执行多次
c.executemany('update user_tb set name=%s where user_id=%s',
    ((' 小孙 ', 2), (' 小白 ', 3), (' 小猪 ', 4),  (' 小牛 ', 5), (' 小唐 ', 6)))
print(' 修改的记录条数：', c.rowcount)                           # 通过 rowcount 获取被修改的记录条数
conn.commit()
c.close()                                                       # ④、关闭游标
conn.close()                                                    # ⑤、关闭连接
```

MySQL 数据库模块的连接对象有一个 autoconunit 属性，如果设为 True，则关闭连接的事务支持，程序每次执行 DML 语句之后都会自动提交，这样程序就无须调用连接对象的 commit() 方法来提交事务了。

```
import mysql.connector                                  # 导入访问 MySQL 的模块
conn = conn = mysql.connector.connect(user='root', password='root', host='localhost', port='3306',
    database='python', use_unicode=True)                # ①、连接数据库
conn.autocommit = True                                  # 将 autocommit 设置 True，关闭事务
# 下面执行的 DML 语句会自动提交
...
c.close()                                               # ④、关闭游标
conn.close()                                            # ⑤、关闭连接
```

MySQL 数据库执行查询语句

使用 MySQL 数据库模块执行查询语句，需注意 SQL 语句中的占位符。

```
import mysql.connector                                  # 导入访问 MySQL 的模块
conn = conn = mysql.connector.connect(user='root', password='root', host='localhost', port='3306',
    database='python', use_unicode=True)                # 连接数据库
c = conn.cursor()                                       # 获取游标
c.execute('select * from user_tb where user_id > %s', (2,))   # 调用 execute 执行 select 语句查询数据
# 通过游标的 description 属性获取列信息
for col in (c.description): print(col[0], end='\t')
# 直接使用 for 循环来遍历游标中的结果集
for row in c: print(row+ '：' +row[1] + '-->' + row[2])
# 支持 fetchone()、fetchmany()、fetchall() 方法
while True:
    rows = c.fetchmany(3)          # 每次抓取 3 条记录，该方法返回一个多个元组组成的列表
    if not rows :  break           # 如果抓取的 row 为 None，退出循环
    for r in rows:  print(r)       # 再次使用循环遍历获取的列表
c.close()                                               # ④、关闭游标
conn.close()                                            # ⑤、关闭连接
```

调用数据库存储过程（callproc() 方法）

callproc(self, procname, args=()) 用于调用数据库存储过程。其中：procname 参数代表存储过程的名字，args 参数则用于为存储过程传入参数。

```
import mysql.connector                                  # 导入访问 MySQL 的模块
conn = conn = mysql.connector.connect(user='root', password='root', host='localhost', port='3306',
    database='python', use_unicode=True)                # 连接数据库
c = conn.cursor()                                       # 获取游标
# 调用 callproc() 方法执行存储过程 add_pro，存储过程需要 3 个参数，最后一个参数是传出参数
result_args = c.callproc('add_pro', (5, 6, 0))
print(result_args)          # 返回的 result_args 既包含了传入参数的值，也包含了传出参数的值
print(result_args[2])       # 如果只想访问传出参数的值，可直接访问 result_args 的第 3 个元素
c.close()                                               # ④、关闭游标
conn.close()                                            # ⑤、关闭连接
```

PyMySQL 模块下载和安装

PyMySQL 是支持 Python 操作数据库的模块，也可以实现 Python 和 MySQL 之间的连接。类似的模块也称为接口程序，即通过该程序，可以对另外一个对象进行操作。

PyMySQL 的安装方式主要有 2 种，一种是先下载源码，然后再安装；另一种是使用 pip 自动下载并安装。

PyMySQL 源码的下载地址 https://github.com/PyMySQL/PyMySQL。

在命令窗口运行 pip install PyMySQL 命令安装 PyMySQL。

安装完成之后，在 Python 交互模式下运行命令 import pymysql 检查是否安装成功。

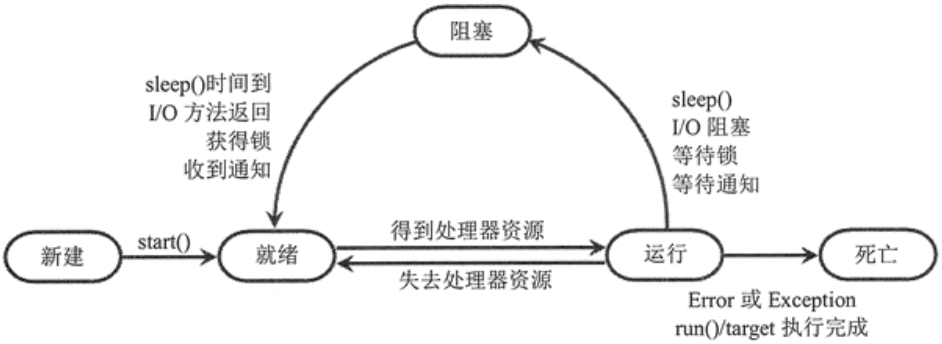
Python 多线程编程

操作系统可以同时执行多个任务，每个任务通常是一个程序，每一个运行中的程序就是一个**进程**。

进程可以同时执行多个任务，每一个任务就是一个**线程**。当一个进程里只有一个线程时叫**单线程**。超过一个线程就叫**多线程**。Python 语言提供了非常优秀的多线程支持，程序可以通过简单的方式来启动多线程。

在多线程中，有一个主线程来完成整个进程从开始到结束的全部操作，其他的线程会在主线程的运行过程中被创建或退出。多个线程共享父进程里的全部资源，需要确保线程不会妨碍同一进程中的其他线程。多线程是并发执行的（即同一时刻，主程序只允许有一个线程执行）。

- **线程的生命周期** 当线程被创建并启动后，不会直接进入执行状态，也不会一直处于执行状态，它会经历新建（new）、就绪（Ready）、运行（Running）、阻塞（Blocked）和死亡（Dead）5 种状态。



当主线程结束时，其他线程不受任何影响，并不会随之结束。一旦子线程启动起来后，就拥有和主线程相同的地位，不会受主线程的影响。

线程的创建和启动

线程开发相关的模块有 2 个：_thread（不建议使用，Python 3 以前版本中 thread 模块的重命名，仅提供了低级别的、原始的线程支持，以及一个简单的锁，功能比较有限）和 threading（推荐使用，Python 3 之后的线程模块，提供了功能丰富的多线程支持）。创建线程主要通过两种方式：

- **调用 Thread 类的构造器创建线程**

  __init__(self, group=None, target=None, name=None, args=(), kwargs=None, *,daemon=None)

  - group：指定该线程所属的线程组（目前该参数还未实现，因此只能设为 None）。
  - target：指定该线程要调度的目标方法。
  - args：指定一个元组，以位置参数的形式为 target 指定的函数传入参数。
  - kwargs：指定一个字典，以关键字参数的形式为 target 指定的函数传入参数。
  - daemon：指定所构建的线程是否为后代线程。

  通过 Thread 类的构造器创建并启动多线程的步骤：

  1. 调用 Thread 类的构造器创建线程对象（target 参数指定的函数将作为线程执行体）。
  2. 调用线程对象的 start() 方法启动该线程。

```
import threading
def action(max):          # 定义一个普通的 action 函数，该函数准备作为线程执行体
    for i in range(max):
        print(threading.current_thread().getName() +  " " + str(i))
# 下面是主程序（也就是主线程的执行体）
for i in range(100):
    print(threading.current_thread().getName() +  " " + str(i))
    if i == 20:
        t1 =threading.Thread(target=action,args=(100,)); t1.start()    # 创建并启动第一个线程
        t2 =threading.Thread(target=action,args=(100,)); t2.start()    # 创建并启动第二个线程
print(' 主线程执行完成 !')
```

- **继承 Thread 类创建线程类**

1. 定义 Thread 类的子类，并重写 run() 方法（线程执行体，线程需要完成的任务）。
2. 创建 Thread 子类的实例，即创建线程对象。
3. 调用线程对象的 start() 方法来启动线程。

```
import threading
class FkThread(threading.Thread):        # 通过继承 threading.Thread 类来创建线程类
    def __init__(self):
        threading.Thread.__init__(self); self.i = 0
    def run(self):              # 重写 run() 方法作为线程执行体
        while self.i < 100:
            print(threading.current_thread().getName() +  " " + str(self.i))
            self.i += 1
# 下面是主程序（也就是主线程的执行体）
for i in range(100):
    print(threading.current_thread().getName() +  " " + str(i))
    if i == 20:
        ft1 = FkThread(); ft1.start()          # 创建并启动第一个线程
        ft2 = FkThread(); ft2.start()          # 创建并启动第二个线程
print(' 主线程执行完成 !')
```

- **启动线程 (start() 方法)**

启动线程使用 start() 方法，而不是 run() 方法。调用 start() 方法来启动线程，系统会把 run() 方法当成线程执行体来处理；如果直接调用线程对象的 run() 方法，run() 方法立即就会被执行，而且在该方法返回之前其他线程无法并发执行，此时变成单线程程序，系统会把线程对象当成一个普通对象，run() 方法是一个普通方法，不是线程执行体。

只能对处于新建状态的线程调用 start() 方法。如果对同一个线程重复调用 start() 方法，将引发 RuntimeError 异常。

- **线程睡眠 (sleep() 函数)**

time 模块的 sleep(secs) 函数让当前正在执行的线程暂停一段时间，并进入阻塞状态（secs 指定线程阻塞的秒数）。在其睡眠时间段内，该线程不会获得执行的机会，即使系统中没有其他可执行的线程，处于 sleep() 中的线程也不会执行，因此 sleep() 函数常用来暂停程序的运行。

- **threading 模块的函数：**
  - threading.current_thread()：返回当前正在执行的线程对象。
  - threading.enumerate()：返回一个正运行线程的 list。“正运行”是指线程处于“启动后，且在结束前”状态，不包括“启动前”和“终止后”状态。
  - threading.activeCount()：返回正在运行的线程数量。与 len(threading.enumerate()) 有相同的结果。
- **Thread 类的实例方法：**
  - getName()：返回调用它的线程名字。
  - join(timeout=None)：让一个线程等待另一个线程完成。当在某个程序执行流中调用其他线程的 join() 方法时，调用线程将被阻塞，直到被 join() 方法加入的 join 线程执行完成。通常由使用线程的程序调用，以将大问题划分成许多小问题，并为每个小问题分配一个线程。当所有的小问题都得到处理后，再调用主线程来进一步操作。可以指定一个 timeout 参数，用来指定等待被 join 的线程的时间最长为 timeout 秒。如果在 timeout 秒内被 join 的线程还没有执行结束，则不再等待。
- **后台线程（又称守护线程或精灵线程）**

后台线程是一种在后台运行的线程，为其他线程提供服务（如 Python 解释器的垃圾回收线程就是典型的后台线程）。如果所有的前台线程都死亡了，那么后台线程会自动死亡。

创建后台线程有两种方式：①主动将线程的 daemon 属性设置为 True（必须在 start() 方法调用之前进行，否则会引发 RuntimeError 异常）；②后台线程启动的线程默认是后台线程。

多线程的安全（互斥锁）

互斥锁的作用是解决数据不同步问题。但是当线程使用共享数据时，可能会由于数据不同步产生“错误情况”，这是由系统的线程调度具有一定的随机性造成的。

- **类的线程安全性** 线程安全的类具有如下特征：
  - 该类的对象可以被多个线程安全地访问。
  - 每个线程在调用该对象的任意方法之后，都将得到正确的结果。
  - 每个线程在调用该对象的任意方法之后，该对象都依然保持合理的状态。

  一般来说，不可变类总是线程安全的，因为它的对象状态不可改变；可变对象需要额外的方法来保证其线程安全。可变类的线程安全是以降低程序的运行效率作为代价的，为了减少线程安全所带来的负面影响，程序可以采用如下策略：
  - 不要对线程安全类的所有方法都进行同步，只对那些会改变竞争资源（竞争资源也就是共享资源）的方法进行同步。
  - 如果可变类有两种运行环境，单线程环境和多线程环境，则应该为该可变类提供两种版本，即线程不安全版本和线程安全版本。在单线程环境中使用线程不安全版本以保证性能，在多线程环境中使用线程安全版本。
- **互斥锁同步线程**

  threading 模块提供了 Lock 和 RLock 两个类，都提供了加锁和释放锁方法：
  - acquire(blocking=True, timeout=-1)：请求对 Lock 或 RLock 加锁，timeout 指定加锁多少秒。
  - release()：释放锁。

  Lock 和 RLock 的区别：
  - threading.Lock：基本锁，每次只能锁定一次，其余的锁请求，需等待锁释放后才能获取。

    Lock 是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对 Lock 对象加锁，线程在开始访问共享资源之前应先请求获得 Lock 对象。当对共享资源访问完成后，程序释放对 Lock 对象的锁定。
  - threading.RLock：可重入锁，在同一个线程中可以多次锁定和释放，锁定和释放次数必须相等。

    RLock 常用于实现线程安全的控制，当加锁和释放锁出现在不同的作用范围内时，通常建议使用 finally 块来确保在必要时释放锁。使用 RLock 的代码格式如下：

```
class X:
    def m () :      # 定义需要保证线程安全的方法
        self.lock.acquire()  # 加锁
        try :
            # 需要保证线程安全的代码
            # ... 方法体
        finally :      # 使用 finally 块来保证释放锁
            self.lock.release()          # 修改完成，释放锁
```

- **死锁**

当两个线程相互等待对方释放同步监视器时就会发生死锁。死锁是不应该在程序中出现的，在编写程序时应该尽量避免出现死锁。常见的解决死锁方式有：

- **避免多次锁定**。尽量避免同一个线程对多个 Lock 进行锁定。
- **具有相同的加锁顺序**。如果多个线程需要对多个 Lock 进行锁定，则应该保证它们以相同的顺序请求加锁。
- **使用定时锁**。程序在调用 acquire() 方法加锁时可指定 timeout 参数，超过 timeout 秒后会自动释放对 Lock 的锁定，这样就可以解开死锁了。
- **死锁检测**。死锁检测是一种依靠算法机制来实现的死锁预防机制，它主要是针对那些不可能实现按序加锁，也不能使用定时锁的场景的。

多线程的通信

当线程在系统中运行时，线程的调度具有一定的透明性，通常程序无法准确控制线程的轮换执行，如果有需要，Python 可通过线程通信来保证线程协调运行。

- **condition 实现线程通信**

  使用 Condition 可以让那些已经得到 Lock 对象却无法继续执行的线程释放 Lock 对象，也可以唤醒其他处于等待状态的线程。

  将 Condition 对象与 Lock 对象组合使用，可以为每个对象提供多个等待集（wait-set）。因此，Condition 对象总是需要有对应的 Lock 对象（从 Condition 的构造器 __init__(self, lock=None) 可以看出）。在创建 Condition 时可通过 lock 参数传入要绑定的 Lock 对象；如果不指定 lock 参数，会自动创建一个与之绑定的 Lock 对象。Condition 类提供了如下几个方法：

  - acquire([timeout])/release()：调用 Condition 关联的 Lock 的 acquire() 或 release() 方法。
  - wait([timeout])：导致当前线程进入 Condition 的等待池等待通知并释放锁，直到其他线程调用该 Condition 的 notify() 或 notify_all() 方法来唤醒该线程。timeout 指定该线程最多等待多少秒。
  - notify()：唤醒在该 Condition 等待池中的单个线程并通知它，收到通知的线程将自动调用 acquire() 方法尝试加锁。如果所有线程都在该 Condition 等待池中等待，则会任意选择唤醒其中一个线程。
  - notify_all()：唤醒在该 Condition 等待池中等待的所有线程并通知它们。

- **Queue 队列实现线程通信**

  queue 模块下提供了几个阻塞队列，用于实现线程通信。模块主要提供三个类，分别代表三种队列，它们的主要区别就在于进队列、出队列的不同：

  - queue.Queue(maxsize=0)：FIFO（先进先出）常规队列，maxsize 用于限制队列的大小（0 或负数表示无限制）。如果队列的大小达到队列的上限，就会加锁，再次加入元素时就会被阻塞，直到队列中的元素被消费。
  - queue.LifoQueue(maxsize=0)：LIFO（后进先出）队列，与 Queue 的区别就是出队列的顺序不同。
  - PriorityQueue(maxsize=0)：优先级队列，优先级最小的元素先出队列。

  这三个队列类的属性和方法基本相同：

  - Queue.qsize()：返回队列的实际大小，也就是该队列中包含几个元素。
  - Queue.empty()：判断队列是否为空。
  - Queue.full()：判断队列是否已满。
  - Queue.put(item, block=True, timeout=None)：向队列中放入元素。如果队列已满，且 block 参数为 True（阻塞），当前线程被阻塞，timeout 指定阻塞时间（None 表示一直阻塞，直到该队列的元素被消费）；如果队列已满，且 block 参数为 False（不阻塞），则引发 queue.FULL 异常。
  - Queue.put_nowait(item)：向队列中放入元素，不阻塞。相当于 Queue.put 中 block 设置为 False。
  - Queue.get(item, block=True, timeout=None)：从队列中取出元素（消费元素）。如果队列已空，且 block 参数为 True（阻塞），当前线程被阻塞，timeout 指定阻塞时间（None 表示一直阻塞，直到有元素被放入队列中）；如果队列已空，且 block 参数为 False（不阻塞），则引发 queue.EMPTY 异常。
  - Queue.get_nowait(item)：从队列中取出元素，不阻塞。相当于 Queue.get 中 block 设置为 False。

- **Event 实现线程通信**

  Event 是一种非常简单的线程通信机制，一个线程发出一个 Event，另一个线程可通过该 Event 被触发。

  Event 本身管理一个内部旗标，程序可以通过 Event 的 set() 方法将该旗标设置为 True，也可以调用 clear() 方法将该旗标设置为 False。程序可以调用 wait() 方法来阻塞当前线程，直到 Event 的内部旗标被设置为 True。

  Event 有点类似于 Condition 和旗标的结合体，但 Event 本身并不带 Lock 对象，因此如果要实现线程同步，还需要额外的 Lock 对象。Event 提供了如下方法：

  - is_set()：返回 Event 的内部旗标是否为 True。
  - set()：将 Event 的内部旗标设置为 True，并唤醒所有处于等待状态的线程。
  - clear()：将 Event 的内部旗标设置为 False，通常接下来会调用 wait() 方法来阻塞当前线程。
  - wait(timeout=None)：阻塞当前线程。

线程池

线程池在系统启动时创建大量空闲的线程，程序只要将一个函数提交给线程池，线程池就会启动一个空闲的线程来执行它。当函数执行结束后，该线程并不会死亡，而是再次返回到线程池中变成空闲状态，等待执行下一个函数。这样可以很好地提升性能。

使用线程池可以有效地控制系统中并发线程的数量。当系统中包含有大量的并发线程时，会导致系统性能急剧下降，甚至导致解释器崩溃，线程池的最大线程数可以控制系统中并发线程的数量。

● 线程池的使用

线程池的基类是 concurrent.futures 模块中的 Executor 类，它提供了两个子类，即 ThreadPoolExecutor（用于创建线程池）和 ProcessPoolExecutor（用于创建进程池）。使用线程池 / 进程池来管理并发编程，只要将相应的 task 函数提交（submit）给线程池 / 进程池，剩下的事情就由线程池 / 进程池来搞定。submit 方法会返回一个 Future 对象（用于获取线程任务函数的返回值）。

Exectuor 提供了如下常用方法：

- submit(fn, *args, **kwargs)：将 fn 函数提交给线程池。*args 代表传给 fn 函数的参数，*kwargs 代表以关键字参数的形式为 fn 函数传入参数。
- map(func, *iterables, timeout=None, chunksize=1)：该函数类似于全局函数 map(func, *iterables)，只是该函数将会启动多个线程，以异步方式立即对 iterables 执行 map 处理。
- shutdown(wait=True)：关闭线程池。调用 shutdown() 后的线程池不再接收新任务，但会将以前所有的已提交任务执行完成。当线程池中的所有任务都执行完成后，该线程池中的所有线程都会死亡。

Future 提供了如下方法：

- cancel()：取消该 Future 代表的线程任务。如果该任务正在执行，不可取消，则该方法返回 False；否则，程序会取消该任务，并返回 True。
- cancelled()：返回 Future 代表的线程任务是否被成功取消。
- running()：如果该 Future 代表的线程任务正在执行、不可被取消，该方法返回 True。
- done()：如果该 Funture 代表的线程任务被成功取消或执行完成，则该方法返回 True。
- result(timeout=None)：获取该 Future 代表的线程任务最后返回的结果。如果 Future 代表的线程任务还未完成，该方法将会阻塞当前线程，timeout 指定最多阻塞多少秒。
- exception(timeout=None)：获取该 Future 代表的线程任务所引发的异常。如果该任务成功完成，没有异常，则该方法返回 None。
- add_done_callback(fn)：为该 Future 代表的线程任务注册一个“回调函数”，当该任务成功完成时，程序会自动触发该 fn 函数。

使用线程池来执行线程任务的步骤如下：

1. 调用 ThreadPoolExecutor 类的构造器创建一个线程池。
2. 定义一个普通函数作为线程任务。
3. 调用 ThreadPoolExecutor 对象的 submit() 方法来提交线程任务。
4. 当不想提交任何任务时，调用 ThreadPoolExecutor 对象的 shutdown() 方法来关闭线程池。

● 获取执行结果

调用了 Future 的 result() 方法来获取线程任务的返回值，会阻塞当前主线程，只有等到线程任务完成后，result() 方法的阻塞才会被解除。

为了防止阻塞，可通过 Future 的 add_done_callback() 方法来添加回调函数，该回调函数形如 fn(future)。当线程任务完成后，程序会自动触发该回调函数，并将对应的 Future 对象作为参数传给该回调函数。

使用 add_done_callback() 方法来获取线程任务的返回值：

```
from concurrent.futures import ThreadPoolExecutor
import threading
import time
def action(max):          # 定义一个准备作为线程任务的函数
    my_sum = 0
    for i in range(max):
        print(threading.current_thread().name + '  ' + str(i))
        my_sum += i
    return my_sum
# 由于线程池实现了上下文管理协议（Context Manage Protocol），
# 因此，可以使用 with 语句来管理线程池，这样即可避免手动关闭线程池。
with ThreadPoolExecutor(max_workers=2) as pool:       # 创建一个包含 2 条线程的线程池
    future1 = pool.submit(action, 50)       # 向线程池提交一个 task, 50 会作为 action() 函数的参数
    future2 = pool.submit(action, 100)      # 向线程池提交一个 task, 100 会作为 action() 函数的参数
    def get_result(future):
        print(future.result())          # 查看 future1 代表的任务返回的结果
    future1.add_done_callback(get_result)          # 为 future1 添加线程完成的回调函数
    future2.add_done_callback(get_result)          # 为 future2 添加线程完成的回调函数
    print('--------------')
    print(future1.done())  # 判断 future1 代表的任务是否结束
    time.sleep(3)
    print(future2.done())  # 判断 future2 代表的任务是否结束
```

使用 Executor 的 map() 方法来启动线程，并收集线程任务的返回值：

```
from concurrent.futures import ThreadPoolExecutor
import threading
import time
def action(max):          # 定义一个准备作为线程任务的函数
    my_sum = 0
    for i in range(max):
        print(threading.current_thread().name + '  ' + str(i))
        my_sum += i
    return my_sum
with ThreadPoolExecutor(max_workers=4) as pool:       # 创建一个包含 4 条线程的线程池
    # 使用线程执行 map 计算
    # 后面元组有 3 个元素，因此程序启动 3 条线程来执行 action 函数
    results = pool.map(action, (50, 100, 150))
    print('--------------')
    for r in results:
        print(r)
```

使用 map() 方法来启动启动 3 个线程，并收集线程的执行结果，不仅具有代码简单的优点，而且虽然程序会以并发方式来执行 action() 函数，但最后收集的 action() 函数的执行结果，依然与传入参数的结果保持一致。也就是说，上面 results 的第一个元素是 action(50) 的结果，第二个元素是 action(100) 的结果，第三个元素是 action(150) 的结果。

返回线程局部变量（Local() 函数）

threading 模块下提供了一个 local() 函数，可以返回一个线程局部变量，通过使用线程局部变量可以很简捷地隔离多线程访问的竞争资源，从而简化多线程并发访问的编程处理。

线程局部变量（Thread Local Variable）的功用其实非常简单，就是为每一个使用该变量的线程都提供一个变量的副本，使每一个线程都可以独立地改变自己的副本，而不会和其他线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量一样。

线程局部变量和其他同步机制一样，都是为了解决多线程中对共享资源的访问冲突的。在普通的同步机制中，是通过为对象加锁来实现多个线程对共享资源的安全访问的。由于共享资源是多个线程共享的，所以要使用这种同步机制，就需要很细致地分析什么时候对共享资源进行读写，什么时候需要锁定该资源，什么时候释放对该资源的锁定等。在这种情况下，系统并没有将这份资源复制多份，只是采用安全机制来控制对这份资源的的访问而已。

线程局部变量从另一个角度来解决多线程的并发访问问题。线程局部变量将需要并发访问的资源复制多份，每个线程都拥有自己的资源副本，从而也就没有必要对该资源进行同步了。线程局部变量提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的整个变量放到线程局部变量中，或者把该对象中与线程相关的状态放入线程局部变量中保存。

线程局部变量并不能替代同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对共享资源的并发访问，是多个线程之间进行通信的有效方式；而线程局部变量是为了隔离多个线程的数据共享，从根本上避免多个钱程之间对共享资源（变量）的竞争，也就不需要对多个线程进行同步了。

通常建议，如果多个线程之间需要共享资源，以实现线程通信，则使用同步机制；如果仅仅需要隔离多个线程之间的共享冲突，则可以使用线程局部变量。

```
import threading
from concurrent.futures import ThreadPoolExecutor
mydata = threading.local()          # 定义线程局部变量
def action (max):          # 定义准备作为线程执行体使用的函数
    for i in range(max):
        try: mydata.x += i
        except: mydata.x = i
        # 访问 mydata 的 x 的值
        print('%s mydata.x 的值为 : %d' %(threading.current_thread().name, mydata.x))
with ThreadPoolExecutor(max_workers=2) as pool:          # 使用线程池启动两个子线程
    pool.submit(action , 10)
    pool.submit(action , 10)
```

控制函数在特定时间执行（Timer 定时器）

Thread 类有一个 Timer 子类，可用于控制指定函数在特定时间内执行一次。如果要使用 Timer 控制函数多次重复执行，则需要再执行下一次调度。如果想取消 Timer 的调度，则可调用 Timer 对象的 cancel() 函数。

```
from threading import Timer
import time
count = 0          # 定义总共输出几次的计数器
def print_time():
    print(" 当前时间：%s" % time.ctime())
    global t, count;  count += 1
    if count < 10: # 如果 count 小于 10，开始下一次调度
        t = Timer(1, print_time); t.start()
t = Timer(1, print_time)  # 指定 1 秒后执行 print_time 函数
t.start()
```

任务调度器（sched.scheduler 类）

如果需要执行更复杂的任务调度，则可使用 sched 模块提供的 sched.scheduler 类（代表一个任务调度器）。sched.scheduler(timefunc=time.monotonic, delayfunc=time.sleep) 构造器支持两个参数：

- timefunc：指定生成时间戳的时间函数，默认使用 time.monotonic 来生成时间戳。
- delayfunc：指定阻塞程序的函数，默认使用 time.sleep 函数来阻塞程序。

sched.scheduler 调度器支持如下常用属性和方法：

- scheduler.enterabs(time, priority, action, argument=(), kwargs={})：指定在 time 时间点执行 action 函数，argument 和 kwargs 用于向 action 函数传入参数，其中 argument 使用位置参数的形式传入参数，kwargs 使用关键字参数的形式传入参数。该方法返回一个 event，它可作为 cancel() 方法的参数用于取消该调度。priority 参数指定该任务的优先级，当在同一个时间点有多个任务需要执行时，优先级高（值越小代表优先级越高）的任务会优先执行。
- scheduler.enter(delay, priority, action, argument=(),kwargs={})：该方法与上一个方法基本相同，只是 delay 参数用于指定多少秒之后执行 action 任务。
- scheduler.cancel(event)：取消任务。如果传入的 event 参数不是当前调度队列中的 event，程序将会引发 ValueError 异常。
- scheduler.empty()：判断当前该调度器的调度队列是否为空。
- scheduler.run(blocking=True)：运行所有需要调度的任务。如果调用该方法的 blocking 参数为 True，该方法将会阻塞线程，直到所有被调度的任务都执行完成。
- scheduler.queue：该只读属性返回该调度器的调度队列。

```
import sched, time
import threading
# 定义线程调度器
s = sched.scheduler()
# 定义被调度的函数
def print_time(name='default'):
    print("%s 的时间 : %s" % (name, time.ctime()))
print(' 主线程：', time.ctime())
# 指定 10 秒之后执行 print_time 函数
s.enter(10, 1, print_time)
# 指定 5 秒之后执行 print_time 函数，优先级为 2
s.enter(5, 2, print_time, argument=(' 位置参数 ',))
# 指定 5 秒之后执行 print_time 函数，优先级为 1
s.enter(5, 1, print_time, kwargs={'name': ' 关键字参数 '})
# 执行调度的任务
s.run()
print(' 主线程：', time.ctime())
```

Python 多进程编程

除可以进行多线程编程之外，Python 还支持使用多进程来实现并发编程。

进程创建和启动（os.fork 方法）

os 模块提供了一个 fork() 方法启动两个进程（一个是父进程，一个是 fork 出来的子进程）来执行从 os.fork() 开始的所有代码。

fork() 方法无参数，只有一个返回值（表明哪个进程在执行）：0（表明是 fork 出来的子进程在执行）和非 0（表明是父进程在执行，值为 fork() 出来的子进程的进程 ID）。

```
import os
print(' 父进程（%s）开始执行 ' % os.getpid())
pid = os.fork()   # 开始 fork 一个子进程，从这行代码开始，下面代码都会被两个进程执行
print(' 进程进入：%s' % os.getpid())
if pid == 0:        # 如果 pid 为 0，表明子进程
   print(' 子进程，其 ID 为 (%s)， 父进程 ID 为 (%s)' % (os.getpid(), os.getppid()))
else:
   print(' 我 (%s) 创建的子进程 ID 为 (%s).' % (os.getpid(), pid))
print(' 进程结束：%s' % os.getpid())
```

进程创建和启动（multiprocessing.Process）

multiprocessing 模块下提供了 Process 来创建新进程。有两种方式：

- 以指定函数作为 target，创建 Process 对象即可创建新进程。
- 继承 Process 类，并重写它的 run() 方法来创建进程类，程序创建 Process 子类的实例作为进程。

Process 类的方法和属性：

- run()：重写该方法可实现进程的执行体。
- start()：该方法用于启动进程。
- join([timeout])：类似于线程的 join() 方法，当前进程必须等待被 join 的进程执行完成才能向下执行。
- name：该属性用于设置或访问进程的名字。
- is_alive()：判断进程是否还活着。
- daemon：该属性用于判断或设置进程的后台状态。
- pid：返回进程的 ID。
- authkey：返回进程的授权 key。
- terminate()：中断该进程。

● 以指定函数作为 target 创建新进程

```
import multiprocessing
import os
def action(max):          # 定义一个普通的 action 函数，该函数准备作为进程执行体
   for i in range(max): print("(%s) 子进程（父进程 :(%s)）：%d" %(os.getpid(), os.getppid(), i))
if __name__ == '__main__':      # 必须先判断。下面是主程序（也就是主进程）
   for i in range(100):
      print("(%s) 主进程 : %d" % (os.getpid(), i))
      if i == 20:
         mp1 = multiprocessing.Process(target=action,args=(100,)); mp1.start() # 创建并启动第一个进程
         mp2 = multiprocessing.Process(target=action,args=(100,)); mp2.start() # 创建并启动第二个进程
         mp2.join()       # 主进程必须等 mp2 进程完成后才能向下执行
   print(' 主进程执行完成 !')
```

● 继承 Process 类创建子进程

继承 Process 类创建子进程的步骤如下：

1. 定义继承 Process 的子类，重写其 run() 方法准备作为进程执行体。
2. 创建 Process 子类的实例。
3. 调用 Process 子类的实例的 start() 方法来启动进程。

```
import multiprocessing
import os
class MyProcess(multiprocessing.Process):
    def __init__(self, max):
        self.max = max
        super().__init__()
    def run(self):              # 重写 run() 方法作为进程执行体
        for i in range(self.max): print("(%s) 子进程（父进程 :(%s)）：%d" %(os.getpid(), os.getppid(), i))
if __name__ == '__main__':      # 必须先判断。下面是主程序（也就是主进程）
    for i in range(100):
        print("(%s) 主进程 : %d" % (os.getpid(), i))
        if i == 20:
            mp1 = MyProcess(100); mp1.start()       # 创建并启动第一个进程
            mp2 = MyProcess(100); mp2.start()       # 创建并启动第二个进程
            mp2.join()          # 主进程必须等 mp2 进程完成后才能向下执行
    print(' 主进程执行完成 !')
```

进程启动方式

根据平台的支持，Python 支持 3 种启动进程的方式：

○ spawn：父进程启动一个全新的 Python 解释器进程。子进程只能继承那些处理 run() 方法所必需的资源。不必要的文件描述器和 handle 都不会被继承。这种方式启动进程，其效率比 fork 或 forkserver 方式低得多。Windows 只支持 spawn 方式来启动进程（Windows 平台的默认启动方式）。

○ fork：父进程使用 os.fork() 启动一个 Python 解释器进程。子进程会继承父进程的所有资源，因此子进程基本等效于父进程。这种方式只在 UNIX 平台上有效（UNIX 平台的默认启动方式）。

○ forkserver：程序将会启动一个服务器进程。在以后的时间内，当程序再次请求启动新进程时，父进程都会连接到该服务器进程，请求由服务器进程来 fork 新进程。通过这种方式启动的进程不需要从父进程继承资源。这种方式只在 UNIX 平台上有效。

multiprocessing 模块提供了 set_start_method() 函数和 get_context() 方法（传入字符串获取 Context 对象）来设置启动方式（必须将这行设置代码放在所有与多进程有关的代码之前）。

```
import multiprocessing; import os
def foo(q):
    print(' 被启动的新进程 : (%s)' % os.getpid()); q.put('Python')
if __name__ == '__main__':
    # 方式一：使用 set_start_method() 设置启动进程
    multiprocessing.set_start_method('spawn'); q = multiprocessing.Queue()
    mp = multiprocessing.Process(target=foo, args=(q, ))  # 创建进程
    # 方式二：用 get_context() 设置启动方式，并获取 Context 对象
    ctx = multiprocessing.get_context('fork'); q = ctx.Queue()
    mp = ctx.Process(target=foo, args=(q, ))                    # 创建进程
    mp.start()      # 启动进程
    print(q.get())  # 获取队列中的消息
    mp.join()
```

进程池

与线程池类似，启动多个进程也可以使用进程池来管理。通过 multiprocessing 模块的 Pool() 函数创建进程池（实际上是 multiprocessing.pool.Pool 类）。进程池具有如下常用方法：

- apply(func[, args[, kwds]])：将 func 函数提交给进程池处理。args 是传给 func 的参数，kwds 代表传给 func 的关键字参数。该方法会被阻塞直到 func 函数执行完成。
- apply_async(func[, args[, kwds[, callback[, error_callback]]]])：apply() 方法的异步版本，不会被阻塞。callback 指定 func 函数完成后的回调函数，error_callback 指定 func 函数出错后的回调函数。
- map(func, iterable[, chunksize])：类似于 Python 的 map() 全局函数，只不过此处使用新进程对 iterable 的每一个元素执行 func 函数。
- map_async(func, iterable[, chunksize[, callback[, error_callback]]])：map() 方法的异步版本，不会被阻塞。callback 指定 func 函数完成后的回调函数，error_callback 指定 func 函数出错后的回调函数。
- imap(func, iterable[, chunksize])：map() 方法的延迟版本。
- imap_unordered(func, iterable[, chunksize])：功能类似于 imap() 方法，但该方法不能保证所生成的结果（包含多个元素）与原 iterable 中的元素顺序一致。
- starmap(func, iterable[,chunksize])：功能类似于 map() 方法，但该方法要求 iterable 的元素也是 iterable 对象，程序会将每一个元素解包之后作为 func 函数的参数。
- close()：关闭进程池。在调用该方法之后，该进程池不能再接收新任务，它会把当前进程池中的所有任务执行完成后再关闭自己。
- terminate()：立即中止进程池。
- join()：等待所有进程完成。

```
import multiprocessing
import time
import os
def action1(name='default'):
    print('(%s) 进程正在执行，参数为 : %s' % (os.getpid(), name)); time.sleep(3)
def action2(max):
    my_sum = 0
    for i in range(max):
        print('(%s) 进程正在执行 : %d' % (os.getpid(), i))
        my_sum += i
    return my_sum
if __name__ == '__main__':
    # 使用 apply_async() 方法启动进程：
    pool = multiprocessing.Pool(processes=4)      # 创建包含 4 条进程的进程池
    # 将 action1 分 3 次提交给进程池
    pool.apply_async(action1)
    pool.apply_async(action1, args=(' 位置参数 ', ))
    pool.apply_async(action1, kwds={'name': ' 关键字参数 '})
    pool.close()
    pool.join()
    # 使用 map() 方法来启动进程：
    with multiprocessing.Pool(processes=4) as pool:
        # 使用进程执行 map 计算，后面元组有 3 个元素，因此程序启动 3 条进程来执行 action2 函数
        results = pool.map(action2, (50, 100, 150))
        for r in results:
            print(r)
```

进程间通信

Python 为进程通信提供了两种机制：

- **使用 Queue 实现进程间通信：**一个进程向 Queue 中放入数据，另一个进程从 Queue 中读取数据。

  multiprocessing 模块下的 Queue 和 queue 模块下的 Queue 基本类似，都提供了 qsize()、empty()、full()、put()、put_nowait()、get()、get_nowait() 等方法。区别是 multiprocessing 模块下的 Queue 为进程提供服务，而 queue 模块下的 Queue 为线程提供服务。

```
import multiprocessing
def f(q):
    print('(%s) 进程开始放入数据 ...' % multiprocessing.current_process().pid)
    q.put('Python')
if __name__ == '__main__':
    q = multiprocessing.Queue()                          # 创建进程通信的 Queue
    p = multiprocessing.Process(target=f, args=(q,))     # 创建子进程
    p.start()                                            # 启动子进程
    print('(%s) 进程开始取出数据 ...' % multiprocessing.current_process().pid)
    print(q.get())                                       # 取出数据：Python
    p.join()
```

- **使用 Pipe 实现进程间通信：**Pipe 代表连接两个进程的管道。调用 multiprocessing.Pipe() 函数来创建一个管道，返回两个 PipeConnection 对象，代表管道的两个连接端，分别用于连接通信的两个进程。

  PipeConnection 对象包含如下常用方法：

  - send(obj)：发送一个 obj 给管道的另一端，另一端使用 recv() 方法接收。该 obj 必须是可 picklable 的（Python 的序列化机制），并且序列化之后超过 32MB，否则很可能会引发 ValueError 异常。
  - recv()：接收另一端通过 send() 方法发送过来的数据。
  - fileno()：关于连接所使用的文件描述器。
  - close()：关闭连接。
  - poll([timeout])：返回连接中是否还有数据可以读取。
  - send_bytes(buffer[, offset[, size]])：发送字节数据。offset、size 参数默认发送 buffer 字节串的全部数据；如果指定了 offset 和 size 参数，则只发送 buffer 字节串中从 offset 开始、长度为 size 的字节数据。通过该方法发送的数据，应该使用 recv_bytes() 或 recv_bytes_into 方法接收。
  - recv_bytes([maxlength])：接收通过 send_bytes() 方法发送的数据，maxlength 指定最多接收的字节数。该方法返回接收到的字节数据。
  - recv_bytes_into(buffer[, offset])：功能与 recv_bytes() 方法类似，只是该方法将接收到的数据放在 buffer 中。

```
import multiprocessing
def f(conn):
    print('(%s) 进程开始发送数据 ...' % multiprocessing.current_process().pid)
    conn.send('Python')              # 使用 conn 发送数据
if __name__ == '__main__':
    parent_conn, child_conn = multiprocessing.Pipe()   # 创建 Pipe，返回两个 PipeConnection 对象
    p = multiprocessing.Process(target=f, args=(child_conn, ))      # 创建子进程
    p.start()                                                       # 启动子进程
    print('(%s) 进程开始接收数据 ...' % multiprocessing.current_process().pid)
    print(parent_conn.recv())                                       # 通过 conn 读取数据 Python
    p.join()
```

多进程编程和多线程编程优缺点

多进程编程和多钱程编程，都可以使用并行机制来提升系统的运行效率。**区别在于运行时所占的内存分布不同**，多钱程是共用一套内存的代码块区间；而多进程是各用一套独立的内存区间。

多进程的优点是稳定性好，一个子进程的崩溃不会影响主进程以及其余进程。因此，常用多进程来实现守护服务器的功能；**缺点**是代价非常大，操作系统要给每个进程分配固定的资源，并且操作系统对进程的总数会有一定的限制，若进程过多，操作系统调度都会存在问题，会造成假死状态。

多线程的优点是效率较高一些，适用于批处理任务等功能；**缺点**是任何一个线程崩溃都可能造成整个进程的崩溃，因为它们共享了进程的内存资源池。

多线程编程和多进程编程分别**适用于不同的场景**。对于计算密集型的任务，多进程效率会更高一下；而对于 IO 密集型的任务（比如文件操作，网络爬虫），采用多线程编程效率更高。

在大型的计算机集群系统中，通常都会将多进程程序分布运行在不同的计算机上协同工作。而每一台计算机上的进程内部，又会由多个线程来并行工作。

注意，对于任务数来说，无论是多进程编程或者多线程编程，其进程数或线程数都不能太多：

- 对于多进程编程来说，操作系统在切换任务时，会有一系列的保护现场措施，这要花费相当多的系统资源，若任务过多，则大部分资源都被用做干这些了，结果就是所有任务都做不好；
- 多线程编程也不是线程个数越多效率越高，通过下面的公式可以计算出线程数量最优的参考值。

$$\text{最佳线程数量} = \frac{\text{线程等待时间} + \text{线程}CPU\text{时间}}{\text{线程}CPU\text{时间}} * CPU\text{数量}$$

Python 网络编程

- **计算机网络**是把分布在不同地理区域的计算机与专门的外部设备用通信线路互联成一个规模大、功能强的网络系统，从而使众多的计算机可以方便地互相传递信息，共享硬件、软件、数据信息等资源。

  计算机网络是现代通信技术与计算机技术相结合的产物，主要功能有：资源共享、信息传输与集中处理、均衡负荷与分布处理、综合信息服务。

  计算机网络有很多种类型，根据不同的分类原则，可以得到不同类型的计算机网络。通常计算机网络是按照规模大小和延伸范围来分类的，常见的类型有：局域网（LAN）、城域网（MAN）和广域网（WAN）。Internet 可以被视为世界上最大的广域网。

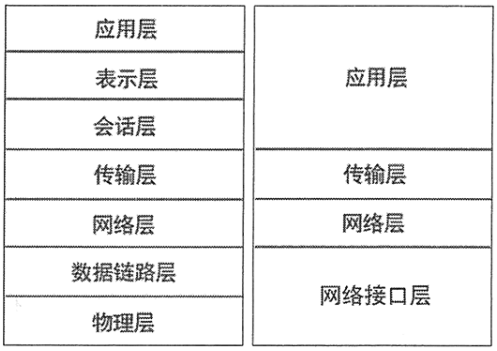
- **通信协议**是计算机网络中实现通信必须遵守的一些约定，负责对传输速率、传输代码、代码结构、传输控制步骤、出错控制等制定处理标准。

  通信协议通常由**语义部分**（用于决定双方对话的类型）、**语法部分**（用于决定双方对话的格式）和**变换规则**（用于决定通信双方的应答关系）三部分组成。

  OSI（Open System Interconnection）参考模型是由国际标准化组织（ISO）于 1978 年提出的“开放系统互连参考模型”，力求将网络简化，并以模块化的方式来设计网络。OSI 参考模型把计算机网络分成**物理层**、**数据链路层**、**网络层**、**传输层**、**会话层**、**表示层**、**应用层**七层，受到计算机界和通信业的极大关注，已成为各种计算机网络结构的参考标准。

  TCP/IP 协议是 IP（Internet Protocol）和 TCP（Transmission Control Protocol）的统称。**IP（网际协议）**是支持网间互联的数据报协议，提供了网间连接的完善功能，包括 IP 数据报规定的互联网络范围内的地址格式。**TCP（传输控制协议）**规定了一种可靠的数据信息传递服务。

  TCP/IP 协议最早出现在 UNIX 操作系统中，现在几乎所有的操作系统都支持 TCP/IP 协议，是 Internet 中最常用的基础协议。按照 TCP/IP 协议模型，网络模型通常被分为四层。



- **IP 地址**用于唯一标识网络中的一个通信实体（一个主机，或者是一台打印机，或者是路由器的某一个端口），而在基于 IP 协议的网络中传输的数据包，都必须使用 IP 地址来进行标识。

  IP 地址是数字型的，它是一个 32 位（32 bit）整数。但为了便于记忆，通常把它分成 4 个 8 位的二进制数，每 8 位之间用圆点隔开，每个 8 位整数都可以转换成一个 0~255 的十进制整数（日常看到的 IP 地址常常是 202.9.128.88 形式）。

  IP 地址被分成 A（10.0.0.0~10.255.255.255）、B（172.16.0.0~172.31.255.255）、C（192.168.0.0~192.168.255.255）、D、E 五类。

  NIC（Internet Network Information Center）统一负责全球 Internet IP 地址的规划和管理，而 InterNIC、APNIC、RIPE 三大网络信息中心则具体负责美国及其他地区的 IP 地址分配。其中 APNIC（总部在日本东京大学）负责亚太地区的 IP 地址管理，我国申请 IP 地址也要通过 APNIC。

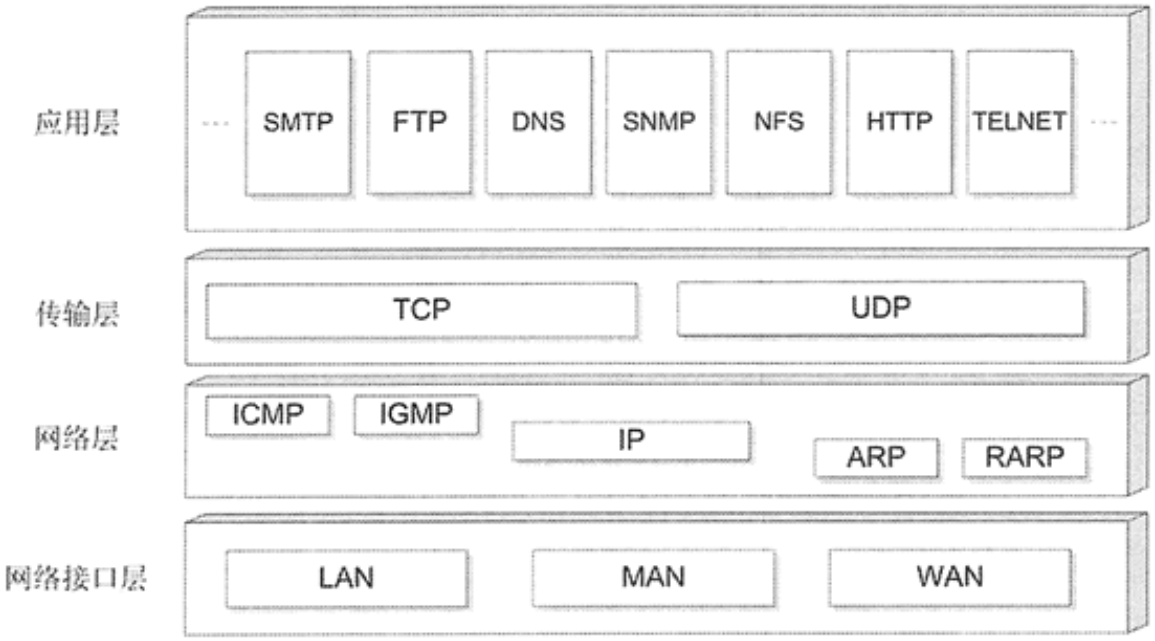
- **端口号**是一个 16 位的整数（0~65535），用于标识一个通信实体（IP 地址）中应用程序提供的网络服务。

  端口号是应用程序与外界交流的出入口，是一种抽象的软件结构，包括一些数据结构和 I/O（输入/输出）缓冲区。不同的应用程序处理不同端口上的数据，在同一台机器中不能有两个程序使用同一个端口。

  通常将端口分为如下三类：**公认端口**（Well Known Port，0~1023，紧密地绑定一些特定的服务）、**注册端口**（Registered Port，1024~49151，松散地绑定一些服务。应用程序通常应该使用这个范围内的端口）和**动态和/或私有端口**（Dynamic and/or Private Port，49152 ~ 65535，是应用程序使用的动态端口，应用程序一般不会主动使用这些端口）。

Python 网络编程模块

四层网络模型对应的网络协议如图所示：



- **网络层协议**主要是 IP，它是所有互联网协议的基础，ICMP（Internet Control Message Protocol）、IGMP（Internet Group Manage Protocol）、ARP（Address Resolution Protocol）、RARP（Reverse Address Resolution Protocol）等协议可认为是 IP 协议族的子协议。通常很少会直接基于网络层进行应用程序编程。
- **传输层协议**主要是 TCP 和 UDP，Python 提供了 socket 等模块针对传输层协议进行编程。
- **应用层协议**有 FTP、HTTP、TELNET 等很多，Python 为基于应用层协议的编程提供了丰富的支持。

Python 自带的标准库已经提供了很多与网络有关的模块，如果在使用时还觉得不够方便，可利用第三方模块来增强 Python 的功能。

python 模块	描述
socket	基于传输层 TCP、UDP 协议进行网络编程的模块
asyncore	socket 模块的异步版，支持基于传输层协议的异步通信
asynchat	asyncore 的增强版
cgi	基本的 CGI 支持
email	E-mail 和 MLME 消息处理模块
ftplib	支持 FTP 协议的客户端模块
httplib、http.client	支持 HTTP 协议以及 HTTP 客户揣的模块
imaplib	支持 IMAP4 协议的客户端模块
mailbox	操作不同格式邮箱的模块
mailcap	支持 Mailcap 文件处理的模块
nntplib	支持 NTTP 协议的客户端模块
smtplib	支持 SMTP 协议（发送邮件）的客户端模块
poplib	支持 POP3 协议的客户端模块
telnetlib	支持 TELNET 协议的客户端模块
urllib 及其子模块	支持 URL 处理的模块
xmlrpc、xmlrpc.server、xmlrpc.client	支持 XML-RPC 协议的服务器端和客户端模块

解析 URL（urllib.parse 模块）

URL（Uniform Resource Locator）对象代表统一资源定位器，它是指向互联网“资源”的指针。资源可以是简单的文件或目录，也可以是对复杂对象的引用。URL 一般由协议名、主机、端口和资源路径组成，即满足格式：protocol://host:port/path。

urllib 模块包含了多个用于处理 URL 的子模块：urllib.request（最核心的子模块，包含打开和读取 URL 的各种函数）、urllib.error（包含由 urllib.request 子模块所引发的各种异常）、urllib.parse（用于解析 URL）、urllib.robotparser（用于解析 robots.txt 文件）。

urllib.parse 子模块中用于解析 URL 地址和查询字符串的函数：

- urllib.parse.urlparse(urlstring, scheme='', allow_fragments=True)：用于解析 URL 字符串。返回一个 ParseResult 对象（tuple 的子类），可通过属性名或索引获取解析出来的数据。

属性名	元组索引	返回值	默认值
scheme	0	返回 URL 的 scheme	scheme 参数
netloc	1	网络位置部分（主机名＋端口）	空字符串
path	2	资源路径	空字符串
params	3	资源路径的附加参数	空字符串
query	4	查询字符串	空字符串
fragment	5	Fragment 标识符	空字符串
username		用户名	None
password		密码	None
hostname		主机名	None
port		端口	None

```
from urllib.parse import *
# 解析 URL 字符串
result = urlparse('http://www.crazyit.org:80/index.php;yeeku?name=fkit#frag')
print('scheme:', result.scheme, result[0])          # scheme: http http
print(' 主机和端口 :', result.netloc, result[1])      # www.crazyit.org:80 www.crazyit.org:80
print(' 主机 :', result.hostname)                    # 主机 : www.crazyit.org
print(' 端口 :', result.port)                        # 端口 : 80
print(' 资源路径 :', result.path, result[2])          # 资源路径 : /index.php /index.php
print(' 参数 :', result.params, result[3])           # 参数 : yeeku yeeku
print(' 查询字符串 :', result.query, result[4])       # 查询字符串 : name=fkit name=fkit
print('fragment:', result.fragment, result[5])      # fragment: frag frag
print(result.geturl())              #http://www.crazyit.org:80/index.php;yeeku?name=fkit#frag
# 解析以 // 开头的 URL
result = urlparse('//www.crazyit.org:80/index.php')
print('scheme:', result.scheme, result[0])          # 以 // 开头，缺少 scheme:
print(' 主机和端口 :', result.netloc, result[1])      # www.crazyit.org:80 www.crazyit.org:80
print(' 资源路径 :', result.path, result[2])          # 资源路径 : /index.php /index.php
# 解析没有 scheme，也没有双斜线（//）开头的 URL，整个 URL 都当成资源路径
result = urlparse('www.crazyit.org/index.php')
print('scheme:', result.scheme, result[0])          # 缺少 scheme:
print(' 主机和端口 :', result.netloc, result[1])      # 缺少主机和端口 :
print(' 资源路径 :', result.path, result[2])          # www.crazyit.org/index.php
```

- urllib.parse.urlunparse(parts)：urlparse 的反向操作，用于将解析结果反向拼接成 URL 地址。

```
result = urlunparse(('http', 'www.crazyit.org:80', 'index.php', 'yeeku', 'name=fkit', 'frag'))
print('URL 为 :', result)    # URL 为 : http://www.crazyit.org:80/index.php;yeeku?name=fkit#frag
```

- urllib.parse.parse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace')：用于解析查询字符串（application/x-www-form-urlencoded 类型的数据），并以 dict 形式返回解析结果。
- urllib.parse.parse_qsl(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace')：该函数用于解析查询字符串（application/x-www-form-urlencoded 类型的数据），并以列表形式返回解析结果。
- urllib.parse.urlencode(query, doseq=False, safe='', encoding=None, errors=None, quote_via=quote_plus)：将字典形式或列表形式的请求参数恢复成请求字符串。该函数相当于 parse_qs()、parse_qsl() 的逆函数。

```
# 解析查询字符串，返回 dict
result = parse_qs('name=fkit&name=%E7%96%AF%E7%8B%82java&age=12')
print(result)                    # {'name': ['fkit', ' 疯狂 java'], 'age': ['12']}
# 解析查询字符串，返回 list
result = parse_qsl('name=fkit&name=%E7%96%AF%E7%8B%82java&age=12')
print(result)                    # [('name', 'fkit'), ('name', ' 疯狂 java'), ('age', '12')]
# 将列表格式的请求参数恢复成请求参数字符串
print(urlencode(result))         # name=fkit&name=%E7%96%AF%E7%8B%82java&age=12
```

- urllib.parse.urljoin(base, url, allow_fragments=True)：用于将一个 base_URL 和另一个资源 URL 连接成代表绝对地址的 URL。主要可能出现 3 种情况：
  1. 被拼接的 URL 只是一个相对路径 path（不以斜线开头），该 URL 将会被拼接到 base 之后，如果 base 本身包含 path 部分，则用被拼接的 URL 替换 base 所包含的 path 部分。
  2. 被拼接的 URL 是一个根路径 path（以单斜线开头），该 URL 将会被拼接到 base 的域名之后。
  3. 被拼接的 URL 是一个绝对路径 path（以双斜线开头），该 URL 将会被拼接到 base 的 scheme 之后。

```
# 被拼接 URL 不以斜线开头
result = urljoin('http://www.crazyit.org/users/login.html', 'help.html')
print(result)                                    # http://www.crazyit.org/users/help.html
result = urljoin('http://www.crazyit.org/users/login.html', 'book/list.html')
print(result)                                    # http://www.crazyit.org/users/book/list.html
# 被拼接 URL 以斜线（代表根路径 path）开头
result = urljoin('http://www.crazyit.org/users/login.html', '/help.html')
print(result)                                    # http://www.crazyit.org/help.html
# 被拼接 URL 以双斜线（代表绝对 URL）开头
result = urljoin('http://www.crazyit.org/users/login.html', '//help.html')
print(result)                                    # http://help.html
```

读取资源（urllib.request 模块）

urllib.request 子模块包含一个非常实用的 urllib.request.urlopen(url, data=None) 方法，用于打开 url 指定的资源，并从中读取数据。根据请求 url 的不同，该方法的返回值会发生动态改变。如果 url 是一个 HTTP 地址，返回一个 http.client.HTTPResponse 对象。

```
from urllib.request import *
result = urlopen('http://www.crazyit.org/index.php')        # 打开 URL 对应的资源
data = result.read(326)             # 按字节读取数据
print(data.decode('utf-8'))         # 将字节数据恢复成字符串
with urlopen('http://www.crazyit.org/index.php') as f:      # 用 context manager 管理打开的 URL 资源
    data = f.read(326)              # 按字节读取数据
    print(data.decode('utf-8'))     # 将字节数据恢复成字符串
```

通过 data 属性向被请求的 URL 发送数据，请求方式可以是：

○ GET（无需 data 属性，直接把请求参数附加在 URL 之后即可）

```
import urllib.parse
params = urllib.parse.urlencode({'name': 'fkit', 'password': '123888'})
url = 'http://localhost/get.jsp?%s' % params                # 将请求参数添加到 URL 的后面
with urlopen(url=url) as f:
    print(f.read().decode('utf-8'))          # 读取服务器全部响应
```

○ POST（通过 data 属性）。

```
import urllib.parse
params = urllib.parse.urlencode({'name': ' 疯狂软件 ', 'password': '123888'})
params = params.encode('utf-8')
with urlopen("http://localhost/post.jsp", data=params) as f:        # 使用 data 指定请求参数
    print(f.read().decode('utf-8'))          # 读取服务器全部响应
```

○ PUT、PATCH、DELETE 等（需要使用 urllib.request.Request 来构建请求参数，构造器如下）

```
urllib.request.Request(  url,                    # 指定请求地址
                         data=None,              # 指定请求参数
                         headers={},             # 指定请求头
                         origin_req_host=None,
                         unverifiable=False,
                         method=None             # 指定请求方法
)
```

使用 urlopen() 函数打开远程资源时，第一个 url 参数既可以是 URL 字符串，也可以使用 urllib.request.Request 对象。

```
from urllib.request import *
params = 'put 请求数据 '.encode('utf-8')
# 创建 Request 对象，设置地址，数据和请求方式
req = Request(url='http://localhost/put', data=params, method='PUT')
# 通过 Request 的 add_header() 方法添加了一个 Referer 请求头
req.add_header('Referer', 'http://www.crazyit.org/')
with urlopen(req) as f:
    print(f.status)
    print(f.read().decode('utf-8'))
```

管理 cookie（http.cookiejar 模块）

如果程序直接使用 urlopen() 发送请求，而未管理与服务器之间的 session，那么服务器就无法识别两次请求是否是同一个客户端发出的。为了有效地管理 session，程序可引入 http.cookiejar 模块。

此外，程序还需要使用 OpenerDirector 对象来发送请求。按如下步骤进行操作：

1. 创建 http.cookiejar.CookieJar 对象或其子类的对象。
2. 以 CookieJar 对象为参数，创建 urllib.request.HTTPCookieProcessor 对象，该对象负责调用 CookieJar 来管理 cookie。
3. 以 HTTPCookieProcessor 对象为参数，调用 urllib.request.build_opener() 函数创建 OpenerDirector 对象。
4. 用 OpenerDirector 对象发送请求，该对象会通过 HTTPCookieProcessor 调用 CookieJar 来管理 cookie。

下面程序示范了先登录 Web 应用，然后访问 Web 应用中的被保护页面：

```
from urllib.request import *
import http.cookiejar, urllib.parse
cookie_jar = http.cookiejar.MozillaCookieJar('a.txt')        # 创建 CookieJar 对象，指定 cookie 保存的文件
cookie_processor = HTTPCookieProcessor(cookie_jar)  # 创建 HTTPCookieProcessor 对象
opener = build_opener(cookie_processor)                      # 创建 OpenerDirector 对象
user_agent = r'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
                    Chrome/56.0.2924.87 Safari/537.36'      # 定义模拟 Chrome 浏览器的 user_agent
headers = {'User-Agent':user_agent, 'Connection':'keep-alive'}   # 定义请求头
#-------------- 下面代码发送登录的 POST 请求 -----------------
params = {'name':'crazyit.org', 'pass':'leegang'}            # 定义登录系统的请求参数
postdata = urllib.parse.urlencode(params).encode()
# 创建向登录页面发送 POST 请求的 Request
request = Request('http://localhost:8888/test/login.jsp', data = postdata, headers = headers)
response = opener.open(request)          # 使用 OpenerDirector 发送 POST 请求
print(response.read().decode('utf-8'))    # 恭喜您，登录成功！
cookie_jar.save(ignore_discard=True, ignore_expires=True) # 将 cookie 信息写入磁盘文件
#-------------- 下面代码发送访问被保护资源的 GET 请求 -----------------
# 创建向 " 受保护页面 " 发送 GET 请求的 Request
request = Request('http://localhost:8888/test/secret.jsp', headers=headers)
response = opener.open(request)
print(response.read().decode())          # 获得受保护页面
```

读取 cookie 文件信息，模拟前面登录过的客户端，从而直接访问被保护页面了。

```
from urllib.request import *
import http.cookiejar, urllib.parse
cookie_jar = http.cookiejar.MozillaCookieJar('a.txt')        # 创建 CookieJar 对象，指定 cookie 保存的文件
cookie_jar.load('a.txt',ignore_discard=True,ignore_expires=True)          # 加载 cookie 信息
for item in cookie_jar:print('Name ='+ item.name+',Value ='+ item.value)  # 遍历 cookie 信息
cookie_processor = HTTPCookieProcessor(cookie_jar)  # 创建 HTTPCookieProcessor 对象
opener = build_opener(cookie_processor)                      # 创建 OpenerDirector 对象
user_agent = r'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
                    Chrome/56.0.2924.87 Safari/537.36'      # 定义模拟 Chrome 浏览器的 user_agent
headers = {'User-Agent':user_agent, 'Connection':'keep-alive'}   # 定义请求头
#-------------- 下面代码发送访问被保护资源的 GET 请求 -----------------
# 创建向 " 受保护页面 " 发送 GET 请求的 Request
request = Request('http://localhost:8888/test/secret.jsp', headers=headers)
response = opener.open(request)
print(response.read().decode())          # 获得受保护页面
```

建立 TCP 通信（socket 模块）

socket 模块为基于 TCP 协议的网络通信提供了良好的封装，socket 对象代表两端的通信端口，并通过 socket 进行网络通信。

- 创建 socket 对象，可通过该类的构造器来创建 socket 实例：

  socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)

  - family：用于指定网络类型。支持 socket.AF_UNIX（UNIX 网络）、socket.AF_INET（基于 IPv4 协议的网络）和 socket.AF_INET6（基于 IPv6 协议的网络）三个常量。
  - type：用于指定网络 Sock 类型。支持 SOCK_STREAM（默认值，创建基于 TCP 协议的 socket）、SOCK_DGRAM（创建基于 UDP 协议的 socket）和 SOCK_RAW（创建原始 socket）。
  - proto：用于指定协议号，如果没有特殊要求，该参数默认为 0，并可以忽略。
- socket 对象提供了如下常用方法：
  - socket.accept()：作为服务器端使用的 socket 调用该方法接收来自客户端的连接。
  - socket.bind(address)：作为服务器端使用的 socket 调用该方法，将该 socket 绑定到指定 address，该 address 可以是一个元组，包含 IP 地址和端口。
  - socket.close()：关闭连接，回收资源。
  - socket.connect(address)：作为客户端使用的 socket 调用该方法连接远程服务器。
  - socket.connect_ex(address)：同上，出错时，该方法不会抛出异常，而是返回一个错误标识符。
  - socket.listen([backlog])：作为服务器端使用的 socket 调用该方法进行监听。
  - socket.makefile(mode='r', buffering=None, *, encoding=None, errors=None, newline=None)：创建一个和该 socket 关联的文件对象。
  - socket.recv(bufsize[, flags])：接收 socket 中的数据。返回 bytes 对象代表接收到的数据。
  - socket.recv_into(buffer[, nbytes[, flags]])：同上，接收到的数据放入 buffer 中。
  - socket.recvfrom(bufsize[,flags])：同上，只是该方法的返回值是 (bytes, address) 元组。
  - socket.recvfrom_into(buffer[, nbytes[, flags]])：同上，接收到的数据放入 buffer 中。
  - socket.recvmsg(bufsize[, ancbufsize[, flags]])：接收来自 socket 的数据和辅助数据，返回一个长度为 4 的元组 (data, ancdata, msg_flags, address)，ancdata 代表辅助数据。
  - socket.recvmsg_into(buffers[, ancbufsize[, flags]])：同上，接收到的数据放入 buffers 中。
  - socket.send(bytes[, flags])：向 socket 发送数据，该 socket 必须与远程 socket 建立了连接。该方法通常用于在基于 TCP 协议的网络中发送数据。
  - socket.sendto(bytes, address)：向 socket 发送数据，该 socket 应该没有与远程 socket 建立连接。该方法通常用于在基于 UDP 协议的网络中发送数据。
  - socket.sendfile(file, offset=0, count=None)：发送整个文件内容，直到遇到文件的 EOF。
  - socket.shutdown(how)：关闭连接。其中 how 用于设置关闭方式，支持 SHUT_RD（关闭 socket 的输入部分，还可通过 socket 输出数据）、SHUT_WR（关闭 socket 的输出部分，可通过 socket 读取数据）和 SHUT_RDWR（全关闭。socket 既不能读取数据，也不能写入数据）。

- TCP 通信的服务器端编程的基本步骤：
  1. 服务器端先创建一个 socket 对象。
  2. 服务器端 socket 将自己绑定到指定 IP 地址和端口。
  3. 服务器端 socket 调用 listen() 方法监听网络。
  4. 程序采用循环不断调用 socket 的 accept() 方法接收来自客户端的连接。
- TCP 通信的客户端编程的基本步骤：
  1. 客户端先创建一个 socket 对象。
  2. 客户端 socket 调用 connect() 方法连接远程服务器。

- socket 提供了大量方法来发送和接收数据：
  - 发送数据：使用 send() 方法。注意，sendto() 方法用于 UDP 协议的通信。
  - 接收数据：使用 recv_xxx() 方法。

简单的服务器端程序，仅建立 socket 来监听客户端的连接，只要客户端连接进来，发送一条信息后，关闭 socket；客户端程序仅连接服务器，获得服务器发送信息。socket 的 recv() 方法在成功读取到数据之前，程序会被阻塞，无法继续执行。

```
# 服务器端程序
import socket                                          # 导入 socket 模块
s = socket.socket(); s.bind (('192.168.1.88', 30000)); s.listen()     # 创建 socket 对象，并开始监听
while True:
    c, addr = s.accept()   # 每当接收到客户端 socket 的请求时，返回对应的 socket 和远程地址
    c.send(' 您好，您收到了服务器的新年祝福！ '.encode('utf-8'))          # 发送信息给客户端
    c.close()                                                          # 关闭连接
# 客户端程序
import socket                                          # 导入 socket 模块
s = socket.socket (); s.connect({'192.168.1.88', 30000))  # 创建 socket 对象，并连接远程服务器
print('--%s--' % s.recv(1024).decode('utf-8'))  # 接收服务器端的信息
s.close()                                              # 关闭连接
```

采用多线程实现 socket 通信，可防止程序阻塞，但线程中的 recv() 方法依然会使线程阻塞。运行下面的服务器端程序，看不到任何输出信息。再运行多个客户端程序（相当于启动多个聊天室客户端登录该服务器），在任何一个客户端通过键盘输入一些内容，然后按回车键，即可在所有客户端（包括自己）的控制台中收到刚刚输入的内容，粗略地实现了一个 C/S 结构的聊天室应用。

```
# 服务器端程序
import socket; import threading
socket_list = []                                                    # 保存 socket 的列表
ss = socket.socket(); ss.bind(('192.168.1.88', 30000)); ss.listen()  # 创建 socket 对象，并监听
def read_from_client(s):
    try:  return s.recv(2048).decode('utf-8')
    except: socket_list.remove(s);    # 若有异常，表明 socket 的客户端已经关闭，删除 socket
 def server_target(s):
    try:
        while True:          # 采用循环不断地从 socket 中读取客户端发送过来的数据
            content = read_from_client(s); print(content)
            if content is None:  break
            for client_s in socket_list: client_s.send(content.encode('utf-8'))
    except e: print(e.strerror)
while True:
    s, addr = ss.accept(); socket_list.append(s)                  # 此行代码会阻塞，将一直等待连接
    threading.Thread(target=server_target, args=(s, )).start()  # 连接后启动一个线程为客户端服务
# 客户端程序
import socket; import threading
s = socket.socket(); s.connect(('192.168.1.88', 30000))   # 创建 socket 对象，并连接服务器
def read_from_server(s):
    while True: print(s.recv(2048).decode('utf-8'))
threading.Thread(target=read_from_server, args=(s, )).start()  # 启动线程读取服务器数据
while True:
    line = input('')
    if line is None or line == 'exit': break
    s.send(line.encode('utf-8'))    # 将用户的键盘输入内容写入 socket
```

通过 selectors 模块允许 socket 以非阻塞方式进行通信，selectors 相当于一个事件注册中心，只要将 socket 的所有事件注册给 selectors 管理，当 selectors 检测到 socket 中的特定事件之后，程序就调用相应的监听方法进行处理。selectors 主要支持两种事件：

- selectors.EVENT_READ：当客户端连接进来时和 socket 有数据可读时触发该事件。
- selectors.EVENT_WRITE：当 socket 将要写数据时触发该事件。

使用 selectors 实现非阻塞式编程的步骤大致如下：

1. 创建 selectors 对象。
2. 通过 selectors 对象为 socket 的 selectors.EVENT_READ 或 selectors.EVENT_WRITE 事件注册监听器函数。每当 socket 有数据需要读写时，系统负责触发所注册的监听器函数。
3. 在监听器函数中处理 socket 通信。

使用 selectors 模块实现非阻塞式通信的服务器端程序：

```
import selectors, socket
sel = selectors.DefaultSelector()          # 创建默认的 selectors 对象
# 负责监听“有数据可读”事件的函数
def read(skt, mask):
    try:
        data = skt.recv(1024)                    # 读取数据
        if data:
            for s in socket_list:  s.send(data)  # 将读取的数据发送给每个 socket
        else:                                    # 如果该 socket 已被关闭
            sel.unregister(skt)
        skt.close();                             # 关闭该 socket
        socket_list.remove(skt)                  # 从 socket_list 列表中删除该 socket
    except:                                      # 如果捕捉到异常
        sel.unregister(skt)
        skt.close();                             # 关闭该 socket
        socket_list.remove(skt)                  # 从 socket_list 列表中删除该 socket
socket_list = []
# 负责监听“客户端连接进来”事件的函数
def accept(sock, mask):
    conn, addr = sock.accept()
    socket_list.append(conn)      # 使用 socket_list 保存代表客户端的 socket
    conn.setblocking(False)
    # 使用 sel 为 conn 的 EVENT_READ 事件注册 read 监听函数
    sel.register(conn, selectors.EVENT_READ, read)
sock = socket.socket()
sock.bind(('192.168.1.88', 30000))
sock.listen()
sock.setblocking(False)           # 设置该 socket 是非阻塞的
# 使用 sel 为 sock 的 EVENT_READ 事件注册 accept 监听函数
sel.register(sock, selectors.EVENT_READ, accept)
while True:       # 采用死循环不断提取 sel 的事件
    events = sel.select()
    for key, mask in events:
        callback = key.data           # key 的 data 属性获取为该事件注册的监听函数
        callback(key.fileobj, mask)  # 调用监听函数 , key 的 fileobj 属性获取被监听的 socket 对象
```

使用 selectors 模块实现非阻塞式通信的客户端程序：

```
import selectors, socket, threading
sel = selectors.DefaultSelector()          # 创建默认的 selectors 对象
# 负责监听“有数据可读”事件的函数
def read(conn, mask):
    data = conn.recv(1024)
    if data: print(data.decode('utf-8'))
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()
s = socket.socket(); s.connect(('192.168.1.88', 30000))   # 创建 socket 对象，并连接服务器
s.setblocking(False)      # 设置该 socket 是非阻塞的
# 使用 sel 为 s 的 EVENT_READ 事件注册 read 监听函数
sel.register(s, selectors.EVENT_READ, read)
# 定义不断读取用户键盘输入的函数
def keyboard_input(s):
    while True:
        line = input('')
        if line is None or line == 'exit': break
        s.send(line.encode('utf-8'))          # 将用户的键盘输入内容写入 socket
# 采用线程不断读取用户的键盘输入
threading.Thread(target=keyboard_input, args=(s, )).start()
while True:
    events = sel.select()  # 获取事件
    for key, mask in events:
        callback = key.data          # key 的 data 属性获取为该事件注册的监听函数
        callback(key.fileobj, mask)  # 调用监听函数 , key 的 fileobj 属性获取被监听的 socket 对象
```

- socket shutdown 方法

前面的服务器端和客户端通信程序，总是以一个 bytes 对象作为通信的最小数据单位的，服务器端在处理信息时也是针对每个 bytes 进行的。在一些协议中，通信的数据单位可能需要多个 bytes 对象，在这种情况下，如果彻底关闭了 socket，则会导致程序无法再从该 socket 中读取数据。socket 的 shutdown(how) 方法可以只关闭 socket 的输出（或输入）来表示输出数据已经发送完成。how 参数支持：

- SHUT_RD：关闭 socket 的输入部分，程序还可通过该 socket 输出数据。
- SHUT_WR：关闭该 socket 的输出部分，程序还可通过该 socket 读取数据。
- SHUT_RDWR：全关闭。该 socket 既不能读取数据，也不能写入数据。

```
import socket
s = socket.socket(); s.bind(('192.168.1.88', 30000)); s.listen()      # 创建 socket 对象，并开始监听
skt, addr = s.accept()              # 接收到客户端 socket 的请求时，返回对应的 socket 和远程地址
skt.send(" 服务器的第一行数据 ".encode('utf-8'))
skt.send(" 服务器的第二行数据 ".encode('utf-8'))
skt.shutdown(socket.SHUT_WR)          # 关闭 socket 的输出，表明输出数据已经结束
while True:
    line = skt.recv(2048).decode('utf-8')   # 从 socket 读取数据
    if line is None or line == '': break
    print(line)
skt.close(); s.close()
```

建立 UDP 通信（socket 模块）

UDP（User Datagram Protocol，用户数据报协议）是一种面向非连接的协议，面向非连接指的是在正式通信前不必与对方先建立连接，不管对方状态就直接发送数据。面向非连接指的是在正式通信前不必与对方先建立连接，不管对方状态就直接发送数据。

TCP 协议可靠，传输大小无限制，但是需要连接建立时间，差错控制开销大。UDP 协议不可靠，差错控制开销较小，传输大小限制在 64 KB 以下，不需要建立连接。所以，UDP 协议适用于一次只传送少量数据、对可靠性要求不高的应用环境。

UDP 协议直接位于 IP 协议之上（IP 属于 OSI 参考模型的网络层协议，而 UDP 和 TCP 都属于传输层协议）。它的主要作用是完成网络数据流和数据报之间的转换。在信息的发送端，UDP 协议将网络数据流封装成数据报，然后将数据报发送出去；在信息的接收端，UDP 协议将数据报转换成实际数据内容。

- 创建 socket 时，可设置 type 为 SOCK_DGRAM 来指定该 socket 是基于 UDP 协议的 socket。
  通过以下两个方法来发送和接收数据：
  - socket.sendto(bytes, address)：将 bytes 数据发送到 address 地址。
  - socket.recvfrom(bufsize[, flags])：接收数据。可以同时返回 socket 中的数据和数据来源地址。

  使用 UDP 协议的 socket 实现了 C/S 结构的网络通信：

```
# 服务器端程序
import socket
DATA_LEN = 4096                                  # 定义每个数据报的大小最大为 4KB
books = (" 疯狂 Python 讲义 ", " 疯狂 Kotlin 讲义 ", " 疯狂 Android 讲义 ", " 疯狂 Swift 讲义 ")
s = socket.socket(type=socket.SOCK_DGRAM)        # 通过 type 属性指定创建基于 UDP 协议的 socket
s.bind(('192.168.1.88', 30000))                  # 将该 socket 绑定到本机的指定 IP 和端口
for i in range(1000):                            # 采用循环接收数据
    data, addr = s.recvfrom(DATA_LEN)            # 读取 s 中的数据的数据的发送地址
    print(data.decode('utf-8'))                  # 将接收到的内容转换成字符串后输出
    send_data = books[i % 4].encode('utf-8')     # 从字符串数组中取出一个元素作为发送数据
    s.sendto(send_data, addr)                    # 将数据报发送给 addr 地址
s.close()
# 客户端程序
import socket
DATA_LEN = 4096;                                 # 定义每个数据报的大小最大为 4KB
s = socket.socket(type=socket.SOCK_DGRAM)        # 通过 type 属性指定创建基于 UDP 协议的 socket
while True:                                      # 不断地读取键盘输入
    line = input('')
    if line is None or line == 'exit': break
    data = line.encode('utf-8')
    s.sendto(data, ("192.168.1.88",30000))       # 发送数据报
    data = s.recv(DATA_LEN)                      # 读取 socket 中的数据
    print(data.decode('utf-8'))
s.close()
```

- **多点广播（多播）**可以将数据报以广播方式式发送到多个客户端。

若要使用多点广播，需要将数据报发送到一个组目标地址，当数据报发出后，整个组的所有主机都能接收到该数据报。IP 多点广播（或多点发送）实现了将单一信息发送给多个接收者的广播，其思想是设置一组特殊的网络地址（范围是 224.0.0.0~239.255.255.255）作为多点广播地址，每一个多点广播地址都被看作一个组，当客户端需要发送和接收广播信息时，加入该组即可。

创建 socket 对象后，需要将该 socket 加入指定的多点广播地址中（使用 setsockopt() 方法）。如果该 socket 仅用于发送数据报，使用默认地址、随机端口即可。如果该 socket 用于接收数据报，需要绑定到指定端口；否则，发送方无法确定发送数据报的目标端口。

支持多点广播的 socket 还可设置广播信息的 TTL，用于设置数据报最多可以跨过多少个网络：

- 当 TTL 的值为 0 时，指定数据报应停留在本地主机中；
- 当 TTL 的值为 1 时，指定将数据报发送到本地局域网中（默认值）；
- 当 TTL 的值为 32 时，意味着只能将数据报发送到本站点的网络上；
- 当 TTL 的值为 64 时，意味着数据报应被保留在本地区；
- 当 TTL 的值为 128 时，意味着数据报应被保留在本大洲；
- 当 TTL 的值为 255 时，意味着数据报可被发送到所有地方；

使用 socket 进行多点广播时，所有的通信实体都是平等的，它们都将自己的数据报发送到多点广播 IP 地址，并使用 socket 接收其他人发送的广播数据报。

使用 socket 实现了一个基于广播的多人聊天室。程序只需要一个 socket、两个线程，其中 socket 既用于发送数据，也用于接收数据；主线程负责读取用户的键盘输入内容，并向 socket 发送数据，子线程则负责从 socket 中读取数据。

```
import time, socket, threading, os
SENDERIP = '192.168.1.88'      # 定义本机 IP 地址
SENDERPORT = 30000             # 定义本地端口
MYGROUP = '230.0.0.1'          # 定义本程序的多点广播 IP 地址
s = socket.socket(type=socket.SOCK_DGRAM)          # 创建基于 UDP 协议的 socket
s.bind(('0.0.0.0', SENDERPORT))                    # 将该 socket 绑定到 0.0.0.0 的虚拟 IP
s.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 64)     # 设置广播消息的 TTL
# 设置允许多点广播使用相同的端口
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# 将 socket 进入广播组
status = s.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP,
    socket.inet_aton(MYGROUP) + socket.inet_aton(SENDERIP))
# 定义从 socket 读取数据的方法
def read_socket(sock):
    while True:
        data = sock.recv(2048); print(" 信息 : ", data.decode('utf-8'))
threading.Thread(target=read_socket, args=(s, )).start()  # 以 read_socket 作为 target 启动多线程
while True:       # 采用循环不断读取键盘输入，并输出到 socket 中
    line = input('')
    if line is None or line == 'exit':
        break; os._exit(0)
    s.sendto(line.encode('utf-8'), (MYGROUP, SENDERPORT))   # 将 line 输出到 socket 中
```

发送邮件（smtplib 模块）

使用 smtplib 模块来发送邮件非常简单，大部分底层的处理都由 smtplib 进行了封装，只需要 3 步即可：

1. 连接 SMTP 服务器，并使用用户名、密码登录服务器。
2. 创建 EmailMessage 对象，该对象代表邮件本身。
3. 调用代表与 SMTP 服务器连接的对象的 sendmail() 方法发送邮件。

```
import smtplib, email.utils; from email.message import EmailMessage
smtp_server = 'smtp.qq.com'                 # 定义 SMTP 服务器地址
from_addr = 'kongyeeku@qq.com'              # 定义发件人地址
password = '123456'                         # 定义登录邮箱的密码
to_addr = 'kongyeeku@163.com'               # 定义收件人地址
conn = smtplib.SMTP(smtp_server, 25)            # 创建 SMTP 连接（普通的网络连接）
conn = smtplib.SMTP_SSL(smtp_server,465)    # 创建 SMTP 连接（基于 SSL 的网络连接）
conn.set_debuglevel(1)          # 打开了 smtplib 调试模式
conn.login(from_addr, password)                                         # 登录邮箱
msg = EmailMessage()                                                    # 创建邮件对象
first_id, second_id = email.utils.make_msgid(), email.utils.make_msgid()  # 随机生成两个图片 id
msg['subject'] = ' 一封 HTML 邮件 '                                      # 主题
msg['from'] = ' 李刚 <%s>' % from_addr                                  # 发件人名字
msg['to'] = ' 新用户 <%s>' % to_addr                                    # 收件人名字
# 设置邮件内容，
msg.set_content('<h2> 邮件内容 </h2>' +
   '<p> 您好，这是一封来自 Python 的邮件 ' +
   '<img src="cid:' + second_id[1:-1] +'"><p>' +
   ' 来自 <a href="http://c.biancheng.net">C 语言中文网 </a>', 'html', 'utf-8')
with open('E:/logo.jpg', 'rb') as f:            # 添加第一个附件
   msg.add_attachment(f.read(), maintype='image', subtype='jpeg', filename='test.png', cid=first_id)
with open('E:/fklogo.gif', 'rb') as f:          # 添加第二个附件
   msg.add_attachment(f.read(), maintype='image', subtype='gif', filename='test.gif', cid=second_id)
with open('E:/fkit.pdf', 'rb') as f:            # 添加第三个附件，邮件正文不需引用该附件，不指定 cid
   msg.add_attachment(f.read(), maintype='application', subtype='pdf', filename='test.pdf',)
conn.sendmail(from_addr, [to_addr], msg.as_string())                    # 发送邮件
conn.quit()                                                             # 退出连接
```

如果希望实现图文并茂的邮件，也就是在邮件中插入图片，则首先要给邮件添加附件（不要直接在邮件中嵌入外链的图片，很多邮箱出于安全考虑，都会禁用邮件中外链的资源）。因此，如果直接在 HTML 邮件中外链其他图片，那么该图片很有可能显示不出来。

为了给邮件添加附件，只需调用 EmailMessage 的 add_attachment() 方法即可。该方法支持很多参数，最常见的参数如下：

- maintype：指定附件的主类型。比如指定 image 代表附件是图片。
- subtype：指定附件的子类型。比如指定为 png，代表附件是 PNG 图片。
- filename：指定附件的文件名。
- cid=img：指定附件的资源 ID，邮件正文可通过资源 ID 来引用该资源。

收取邮件（poplib 模块）

使用 poplib 模块收取邮件也很简单，该模块提供了 poplib.POP3 和 poplib.POP3_SSL 两个类，分别用于连接普通的 POP 服务器和基于 SSL 的 POP 服务器。

POP3 协议也属于请求，响应式交互协议，当客户端连接到服务器之后，客户端向 POP 服务器发送请求，而 POP 服务器则对客户端生成响应数据，客户端可通过响应数据下载得到邮件内容。当下载完成后，邮件客户端可以删除或修改任意邮件，而无须与电子邮件服务器进行进一步交互。

POP3 的命令和响应数据都是基于 ASCII 文本的，并以 CR 和 LF(/r/n) 作为行结束符，响应数据包括一

个表示返回状态的符号（+/）和描述信息。请求标准格式：**命令 [参数]** CRLF；响应标准格式：+OK /[−ERR] description CRLF。POP3 协议客户端的命令和服务器端对应的响应数据如下：

- user name：向 POP 服务器发送登录的用户名。
- pass string：向 POP 服务器发送登录的密码。
- quit：退出 POP 服务器。
- stat：统计邮件服务器状态，包括邮件数和总大小。
- list [msg_no]：列出全部邮件或指定邮件。返回邮件编号和对应大小。
- retr msg_no：获取指定邮件的内容（根据邮件编号来获取，编号从 1 开始）。
- dele msg_no：删除指定邮件（根据邮件编号来删除，编号从 1 开始）。
- noop：空操作。仅用于与服务器保持连接。
- rset：用于撤销 dele 命令。

poplib 模块完全模拟了上面命令，poplib.POP3 或 poplib.POP3_SSL 为上面命令提供了相应的方法，只要依次使用上面命令即可从服务器端下载对应的邮件。使用 poplib 收取邮件可分为两步：

1. 使用 poplib.POP3 或 poplib.POP3_SSL 按 POP3 协议从服务器端下载邮件。
2. 使用 email.parser.Parser 或 email.parser.BytesParser 解析邮件内容，得到 EmailMessage 对象，从 EmailMessage 对象中读取邮件内容。

```
import poplib, os.path, mimetypes
from email.parser import BytesParser, Parser; from email.policy import default
# 输入邮件地址，密码和 POP3 服务器地址
email,password,pop3_server=('kongyeeku@qq.com','123456','pop.qq.com')
#conn = poplib.POP3(pop3_server, 110)          # 连接 POP 3 服务器
conn = poplib.POP3_SSL(pop3_server, 995)      # 连接 POP 3 服务器
conn.set_debuglevel(1)                        # 可以打开或关闭调试信息
print(conn.getwelcome().decode('utf-8'))      # 可选 : 打印 POP 3 服务器的欢迎文字
conn.user(email)                          # 输入用户名，相当于发送 POP 3 的 user 命令
conn.pass_(password)                      # 输入密码，相当于发送 POP 3 的 pass 命令
message_num, total_size = conn.stat()  # 获取邮件统计信息，相当于发送 POP 3 的 stat 命令
print(' 邮件数 : %s. 总大小 : %s' % (message_num, total_size))
resp, mails, octets = conn.list()  # 获取服务器上的邮件列表，相当于发送 POP 3 的 list 命令
print(resp, mails)                 # resp 为服务器的响应码， mails 列表保存每封邮件的编号、大小
resp, data, octets = conn.retr(len(mails)) # 获取指定邮件的内容，相当于发送 POP 3 的 retr 命令
                # 传入总长度就是获取最后一封邮件，esp 为服务器的响应码，data 为该邮件的内容
msg_data = b'\r\n'.join(data)      # 将 data 的所有数据（原本是一个字节列表）拼接在一起
msg = BytesParser(policy=default).parsebytes(msg_data) # 将字符串内容解析成邮件，必须 policy=default
print(type(msg))
print(' 发件人 :' + msg['from']+'，收件人 :' + msg['to'])
print(' 主题 :' + msg['subject'])
print(' 第一个收件人名字 :' + msg['to'].addresses[0].username)
print(' 第一个发件人名字 :' + msg['from'].addresses[0].username)
for part in msg.walk():    # walk() 方法读取 EmailMessage 的各部分
    counter = 1
    if part.get_content_maintype() == 'multipart' : continue # maintype=multipart,容器( 包含正文、附件等 )
    elif part.get_content_maintype() == 'text': print(part.get_content()) # maintype=text，邮件正文
    else :          # maintype 为其他，是附件，处理附件
        filename = part.get_filename()       # 获取附件的文件名
        if not filename:      # 如果没有文件名，程序要负责为附件生成文件名
            ext = mimetypes.guess_extension(part.get_content_type())   # 根据 contnet_type 推测后缀名
            if not ext: ext = '.bin'                                   # 推测不出后缀名，使用 .bin 作为后缀名
            filename = 'part-%03d%s' % (counter, ext)                  # 程序为附件来生成文件名
        counter += 1
        # 将附件写入的本地文件
        with open(os.path.join('.', filename), 'wb') as fp: fp.write(part.get_payload(decode=True))
conn.quit()       # 退出服务器，相当于发送 POP 3 的 quit 命令
```

Python 文档

对于实际的开发来说，有效的说明文档是非常重要的一环，否则其他人员就不能有效地使用该程序。

文档查看（pydoc 模块）

pydoc 模块可以方便地查看、生成帮助文档，该文档是 HTML 格式的，因此查看、使用起来非常方便。如下为带文档的 Python 源文件（文件名为 fkmodule.py）：

```
'''
模块的说明
'''
MY_NAME = 'C 语言中文网 '
def say_hi(name):
    '''
    定义一个打招呼的函数
    返回对指定用户打招呼的字符串
    '''
    print(" 执行 say_hi 函数 ")
    return name + ' 您好！  '
def print_rect(height, width):
    '''
    定义一个打印矩形的函数
    height – 代表矩形的高
    width – 代表矩形的宽
    '''
    print(('*' * width + '\n') * height)
class User:
    NATIONAL = 'China'
    '''
    定义一个代表用户的类
    该类包括 name、age 两个变量
    '''
    def __init__(self, name, age):
        '''
        name 初始化该用户的 name
        age 初始化该用户的 age
        '''
        self.name = name
        self.age = age
    def eat (food):
        '''
        定义用户吃东西的方法
        food – 代表用户正在吃的东西
        '''
        print('%s 正在吃 %s' % (self.name, food))
```

- **pydoc 在控制台中查看文档**

  使用 pydoc 模块在控制台中查看帮助文档的命令是：python –m pydoc 模块名。例如：在 fkmodule.py 所在当前目录下运行命令 python –m pydoc fkmodule，即可看到 fkmodule 模块的帮助信息。帮助信息按如下顺序来显示模块中的全部内容：

  - **模块的文档说明**：就是 * .py 文件顶部的注释信息，这部分信息会被提取成模块的文档说明。
  - CLASSES 部分：这部分会列出该模块所包含的全部类。
  - FUNCTIONS 部分：这部分会列出该模块所包含的全部函数。
  - DATA 部分：这部分会列出该模块所包含的全部成员变量。
  - FILE 部分：这部分会显示该模块对应的源文件。

- **pydoc 生成 HTML 文档**

  使用 pydoc 模块在控制台中查看帮助文档的命令：python –m pydoc –w 模块名。例如：在 fkmodule.py 所在当前目录下运行命令 python –m pydoc –w fkmodule，在该目录下生成了一个 fkmodule.html 文件，可使用浏览器打开该文件。

  pydoc 还可用于为指定目录生成 HTML 文档：python –m pydoc –w 目录名。为指定目录下的所有模块生成 HTML 文档。

- **pydoc 启动本地服务器来查看文档信息**

  启动本地服务器来查看文档信息，可以使用如下两个命令：

  - python –m pydoc –p 端口号：在指定端口启动 HTTP 服务器；
  - python –m pydoc –b：在任意一个未占用的端口启动 HTTP 服务器。

  命令会显示一个地址，使用浏览器打开该地址，就可以查看 Python 的所有模块的文档信息。

- **pydoc 查找模块**

  pydoc 还提供了一个 –k 选项，用于查找模块：python –m pydoc –k 被搜索模块的部分内容。例如：在 fkmodule.py 所在目录下运行命令 python –m pydoc –k fk，输出信息为 fkmodule。

文档测试（doctest 模块）

文档测试是指通过 doctest 模块运行 Python 源文件的说明文档中的测试用例，从而生成测试报告。

文档测试工具可以提取说明文档中的测试用例，其中“>>>”之后的内容表示测试用例，接下来的一行则代表该测试用例的输出结果。文档测试工具会判断测试用例的运行结果与输出结果是否一致，如果不一致就会显示错误信息。错误信息分 4 个部分：

- 第一部分：显示在哪个源文件的哪一行。
- 第二部分：Failed example，显示是哪个测试用例出错了。
- 第三部分：Expected，显示程序期望的输出结果。就是“>>> 命令”的下一行给出的期望结果。
- 第四部分：Got，显示程序实际运行产生的输出结果。只有当实际运行产生的输出结果与期望结果一致时，才表明该测试用例通过。

```
def square (x):
    '''
    一个用于计算平方的函数
    例如
    >>> square(2)
    4
    >>> square(3)
    9
    >>> square(–3)
    9
    >>> square(0)
    0
    '''
    return x * 2      # 故意写错的
class User:
    '''
    定义一个代表用户的类，该类包含如下两个属性：
    name – 代表用户的名字
    age – 代表用户的年龄
    例如
    >>> u = User('fkjava', 9)
    >>> u.name
    'fkjava'
    >>> u.age
    9
    >>> u.say('i love python')
    'fkjava 说 : i love python'
    '''
    def __init__(self, name, age):
        self.name = 'fkit'        # 故意写错的
        self.age = age
    def say(self, content):
        return self.name + ' 说 : ' + content
if __name__=='__main__':
    import doctest
    doctest.testmod()
```

Python 为文档注释提供了 doctest 模块，该模块的用法非常简单，程序只要导入该模块，并调用该模块的 testmod() 函数即可。testmod() 函数会自动提取该模块的说明文档中的测试用例，并执行这些测试用例，最终生成测试报告。如果存在没有通过的测试用例，程序就会显示有多少个测试用例没有通过；如果所有测试用例都能通过测试，则不生成任何输出结果。

Python 测试（PyUnit）

PyUnit（unittest） 是 Python 自带的单元测试框架，用于编写和运行可重复的测试。主要用于进行白盒测试和回归测试。PyUnit 可以让测试具有持久性，测试与开发同步进行，测试代码与开发代码一同发布。使用 PyUnit 具有如下好处：

- 可以使测试代码与产品代码分离。
- 针对某一个类的测试代码只需要进行较少的改动，便可以应用于另一个类的测试。
- PyUnit 开放源代码，可以进行二次开发，方便对 PyUnit 的扩展。

PyUnit 是一个简单、易用的测试框架，其具有如下特征：

- 使用断言方法判断期望值和实际值的差异，返回 bool 值。
- 测试驱动设备可使用共同的初始化变量或实例。
- 测试包结构便于组织和集成运行。

测试用例类 (TestCase 的子类)

1. **编写要测试的程序（fk_math.py）：**

程序包含两个函数，one_equation(a , b) 用于计算一元一次方程的解，two_equation(a , b , c) 用于计算二元一次方程的解

```
def one_equation(a , b):
    if a == 0: raise ValueError(" 参数错误 ")          # 如果 a = 0，则方程无法求解
    else:           # 返回方程的解
        #return –b / a       # 这是正确的结果
        return b / a          # 这里有意写错
def two_equation(a , b , c):
    if a == 0:  raise ValueError(" 参数错误 ")          # 如果 a == 0，变成一元一次方程
    elif b * b – 4 * a * c < 0: raise ValueError(" 方程在有理数范围内无解 ")  # 有理数范围内无解
    elif b * b – 4 * a * c == 0: return –b / (2 * a)     # 方程有唯一的解 , 使用数组返回方程的解
    else:                                               # 方程有两个解
        r1 = (–b + (b * b – 4 * a * c) ** 0.5) / 2 / a; r2 = (–b – (b * b – 4 * a * c) ** 0.5) / 2 / a
        return r1, r2                                   # 返回方程的两个解
```

2. **编写测试程序（test_fk_math.py）：**

文件名以 test 开头。unittest 要求单元测试类必须继承 unittest.TestCase，该类中的测试方法需要满足：① 没有返回值；② 没有任何参数；③ 以 test 开头 3 个要求：。

```
import unittest
from fk_math import *
class TestFkMath(unittest.TestCase):
    def test_one_equation(self):   # 测试一元一次方程的求解
        self.assertEqual(one_equation(5 , 9) , –1.8)                    # 断言该方程求解应该为 –1.8
        self.assertTrue(one_equation(4 , 10) == –2.5 , .00001)          # 断言该方程求解应该为 –2.5
        self.assertTrue(one_equation(4 , –27) == 27 / 4)                # 断言该方程求解应该为 27/4
        with self.assertRaises(ValueError): one_equation(0 , 9)         # 断言 a = 0 时引发 ValueError
    def test_two_equation(self):   # 测试一元二次方程的求解
        r1, r2 = two_equation(1 , –3 , 2); self.assertCountEqual((r1, r2), (1.0, 2.0), ' 求解出错 ')
        r1, r2 = two_equation(2 , –7 , 6); self.assertCountEqual((r1, r2), (1.5, 2.0), ' 求解出错 ')
        r = two_equation(1 , –4 , 4); self.assertEqual(r, 2.0, ' 求解出错 ')   # 断言只有一个解的情形
        with self.assertRaises(ValueError): two_equation(0, 9, 3)       # 断言 a == 0 时引发 ValueError
        with self.assertRaises(ValueError): two_equation(4, 2, 3)       # 断言引发 ValueError
```

unittest.TestCase 内置了大量 assertXxx 方法来执行断言

断言方法	检查条件	断言方法	检查条件
assertEqual(a, b)	a == b	assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True	assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b	assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None	assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b	assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)	assertNotIsInstance(a, b)	not isinstance(a, b)

unittest.TestCase 还提供了对异常、错误、警告和日志进行断言判断

断言方法	检查条件
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) 引发 exc 异常
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) 引发 exc 异常 且异常信息匹配 r 正则表达式
assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) 引发 warn 警告
assertWamsRegex(warn, r, fun, *args, **kwds)	fun(*args, **kwds) 引发 warn 警告 且警告信息匹配 r 正则表达式
assertLogs(logger, level)	With 语句块使用日志器生成 level 级别的日志

unittest.TestCase 还包含了完成某种特定检查的断言方法

断言方法	检查条件	断言方法	检查条件
assertAlmostEqual(a, b)	round(a−b, 7) == 0	assertNotAlmostEqual(a, b)	round(a−b, 7) != 0
assertGreater(a, b)	a > b	assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b	assertLessEqual(a, b)	a <= b
assertRegex(s, r)	r.search(s)	assertNotRegex(s, r)	not r.search(s)
assertIn(a, b)	a in b	assertNotIn(a, b)	a not in b
assertCountEqual(a, b)	a、b 两个序列包含的元素相同，不管元素出现的顺序如何		

3. **运行测试**

在编写完测试用例之后，可以使用如下两种方式来运行测试：

○ 在测试程序后增加以下代码，运行测试程序。

```
if __name__ == '__main__': unittest.main()
```

○ 在控制台通过命令 python –m unittest **测试文件** 运行测试。若不指定**测试文件**，将自动查找并运行当前目录下的所有测试用例（每个以 test 开头的 py 文件中以 test 开头的方法都是一个真正独立的测试用例）。

测试结果的第一行显示测试的测试用例数量和结果：每个测试用例用一个字符表示（有几个字符表示测试了几个测试用例）；字符表示测试结果（. 表示测试通过、F 表示测试失败、E 表示测试出错、S 表示跳过该测试）；接着是未通过的测试用例的具体信息；最后是总的测试用例数量和测试所用时间。

```
F.        # 测试了 2 个用例，第一个失败，第二个通过
======================================================================
FAIL: test_one_equation (test1.TestFkMath)      # 失败用例方法名和所属的类
----------------------------------------------------------------------
Traceback (most recent call last):
  File "E:\python\test1.py", line 5, in test_one_equation    # 失败用例所在的文件、行号和名称
    self.assertEqual(one_equation(5 , 9) , –1.8)             # 失败的语句
AssertionError: 1.8 != –1.8                                  # 失败的原因
----------------------------------------------------------------------
Ran 2 tests in 0.002s     # 进行了 2 个测试，用时 0.002 秒
FAILED (failures=1)       # 测试失败，失败数量 1
```

测试包（TestSuite）和 测试运行器（TestRunner）

测试包（TestSuite）可以组织多个测试用例，测试包还可以嵌套测试包。在使用测试包组织多个测试用例和测试包之后，程序可以使用测试运行器（TestRunner）来运行该测试包所包含的所有测试用例。

1. 再开发一个程序（hello.py）：

```
def say_hello(): return "Hello World."      # 该方法简单地返回字符串
def add(nA, nB): return nA + nB             # 计算两个整数的和
```

2. 为 hello.py 提供如下测试类 (test_hello.py)：

```
import unittest
from hello import *
class TestHello(unittest.TestCase):
    def test_say_hello(self):          # 测试 say_hello 函数
        self.assertEqual(say_hello() , "Hello World.")
    def test_add(self):                # 测试 add 函数
        self.assertEqual(add(3, 4) , 7)
        self.assertEqual(add(0, 4) , 4)
        self.assertEqual(add(–3, 0) , –3)
```

3. 将 test_fk_math.py 和 test_hello.py 文件放在同一目录，通过 TestSuite 将它们组织在一起，使用 TestRunner 来运行该测试包：

```
import unittest
from test_fk_math import TestFkMath     # 导入 test_fk_math 中的 TestFkMath 类
from test_hello import TestHello        # 导入 test_hello.py 中的 TestHello 类
test_cases = (TestHello, TestFkMath)
def whole_suite():
    loader = unittest.TestLoader()          # 创建测试加载器
    suite = unittest.TestSuite()            # 创建测试包
    for test_class in test_cases:           # 遍历所有测试类
        tests = loader.loadTestsFromTestCase(test_class)  # 从测试类中加载测试用例
        suite.addTests(tests)                             # 将测试用例添加到测试包中
    return suite
if __name__ == '__main__':
    # 创建测试运行器（TestRunner），verbosity=2 可以生成更详细的测试信息
    runner = unittest.TextTestRunner(verbosity=2)
    runner.run(whole_suite())
    # 创建测试运行器（TestRunner），将测试报告输出到文件中
    with open('fk_test_report.txt', 'a') as f:
        runner = unittest.TextTestRunner(verbosity=2, stream=f)
        runner.run(whole_suite())
```

4. 运行上面程序，可以看到生成如下测试报告：

```
test_add (test_hello.TestHello) ... ok
test_say_hello (test_hello.TestHello) ... ok
test_one_equation (test_fk_math.TestFkMath) ... FAIL
test_two_equation (test_fk_math.TestFkMath) ... ok
======================================================================
失败测试用例的具体信息
----------------------------------------------------------------------
Ran 4 tests in 0.002s
FAILED (failures=1)
```

测试固件（Test Fixture）

unittest 包含测试用例类（TestCase 的子类）、测试包（TestSuite）、测试运行器（TestRunner）和测试固件（Test Fixture）概念：

- **测试用例类**：单个的测试单元，其负责检查特定输入和对应的输出是否匹配。
- **测试包**：用于组合多个测试用例，测试包也可以嵌套测试包。
- **测试运行器**：负责组织、运行测试用例，并向用户呈现测试结果。
- **测试固件**：代表执行一个或多个测试用例所需的准备工作及相关联的准备操作（可能包括创建临时数据库、创建目录、开启服务器进程等）。

针对测试固件，unittest.TestCase 提供以下自动执行方法，程序可重写这些方法：

- setUp() 方法：用于运行每个测试用例前，初始化测试固件
- tearDown() 方法：用于运行每个测试用例后，销毁测试固件
- setUpClass() 方法：用于运行所有测试用例前，初始化测试固件
- tearDownClass() 方法：用于运行所有测试用例后，销毁测试固件

```
import unittest
from hello import *
class TestHello(unittest.TestCase):
    def test_say_hello(self):          # 测试 say_hello 函数
        self.assertEqual(say_hello() , "Hello World.")
    def test_add(self):                # 测试 add 函数
        self.assertEqual(add(3, 4) , 7)
        self.assertEqual(add(0, 4) , 4)
        self.assertEqual(add(–3, 0) , –3)
    def setUp(self):                   # 运行每个测试用例前，初始化测试固件
        print('\n==== 执行 setUp 模拟初始化固件 ====')
    def tearDown(self):                # 运行每个测试用例后，销毁测试固件
        print('\n==== 调用 tearDown 模拟销毁固件 ====')
    @classmethod
    def setUpClass(cls):               # 运行所有测试用例前，初始化测试固件
        print('\n==== 执行 setUpClass 在类级别模拟初始化固件 ====')
    @classmethod
    def tearDownClass(cls):            # 运行所有测试用例后，销毁测试固件
        print('\n==== 调用 tearDownClass 在类级别模拟销毁固件 ====')
```

跳过测试用例

在默认情况下，unittest 会自动测试每一个测试用例（以 test 开头的方法），如果希望临时跳过某个测试用例，则可以通过如下两种方式来实现：

- 使用 skipXxx 装饰器来跳过测试用例。@ unittest.skip(reason)（无条件跳过）、@unittest.skipIf(condition, reason)（当 condition 为 True 时跳过）和 @ unittest.skipUnless(condition, reason)（当 condition 为 False 时跳过）。

```
@unittest.skip(' 临时跳过 test_add')
def test_add(self):
```

- 使用 TestCase 的 skipTest() 方法来跳过测试用例。

```
def test_add(self):
    self.skipTest(' 临时跳过 test_add')
```

Python 打包（zipapp）

zipapp 模块可以将一个 Python 模块（可能包含很多个源程序）打包成一个 Python 应用（.pyz 文件），不足之处是该文件依然需要 Python 环境来执行。

- **生成可执行的 Python 档案包**

  zipapp 模块的命令行语法：python –m zipapp source [options]

  - source：要打包的 Python 源程序或目录，若为目录，会打包该目录中的所有 Python 文件。
  - options：支持如下选项：
    - –o <output>，––output=<output>：指定输出的档案包名（默认同 source，后缀为 .pyz ）
    - –p <interpreter>，––python=<interpreter>：指定 Python 解释器。
    - –m <mainfn>，––main=<mainfn>：指定 Python 程序的入口函数（默认为 __main__.py）。应该为 pkg.mod:fn 形式，其中 pkg.mod 是一个包或模块，fn 是指定模块中的函数。
    - –c，––compress：指定是否对档案包进行压缩（默认不压缩。从 Python 3.7 开始支持）。
    - ––info：该选项用于在诊断时显示档案包中的解释器。
    - –h，––help：该选项用于显示 zipapp 模块的帮助信息。

  例如：建立一个 app 目录，在目录下建立 2 个 py 文件（say_hello.py 和 app.py）

```
say_hello.py 程序：
def say_hello(name): return name + "，您好！ "
app.py 程序：
from say_hello import *
def main():
    print(' 程序开始执行 ')
    print(say_hello(' 孙悟空 '))
```

  通过控制台（app 目录的父目录中）执行命令：python –m zipapp app –o first.pyz –m "app:main"
  生成一个 first.pyz 文件，执行 first.pyz：python first.pyz，获得结果。
  若执行：python –m zipapp app –m "app:main"，未指定 –o，使用 source 参数，生成 app.pyz。

- **zipapp 创建独立应用**

  如果 Python 应用还需要使用第三方模块和包（比如需要连接 MySQL 的应用），仅打包该应用的 Python 程序是不够的。为了创建能独立启动的应用（自带依赖模块和包），需要执行两步操作：

  1. 将应用依赖的模块和包下载到应用目录中。
  2. 使用 zipapp 将应用和依赖模块一起打包成档案包。

  例如：在 app 所在目录下再创建一个 dbapp 子目录，在 dbapp 下新建 __main__.py 文件作为程序入口

```
from exec_select import *        # exec_select.py 文件需要自己添加，为数据库操作模块
query_db()                       # 执行 exec_select.py 中的 query_db() 函数
```

  按照如下步骤将 dbapp 子目录下的应用打包成独立应用：

  1. 通过控制台（app 目录中）执行命令：python –m pip install –r requirements.txt ––target dbapp。在 dbapp 目录下安装模块，requirements.txt 文件中指定要安装的模块（如 mysql–connector–python，可加入多个模块，每行定义一个模块）。
  2. 建议删除 dbapp 子目录下生成的 .dist–info 目录（若存在）。
  3. 使用 zipapp 模块执行打包操作：python –m zipapp dbapp（因为 dbapp 子目录下包含了 __main__.py 文件，该文件将会作为程序入口，因此打包时不需要指定 –m 选项），生成 dbapp.pyz 程序。因为 dbapp.pyz 中已包含 mysql–connector–python 模块，所以不再需要此模块，dbapp.pyz 也可独立运行。

Python 发布（PyInstaller）

PyInstaller 工具可以将独立应用（自包含该应用的依赖包）打包成可执行程序，这个程序可以被分发到对应平台（Windows 或 Mac OS X）的目标机器上直接运行，无须在目标机器上安装 Python 解释器环境。

- **安装 PyInstalle**

Python 默认不包含 PyInstaller 模块，需要自行安装（使用 pip 命令 pip install pyinstaller)。安装成功之后，在 Python 的安装目录下的 Scripts 目录下会增加一个 pyinstaller.exe 程序，接下来就可以使用该工具将 Python 程序生成 EXE 程序了。（PyInstaller 工具是跨平台的，可以在 Windows 平台和 Mac OS X 平台上运行。在不同的平台上使用 PyInstaller 工具的方法是一样的，支持的选项也是一样的）

- **生成可执行程序**

PyInstaller 工具的命令语法：pyinstaller 选项 Python 源文件。PyInstaller 支持的常用选项：

–h，––help	查看该模块的帮助信息
–F，–onefile	产生单个的可执行文件
–D，––onedir	产生一个目录（包含多个文件）作为可执行程序
–a，––ascii	不包含 Unicode 字符集支持
–d，––debug	产生 debug 版本的可执行文件
–w，––windowed，––noconsolc	指定程序运行时不显示命令行窗口（仅对 Windows 有效）
–c，––nowindowed，––console	指定使用命令行窗口运行程序（仅对 Windows 有效）
–o DIR，––out=DIR	指定 spec 文件的生成目录。默认使用当前目录来生成 spec 文件
–p DIR，––path=DIR	设置 Python 导入模块的路径（和设置 PYTHONPATH 环境变量的作用相似）。也可使用路径分隔符（Windows 使用分号，Linux 使用冒号）来分隔多个路径
–n NAME，––name=NAME	指定项目（产生的 spec）名字。默认使用第一个脚本的主文件名将作为 spec 的名字

创建 app 目录，目录下创建 exec_select.py（负责查询数据）和 main.py（负责创建图形用户界面来显示查询结果），目录下使用命令行工具执行命令：Pyinstaller –F –w main.py，将显示详细的生成过程。完成后，app 目录下多了一个 dist 目录和该目录下的 main.exe 文件，这就是使用 PyInstaller 工具生成的 EXE 程序。s 双击 dist 目录下 main.exe，将会看到该程序生成的输出结果。

exec_select.py 程序：

```
import mysql.connector # 导入访问 MySQL 的模块
def query_db():
    conn = conn = mysql.connector.connect(user='root', password='rootpassword',host='localhost',
port='3306', database='python', use_unicode=True);c = conn.cursor()      # 连接数据库，获取游标
   c.execute('select * from user_tb where user_id > %s', (2,))              # 执行 select 语句查询数据
   description = c.description                                  # 通过游标的 description 属性获取列信息
   rows = c.fetchall()                                          # 使用 fetchall 获取游标中的所有结果集
   c.close();conn.close()                                       # 关闭游标，关闭连接
   return description, rows
```

main.py 程序：

```
from exec_select import *; from tkinter import *
def main():
   description, rows = query_db()
   win = Tk(); win.title(' 数据库查询 ')      # 创建窗口
   for i, col in enumerate(description):     # 通过 description 获取列信息
      lb = Button(win, text=col[0], padx=50, pady=6); lb.grid(row=0, column=i)
   for i, row in enumerate(rows):            # 直接使用 for 循环查询得到的结果集
      for j in range(len(row)):
         en = Label(win, text=row[j]); en.grid(row=i+1, column=j)
   win.mainloop()
if __name__ == '__main__': main()
```

Python 数据可视化（Matplotlib 库）

Matplotlib 是一个非常优秀的 Python 2D 绘图库，只要给出符合格式的数据，通过 Matplotlib 就可以方便地制作折线图、柱状图、散点图等各种高质量的数据图。

安装 Matplotlib 包命令：pip install matplotlib；查看文档命令：python –m pydoc –p 8899

■ 折线图（plot 函数）

折线图只需调用 pyplot 子模块下的 plot() 函数加载 X 轴和 Y 轴数据：plot(x_data, y_data)；
若只传入一个列表，将作为 Y 轴数据，X 轴自动使用 0、1、2、...：plot(y_data)；
也可同时加载多条折线：plot(x_data, y_data1,x_data, y_data2)

```
import matplotlib.pyplot as plt
# Matplotlib 默认不支持中文字体，如果希望修改 Matplotlib 的默认字体，可使用 matplotlib.font_manager
子模块下的 FontProperties 类加载中文字体。
import matplotlib.font_manager as fm
my_font=fm.FontProperties(fname="C:\Windows\Fonts\simkai.ttf")        # 加载中文字体
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']    # 定义 X 轴数据
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]         # 定义 Y 轴数据 1
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]           # 定义 Y 轴数据 2
# 可多次调用 plot() 函数生成多条折线
ln1, = plt.plot(x_data, y_data1,   # 折线数据，第一个是 X 轴数据，第二个是 Y 轴数据
        color = 'red',             # 折线颜色
        linewidth = 2.0,           # 折线宽度
        linestyle = '--',          # 折线样式（支持的样式见折线样式）
        label='Java 基础           # 折线标签（此处定义了标签，legend 中就可以不用定义了）
        ')      # 折线 1
ln2, = plt.plot(x_data, y_data2, color = 'blue', linewidth = 3.0, linestyle = '-.', label='C 语言基础 ')
plt.legend(handles=[ln1, ln2], labels=['Java 基础 ','C 语言基础 '],  # 折线和对应的标签
        loc='best',      # 图例的添加位置（支持的位置见图例位置）
        prop=my_font   # 使用的字体（可修改 Matplotlib 的默认字体（方法如下），这里无需设置）
        )      # 设置图例，若 plot 中定义了 label，这里就不需要 handles 和 labels 了
plt.title('C 语言中文网的历年销量 ')                                  # 设置标题
plt.xlabel(" 年份 "); plt.ylabel(" 教程销量 ")                        # 设置坐标轴名字
# 设置 Y 轴刻度值。xticks() 和 yticks() 可设置 X 轴和 Y 轴的刻度值（允许使用文本作为刻度值）
plt.yticks([50000, 70000, 100000], [r' 挺好 ', r' 优秀 ', r' 火爆 '])
ax = plt.gca()             # gca() 可获取坐标轴信息对象，然后对坐标轴进行控制
ax.xaxis.set_ticks_position('bottom')          # 设置将 X 轴的刻度值放在底部 X 轴上
ax.yaxis.set_ticks_position('left')            # 设置将 Y 轴的刻度值放在左面 Y 轴上
ax.spines['right'].set_color('none')           # 设置右边坐标轴线的颜色（none 表示不显示）
ax.spines['top'].set_color('none')             # 设置顶部坐标轴线的颜色（none 表示不显示）
ax.spines['bottom'].set_position(('data', 70000))  # 定义底部坐标轴线的位置（放在 70000 数值处）
plt.show()                                     # 调用 show() 函数显示图形
```

- **折线样式**：-（默认，实线）、--（虚线）、:（点线）、-.（短线、点相间的虚线）。
- **图例位置**：best（自动选择最佳位置）、upper right（右上角）、upper left（左上角）、lower left（左下角）、lower right（右下角）、right（右边）、center left（左边居中）、center right（右边居中）、lower center（底部居中）、upper center（顶部居中）、center（中心）。
- **修改 Matplotlib 默认字体**：修改后可以避免每次都要设置字体，方法如下：打开 Matplotlib 配置文件（python 安装目录 \Lib\site-packages\matplotlib\mpl-data\matplotlibrc），找到“#font.family: sans-serif”，改成：“font.family: Microsoft YaHei”即可。以下示例不再设置字体。

- **创建子图**：subplot() 函数可以创建一个子图，然后程序就可以在子图上进行绘制。

  subplot(nrows, ncols, index, **kwargs) 将数据图区域分成 nrows 行 ncols 列，获得第 index 个区域。也支持直接传入一个三位数的参数，分别作为 nrows，ncols 和 index 。

```
import matplotlib.pyplot as plt; import numpy as np
plt.figure()
x_data = np.linspace(–np.pi, np.pi, 64, endpoint=True)   # 从 –pi 到 pi 之间平均取 64 个数据点
plt.subplot(2, 1, 1)       # 将整个 figure 分成两行一列，该图形放在第 1 个网格
plt.title(' 正弦曲线 ')      # 绘制正弦曲线
plt.plot(x_data, np.sin(x_data))          # np.sin(x_data) 获得 x_data 的 sin 值列表
plt.gca().spines['right'].set_color('none'); plt.gca().spines['top'].set_color('none')
plt.gca().spines['bottom'].set_position(('data', 0)); plt.gca().spines['left'].set_position(('data', 0))
plt.subplot(223)           # 将整个 figure 分成两行两列，该图形放在第 2 个网格
plt.title(' 余弦曲线 ')      # 绘制余弦曲线
plt.plot(x_data, np.cos(x_data))          # np.cos(x_data) 获得 x_data 的 cos 值列表
plt.gca().spines['right'].set_color('none'); plt.gca().spines['top'].set_color('none')
plt.gca().spines['bottom'].set_position(('data', 0)); plt.gca().spines['left'].set_position(('data', 0))
plt.subplot(224)           # 将整个 figure 分成两行两列，并该图形放在第 3 个网格
plt.title(' 正切曲线 ')      # 绘制正切曲线
plt.plot(x_data, np.tan(x_data))          # np.tan(x_data) 获得 x_data 的 tan 值列表
plt.gca().spines['right'].set_color('none'); plt.gca().spines['top'].set_color('none')
plt.gca().spines['bottom'].set_position(('data', 0)); plt.gca().spines['left'].set_position(('data', 0))
plt.show()
```

还可用 matplotlib.gridspec 的 GridSpec 函数来分隔绘图区

```
gs = matplotlib.gridspec.GridSpec(2, 3)                          # 绘图区分隔成 2 行 3 列
ax1 = plt.subplot(gs[0, :]); ax1.plot(x_data, np.sin(x_data))    # ax1 占第 1 行整行 , 绘图
ax2 = plt.subplot(gs[1, 0]); ax2.plot(x_data, np.cos(x_data))    # ax2 占第 2 行的第 1 格 , 绘图
ax3 = plt.subplot(gs[1, 1:3]); ax3.plot(x_data, np.tan(x_data))  # ax3 占第 2 行的第 2、3 格 , 绘图
```

饼图（pie 函数）

使用 Matplotlib 提供的 pie() 函数来绘制饼图。

```
import matplotlib.pyplot as plt
# 准备数据和标签
data=[0.16881,0.14966,0.07471,0.06992,0.04762,0.03541,0.02925,0.02411,0.02316,0.01409,0.36326]
labels=['Java','C','C++','Python','VB.NET','C#','PHP','JavaScript','SQL','Assembly langugage',' 其他 ']
explode = [0, 0, 0, 0.3, 0, 0, 0, 0, 0, 0, 0]          # 将第 4 个语言（Python）分离出来
colors=['red', 'pink', 'magenta','purple','orange']    # 使用自定义颜色
plt.axes(aspect='equal')                  # 将纵横坐标轴标准化，保证饼图是一个正圆，否则为椭圆
plt.xlim(0,8);plt.ylim(0,8)               # 控制 X 轴和 Y 轴的范围（用于控制饼图的圆心，半径）
# 绘图饼图
plt.pie(x = data,                                          # 绘图数据
    labels=labels,                                         # 添加编程语言标签
    explode=explode,                                       # 突出显示 Python
    colors=colors,                                         # 设置饼图的自定义填充色
    autopct='%.3f%%',                                      # 设置百分比的格式，此处保留 3 位小数
    pctdistance=0.8,                                       # 设置百分比标签与圆心的距离
    labeldistance = 1.15,                                  # 设置标签与圆心的距离
    startangle = 180,                                      # 设置饼图的初始角度
    center = (4, 4),                                       # 设置饼图的圆心
    radius = 3.5,                                          # 设置饼图的半径
    counterclock = False,                                  # 是否逆时针，这里设置为顺时针方向
    wedgeprops = {'linewidth': 1, 'edgecolor':'green'},    # 设置饼图内外边界的属性值
    textprops = {'fontsize':12, 'color':'black'},          # 设置文本标签的属性值
    frame = 1)                                             # 是否显示饼图的圆圈，此处设为显示
plt.xticks(());plt.yticks(())                              # 不显示 X 轴和 Y 轴的刻度值
plt.title('2018 年 8 月的编程语言指数排行榜 ')               # 添加图标题
plt.show()                                                 # 显示图形
```

柱状图（bar 和 barh 函数）

- **垂直柱状图**：使用 Matplotlib 提供的 bar() 函数

```
import matplotlib.pyplot as plt; import numpy as np
bar_width=0    # 准备数据
x_data = ['2012', '2013', '2014', '2015', '2016', '2017', '2018']
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
# 绘制重叠柱状图
plt.bar(x=x_data, height=y_data1, label='C 语言基础 ', color='steelblue', alpha=0.8)
plt.bar(x=x_data, height=y_data2, label='Java 基础 ', color='indianred', alpha=0.8)
# 绘制并列柱状图
bar_width=0.3; ;plt.xticks(np.arange(len(x_data))+bar_width/2, x_data)
plt.bar(x=range(len(x_data)), height=y_data1, label='C 语 言 基 础 ', color='steelblue', alpha=0.8,
width=bar_width)                    # X 轴数据为 range(len(x_data), 即 0、1、2...
plt.bar(x=np.arange(len(x_data))+bar_width, height=y_data2, label='Java 基 础 ', color='indianred',
alpha=0.8, width=bar_width)    # X 轴数据为 np.arange(len(x_data))+bar_width
# 在柱状图上显示具体数值 , ha 参数控制水平对齐方式 , va 控制垂直对齐方式
for x, y in enumerate(y_data1): plt.text(x, y, '%s' % y, ha='center', va='bottom')
for x, y in enumerate(y_data2): plt.text(x+bar_width, y, '%s' % y, ha='center', va='bottom')
plt.title("Java 与 Android 图书对比 ")      # 设置标题
plt.xlabel(" 年份 "); plt.ylabel(" 销量 ")    # 为两条坐标轴设置名称
plt.legend()                                # 显示图例
plt.show()
```

- **水平柱状图**：使用 Matplotlib 提供的 barh() 函数。barh() 的用法与 bar() 的用法基本一样，只是在调用 barh() 函数时使用 y 参数传入 Y 轴数据，使用 width 参数传入代表条柱宽度的数据。

```
import matplotlib.pyplot as plt; import numpy as np
bar_width=0    # 准备数据
x_data = ['2012', '2013', '2014', '2015', '2016', '2017', '2018']
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
# 绘制重叠柱状图
plt.barh(y=x_data, width=y_data1, label='C 语言基础 ', color='steelblue', alpha=0.8)
plt.barh(y=x_data, width=y_data2, label='Java 基础 ', color='indianred', alpha=0.8)
# 绘制并列柱状图
bar_width=0.3; plt.yticks(np.arange(len(x_data))+bar_width/2, x_data)
plt.barh(y=range(len(x_data)), width=y_data1, label='C 语 言 基 础 ', color='steelblue', alpha=0.8,
height=bar_width)
plt.barh(y=np.arange(len(x_data))+bar_width, width=y_data2, label='Java 基 础 ', color='indianred',
alpha=0.8, height=bar_width)
# 在柱状图上显示具体数值 , ha 参数控制水平对齐方式 , va 控制垂直对齐方式
for y, x in enumerate(y_data1): plt.text(x+5000, y-bar_width/2, '%s' % x, ha='center', va='bottom')
for y, x in enumerate(y_data2): plt.text(x+5000, y+bar_width/2, '%s' % x, ha='center', va='bottom')
plt.title("Java 与 Android 图书对比 ")      # 设置标题
plt.xlabel(" 年份 "); plt.ylabel(" 销量 ")    # 为两条坐标轴设置名称
plt.legend()                                # 显示图例
plt.show()
```

■ 散点图（scatter 函数）

散点图和折线图需要的数组非常相似，区别是折线图会将各数据点连接起来；而散点图则只是描绘各数据点，并不会将这些数据点连接起来。调用 Matplotlib 的 scatter() 函数来绘制散点图。

```
import matplotlib.pyplot as plt;import numpy as np
x_data = np.linspace(–np.pi, np.pi, 64, endpoint=True)   # 从 –pi 到 pi 之间平均取 64 个数据点
plt.scatter(x=x_data, y=np.sin(x_data),   # 指定 X 轴和 Y 轴数据
   c='purple',                            # 设置散点的颜色
   s=50,                                  # 设置散点半径（即散点大小）
   alpha = 0.5,                           # 设置散点透明度
   marker='p',                            # 设置散点的图形样式（支持的值见散点图形样式）
   linewidths=1,                          # 设置散点的边框线宽
   edgecolors=['green', 'yellow']         # 设置散点的边框颜色
)       # 沿着正弦曲线绘制散点图
plt.scatter(x_data[0], np.sin(x_data)[0], c='red', s=150,  alpha = 1)          # 绘制起点
plt.scatter(x_data[63], np.sin(x_data)[63], c='black', s=150,  alpha = 1)     # 绘制终点
plt.gca().spines['right'].set_color('none'); plt.gca().spines['top'].set_color('none')
plt.gca().spines['bottom'].set_position(('data', 0)); plt.gca().spines['left'].set_position(('data', 0))
plt.title(' 正弦曲线的散点图 '); plt.show()
```

○ **散点图形样式：**'.'（点标记）、','（像素标记）、'o'（圆形标记）、'v'（向下三角形标记）、'^'（向上三角形标记）、'<'（向左三角形标记）、'>'（向右三角形标记）、'1'（向下三叉标记）、'2'（向上三叉标记）、'3'（向左三叉标记）、'4'（向右三叉标记）、's'（正方形标记）、'p'（五地形标记）、'*'（星形标记）、'h'（八边形标记）、'H'（另一种八边形标记）、'+'（加号标记）、'x'（x 标记）、'D'（菱形标记）、'd'（尖菱形标记）、'|'（竖线标记）、'_'（横线标记）等值。

■ 等高线（contour() 函数和 contourf() 函数）

等高线图需要三维数据，其中 X、Y 轴数据决定坐标点，高度数据（相当于 Z 轴数据）来决定不同坐标点的高度。调用 contour() 函数绘制等高线，调用 contourf() 函数为等高线图填充颜色。

```
import matplotlib.pyplot as plt;import numpy as np
delta = 0.025
x = np.arange(–3.0, 3.0, delta)          # 生成代表 X 轴数据的列表
y = np.arange(–2.0, 2.0, delta)          # 生成代表 Y 轴数据的列表
X, Y = np.meshgrid(x, y)                 # 对 x、y 数据执行网格化
Z1 = np.exp(–X**2 – Y**2); Z2 = np.exp(–(X – 1)**2 – (Y – 1)**2)
Z = (Z1 – Z2) * 2                        # 计算 Z 轴数据（高度数据）
plt.contourf(x, y, Z, 16,                # 16 指定将等高线分为几部分
   alpha = 0.75,                         # 设置透明度
   cmap='rainbow'                        # 使用颜色映射来区分不同高度的区域
)       # 根据 X、Y、Z 为等高线图填充颜色
C = plt.contour(x, y, Z, 16,             # 16 指定将等高线分为几部分
   colors = 'black',                      # 指定等高线的颜色
   linewidths = 0.5                      # 指定等高线的线宽
)       # 根据 X、Y、Z 绘制等高线
plt.clabel(C, inline = True, fontsize = 10) # 绘制等高线数据
plt.xticks(()); plt.yticks(())           # 去除坐标轴刻度
plt.xlabel(" 纬度 "); plt.ylabel(" 经度 ")   # 设置坐标轴名称
plt.title(" 等高线图 ");plt.show()
```

3D 图形（plot_surface() 函数）

3D 图形需要的数据与等高线图基本相同：X、Y 数据决定坐标点，Z 轴数据决定 X、Y 坐标点对应的高度。与等高线图使用等高线来代表高度不同，3D 图形将会以更直观的形式来表示高度。调用 Axes3D 对象的 plot_surface() 方法来绘制 3D 图形。

```
import matplotlib.pyplot as plt;import numpy as np
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(12, 8))
ax = Axes3D(fig)
delta = 0.125
x = np.arange(–3.0, 3.0, delta)              # 生成代表 X 轴数据的列表
y = np.arange(–2.0, 2.0, delta)              # 生成代表 Y 轴数据的列表
X, Y = np.meshgrid(x, y)                     # 对 x、y 数据执行网格化
Z1 = np.exp(–X**2 – Y**2); Z2 = np.exp(–(X – 1)**2 – (Y – 1)**2)
Z = (Z1 – Z2) * 2                            # 计算 Z 轴数据（高度数据）
ax.plot_surface(X, Y, Z,
    rstride=1,                               # rstride（row）指定行的跨度
    cstride=1,                               # cstride(column) 指定列的跨度
    cmap=plt.get_cmap('rainbow')             # 设置颜色映射
)                                 # 根据 X, Y, Z 绘制 3D 图形
ax.set_zlim(–2, 2)          # 设置 Z 轴范围
plt.title("3D 图 "); plt.show()
```

Python 数据可视化（Pygal 模块）

Pygal 是一个简单易用的数据图库，以面向对象的方式来创建各种数据图。而且可以方便地生成各种格式的数据图（包括 PNG、SVG 等），也可以生成 XML etree、HTML 表格（需要安装其他包）。

安装 Pygal 包命令：pip install pygal；查看文档命令：python –m pydoc –p 8899

Pygal 数据图入门

- **生成数据图：**Pygal 使用面向对象的方式来生成数据图。生成数据图的步骤大致如下：
  1. 创建 Pygal 数据图对象。不同的数据图提供了不同的类（如：柱状图 pygal.Bar 类，饼图 pygal.Pie 类，折线图 pygal.Line 类等）。
  2. 调用数据图对象的 add() 方法添加数据。
  3. 调用 Config 对象的属性配置数据图。
  4. 调用数据图对象的 render_to_xxx() 方法将数据图渲染到指定的输出节点（此处的输出节点可以是 PNG 图片、SVG 文件，也可以是其他节点）。
- **配置数据图：**配置 Pygal 数据图使用 BaseConfig、CommonConfig、Config、SerieConfig 等配置类（不同的数据图，支持大量对应的配置，请参见 http://localhost:8899/pygal.config.html 进行设置和测试）。

```
import pygal
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']          # 定义 X 轴数据
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]               # 定义 Y 轴数据 1
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]                 # 定义 Y 轴数据 2
bar = pygal.Bar()                                  # 创建 pygal.Bar 对象（柱状图）
bar.add('C 语言基础 ', y_data1)                    # 添加数据 1
bar.add('Python 语言基础 ', y_data2)               # 添加数据 2
bar.x_labels = x_data                              # 设置 X 轴的刻度值
bar.title = ' 编程教程的历年销量 '                 # 设置图形标题
bar.x_title = ' 年份 ';bar.y_title = ' 销量 '      # 设置 X、Y 轴的标题
bar.x_label_rotation = 45                          # 设置 X 轴的刻度值旋转 45 度
bar.legend_at_bottom = True                        # 设置将图例放在底部
bar.margin = 35                                    # 设置数据图四周的页边距，
                # 也可通过 margin_bottom、margin_left、margin_right、margin_top 单独设置页边距
bar.show_y_guides=False                            # 隐藏 X 轴上的网格线
bar.show_x_guides=True                             # 显示 X 轴上的网格线
bar.render_to_file('fk_books.svg')                 # 指定将数据图输出到 SVG 文件中
```

Pygal 常见数据图

Pygal 的设计很好，不管是创建哪种数据图，创建方式基本是一样的：首先创建对应的数据图对象，然后添加数据，最后对数据图进行配置。Pygal 支持的常用数据图对应的类如下：

- ○ 折线图　　pygal.Line 类
- ○ 水平折线图　　pygal.HorizontalLine 类
- ○ 叠加折线图　　pygal.StackedLine 类
- ○ 柱状图　　pygal.Bar 类
- ○ 水平柱状图　　pygal.HorizontalBar 类
- ○ 叠加柱状图　　pygal.StackedBar 类
- ○ 饼图　　pygal.Pie
- ○ 点图　　pygal.Dot 类
- ○ 仪表图　　pygal.Gauge 类
- ○ 雷达图　　pygal.Rader 类

- 折线图

```
import pygal
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
line = pygal.Line()          # 创建 pygal.Line 对象（折线图）
line.add('C 语言教程 ', y_data1); line.add('Python 教程 ', y_data2)          # 添加 2 组折线的数据
line.x_labels = x_data                                                      # 设置 X 轴数据
line.y_labels = [20000, 40000, 60000, 80000, 100000]                        # 重新设置 Y 轴的刻度值
line.title,line.x_title,line.y_title = (' 编程教程的历年销量 ',' 年份 ',' 销量 ')  # 设置图形、X、Y 轴的标题
line.legend_at_bottom = True                                                # 设置将图例放在底部
line.render_to_file('fk_books.svg')
```

- 水平折线图

```
import pygal
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
horizontal_line = pygal.HorizontalLine()          # 创建 pygal.HorizontalLine 对象（水平折线图）
horizontal_line.add('C 语言教程 ', y_data1); horizontal_line.add('Python 教程 ', y_data2)
horizontal_line.x_labels,horizontal_line.y_labels = (x_data,[20000, 40000, 60000, 80000, 100000])
horizontal_line.title,horizontal_line.x_title,horizontal_line.y_title = (' 编程教程的历年销量 ',' 年份 ',' 销量 ')
horizontal_line.legend_at_bottom = True; horizontal_line.render_to_file('fk_books.svg')
```

- 叠加折线图

```
import pygal
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
stacked_line = pygal.StackedLine()          # 创建 pygal.StackedLine 对象（叠加折线图）
stacked_line.add('C 语言教程 ', y_data1); stacked_line.add('Python 教程 ', y_data2)
stacked_line.x_labels,stacked_line.y_labels = (x_data,[20000, 40000, 60000, 80000, 100000])
stacked_line.title,stacked_line.x_title,stacked_line.y_title = (' 编程教程的历年销量 ',' 年份 ',' 销量 ')
stacked_line.legend_at_bottom = True; stacked_line.render_to_file('fk_books.svg')
```

- 柱状图（示例见数据图入门）
- 水平柱状图

```
import pygal
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
horizontal_bar = pygal.HorizontalBar()          # 创建 pygal.HorizontalBar 对象（水平柱状图）
horizontal_bar.add('C 语言教程 ', y_data1); horizontal_bar.add('Python 教程 ', y_data2)
horizontal_bar.x_labels,horizontal_bar.y_labels = (x_data,[20000, 40000, 60000, 80000, 100000])
horizontal_bar.title,horizontal_bar.x_title,horizontal_bar.y_title = (' 编程教程的历年销量 ',' 年份 ',' 销量 ')
horizontal_bar.legend_at_bottom = True; horizontal_bar.render_to_file('fk_books.svg')
```

- 叠加柱状图

```
import pygal
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
stacked_bar = pygal.StackedBar()          # 创建 pygal.StackedBar 对象（叠加柱状图）
stacked_bar.add('C 语言教程 ', y_data1); stacked_bar.add('Python 教程 ', y_data2)
stacked_bar.x_labels,stacked_bar.y_labels = (x_data,[20000, 40000, 60000, 80000, 100000])
stacked_bar.title,stacked_bar.x_title,stacked_bar.y_title = (' 编程教程的历年销量 ',' 年份 ',' 销量 ')
stacked_bar.legend_at_bottom = True; stacked_bar.render_to_file('fk_books.svg')
```

- 饼图

```
import pygal
data = [0.16881, 0.14966, 0.07471, 0.06992,0.04762, 0.03541, 0.02925, 0.02411, 0.02316, 0.01409, 0.36326]
labels = ['Java', 'C', 'C++', 'Python', 'VB.NET', 'C#', 'PHP', 'JavaScript', 'SQL', 'Assembly langugage', ' 其他 ']
pie = pygal.Pie()                    # 创建 pygal.Pie 对象（饼图）
for i, per in enumerate(data):  pie.add(labels[i], per)        # 采用循环为饼图添加数据
pie.title = '2018 年 8 月编程语言 '                                  # 设置标题
pie.inner_radius = 0.4                                              # 设置内圈的半径长度（饼图特有属性）
pie.half_pie = True                                                 # 创建半圆数据图（饼图特有属性）
pie.legend_at_bottom = True; pie.render_to_file('fk_books.svg')
```

- 点图

```
import pygal
x_data = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']
y_data1 = [58000, 60200, 63000, 71000, 84000, 90500, 107000]
y_data2 = [52000, 54200, 51500,58300, 56800, 59500, 62700]
dot = pygal.Dot()                    # 创建 pygal.Dot 对象（点图）
dot.add('C 语言教程 ', y_data1); dot.add('Python 教程 ', y_data2)
dot.x_labels,dot.y_labels = (x_data,['C 语言教程 ', 'Python 教程 '])
dot.title,dot.x_title = (' 编程教程的历年销量 ',' 年份 ')
dot.dots_size,dot.y_label_rotation=(5,45)          # 设置点大小和 Y 轴刻度值的旋转角度
dot.legend_at_bottom = True; dot.render_to_file('fk_books.svg')
```

- 仪表图

```
import pygal
data = [0.16881, 0.14966, 0.07471, 0.06992,0.04762, 0.03541, 0.02925, 0.02411, 0.02316, 0.01409, 0.36326]
labels = ['Java', 'C', 'C++', 'Python', 'VB.NET', 'C#', 'PHP', 'JavaScript', 'SQL', 'Assembly langugage', ' 其他 ']
gauge = pygal.Gauge()                # 创建 pygal.Gauge 对象（仪表图）
gauge.range = [0, 1]                 # 设置仪表图的最小值和最大值（仪表图特有属性）
for i, per in enumerate(data):  gauge.add(labels[i], per)    # 采用循环为仪表图添加数据
gauge.title = '2018 年 8 月编程语言 '                                # 设置标题
gauge.legend_at_bottom = True; gauge.render_to_file('fk_books.svg')
```

- 雷达图

```
import pygal
labels,data = (['Java', 'C', 'C++', 'Python', 'C#', 'PHP'],[[5, 4.0, 5, 5, 5],[4.8, 2.8, 4.8, 4.8, 4.9],[4.5, 2.9,
4.6, 4.0, 4.9],[4.0, 4.8, 4.9, 4.0, 5],[3.0, 4.2, 2.3, 3.5, 2],[4.8, 4.3, 3.9, 3.0, 4.5]])
rader = pygal.Radar()                # 创建 pygal.Radar 对象（雷达图）
for i, per in enumerate(labels): rader.add(labels[i], data[i])          # 采用循环为雷达图添加数据
rader.x_labels = [' 平台健壮性 ', ' 语法易用性 ', ' 社区活跃度 ', ' 市场份额 ', ' 未来趋势 ']
rader.title = ' 编程语言对比图 '
rader.dots_size = 8       # 控制各数据点的大小
rader.legend_at_bottom = True;rader.render_to_file('fk_books.svg')
```

Python 读取数据

读取 csv 文件

csv 文件是一种以文本存储的表格数据（Excel 可读写 csv 文件）。csv 文件的每行代表一行数据，每行数据中每个单元格内的数据以逗号隔开。使用 csv 模块读取 csv 文件也非常简单：

1. 创建 csv 模块的读取器。
2. 循环调用 csv 读取器的 next() 方法逐行读取 csv 文件内容。next() 返回一个 list 列表代表一行数据，list 列表的每个元素代表一个单元格数据。

使用 Matplotlib 绘制广州 2017 年 7 月最高气温和最低气温折线图

```
import csv; from datetime import datetime; from matplotlib import pyplot as plt
with open('guangzhou–2017.csv') as f:   # 打开 csv 文件
    reader = csv.reader(f)              # 创建 cvs 文件读取器
    header_row = next(reader)           # 读取第一行，这行是表头数据。
    start_date,end_date = (datetime(2017, 6, 30),datetime(2017, 8, 1))     # 定义读取起始和结束日期
    dates, highs, lows = [], [], []    # 定义 3 个 list 列表（日期，最高温度，最低温度）作为展示的数据
    for row in reader:
        d = datetime.strptime(row[0], '%Y–%m–%d')          # 将第一列的值格式化为日期
        if start_date < d < end_date:                      # 只展示 2017 年 7 月的数据
            dates.append(d);  highs.append(int(row[1])); lows.append(int(row[2]))
fig = plt.figure(dpi=128, figsize=(12, 9))  # 配置图形，绘制最高温度，最低温度折线
plt.plot(dates, highs, c='red', label=' 最高气温 ',alpha=0.5, linewidth = 2.0, linestyle = '–', marker='v')
plt.plot(dates, lows, c='blue', label=' 最低气温 ', alpha=0.5, linewidth = 3.0, linestyle = '–.', marker='o')
plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)   # 为两个数据的绘图区域填充颜色
plt.title(" 广州 2017 年 7 月最高气温和最低气温 ")          # 设置标题
plt.xlabel(" 日期 ");plt.ylabel(" 气温（℃）")              # 为两条坐标轴设置名称
fig.autofmt_xdate()                                        # 该方法绘制斜着的日期标签
plt.legend()                                               # 显示图例
ax = plt.gca()
ax.spines['right'].set_color('none')        # 设置右边坐标轴线的颜色（设置为 none 表示不显示）
ax.spines['top'].set_color('none')          # 设置顶部坐标轴线的颜色（设置为 none 表示不显示）
plt.show()
```

使用 pygal 生成绘制广州 2017 年天气分布图

```
import csv,pygal
with open('guangzhou–2017.csv') as f:              # 打开 csv 文件
    reader = csv.reader(f)                         # 创建 cvs 文件读取器
    header_row = next(reader)                      # 读取第一行，这行是表头数据。
    shades, sunnys, cloudys, rainys = 0, 0, 0, 0   # 准备展示的数据
    for row in reader:
        if ' 阴 ' in row[3]: shades += 1
        elif ' 晴 ' in row[3]: sunnys += 1
        elif ' 云 ' in row[3]: cloudys += 1
        elif ' 雨 ' in row[3]: rainys += 1
        else: print(rows[3])
pie = pygal.Pie()          # 创建 pygal.Pie 对象（饼图）
pie.add(" 阴 ", shades);pie.add(" 晴 ", sunnys);pie.add(" 多云 ", cloudys);pie.add(" 雨 ", rainys)
pie.title = '2017 年广州天气汇总 '
pie.legend_at_bottom = True; pie.render_to_file('guangzhou_weather.svg')
```

读取 JSON 文件

Python 的 json 模块读取 JSON 数据非常简单，只要使用 load() 函数加载 JSON 数据即可。JSON 格式的数据通常会被转换为 list 列表或 dict 字典。

使用 Matplotlib 生成中国、美国、日本、俄罗斯、加拿大这 5 个国家的 GDP 数据柱状图：

```
import json;from matplotlib import pyplot as plt;import numpy as np
with open('gdp_json.json') as f:gpd_list = json.load(f)       # 读取 JSON 格式的 GDP 数据
country_gdps = [{}, {}, {}, {}, {}]                           # 使用 list 列表依次保存 5 个国家的 GDP 值
country_codes = ['CHN', 'USA', 'JPN', 'RUS', 'CAN']
for gpd_dict in gpd_list:            # 遍历列表的每个元素，每个元素是一个 GDP 数据项
    for i, country_code in enumerate(country_codes):
        if gpd_dict['Country Code'] == country_code:          # 只读取指定国家的数据
            year = gpd_dict['Year']
            if 2017 > year > 2000:country_gdps[i][year] = gpd_dict['Value']     # 只读取 2001 年到 2016
country_gdp_list = [[], [], [], [], []]                       # 使用 list 列表依次保存 5 个国家的 GDP 值
x_data = range(2001, 2017)                                    # 构建时间数据
for i in range(len(country_gdp_list)):
    for year in x_data:country_gdp_list[i].append(country_gdps[i][year] / 1e8)   # 除以 1e8，单位变成亿
bar_width=0.15
fig = plt.figure(dpi=128, figsize=(15, 8))
colors = ['indianred', 'steelblue', 'gold', 'lightpink', 'seagreen']
countries = [' 中国 ', ' 美国 ', ' 日本 ', ' 俄罗斯 ', ' 加拿大 ']            # 定义国家名称列表
for i in range(len(colors)):          # 采用循环绘制 5 组柱状图
    plt.bar(x=np.arange(len(x_data))+bar_width*i, height=country_gdp_list[i],label=countries[i],
color=colors[i], alpha=0.8, width=bar_width)       # 使用自定义 X 坐标将数据分开
    if i < 2:          # 仅为中国、美国的条柱上绘制 GDP 数值
        for x, y in enumerate(country_gdp_list[i]): plt.text(x, y + 100, '%.0f' % y, ha='center', va='bottom')
plt.xticks(np.arange(len(x_data))+bar_width*2, x_data)    # 为 X 轴设置刻度值
plt.title("2001 到 2016 年各国 GDP 对比 ")                  # 设置标题
plt.xlabel(" 年份 ");plt.ylabel("GDP( 亿美元 )")            # 为两条坐标轴设置名称
plt.legend();plt.show()
```

使用 Pygal 生成美国、日本、俄罗斯、加拿大的人均 GDP 数据。

```
import json,import pygal
with open('gdp_json.json') as f:gpd_list = json.load(f)                    # 读取 GDP 数据
with open(population-figures-by-country.json) as f:pop_list = json.load(f)# 读取人口数据
country_mean_gdps = [{}, {}, {}, {}]        # 使用 list 列表依次保存 4 个国家的人均 GDP 值
country_codes = ['USA', 'JPN', 'RUS', 'CAN']
for gpd_dict in gpd_list:            # 遍历列表的每个元素，每个元素是一个 GDP 数据项
    for i, country_code in enumerate(country_codes):
        if gpd_dict['Country Code'] == country_code:          # 只读取指定国家的数据
            year = gpd_dict['Year']
            if 2017 > year > 2000:    # 只读取 2001 年到 2016
                for pop_dict in pop_list:
                    if pop_dict['Country_Code'] == country_code:          # 获取指定国家的人口数据
                        country_mean_gdps[i][year] = round(gpd_dict['Value']/ pop_dict['Population_in_%d' % year])
country_mean_gdp_list = [[], [], [], []]       # 使用 list 列表依次保存 4 个国家的人均 GDP 值
x_data = range(2001, 2017)                     # 构建时间数据
for i in range(len(country_mean_gdp_list)):
    for year in x_data:country_mean_gdp_list[i].append(country_mean_gdps[i][year])
countries = [' 美国 ', ' 日本 ', ' 俄罗斯 ', ' 加拿大 ']  # 定义国家名称列表
bar = pygal.Bar()          # 创建 pygal.Bar 对象（柱状图）
for i in range(len(countries)):bar.add(countries[i], country_mean_gdp_list[i])        # 添加条柱的数据
bar.x_labels = x_data    # 设置 X 轴的刻度值
bar.title,bar.x_title,bar.y_title =('2001 到 2016 年各国人均 GDP 对比 ',' 年份 ',' 人均 GDP( 美元 )')
bar.x_label_rotation,bar.width=(45,1100)          # 设置 X 轴的刻度值旋转 45 度，柱状图宽度 1100
bar.legend_at_bottom = True;bar.render_to_file('mean_gdp.svg')
```

读取网络数据（request 库和 re 模块）

Python 网络支持库 urllib 下的 request 模块可以方便地向远程发送 HTTP 请求，获取服务器响应。本程序的思路是使用 urllib.request 向 lishi.tianqi.com 发送请求，获取该网站的响应，然后使用 re 模块来解析服务器响应，从中提取天气数据。

```
from datetime import datetime, timedelta; from matplotlib import pyplot as plt
import re; from urllib.request import *
def get_html(city, year, month):            # 定义函数读取 lishi.tianqi.com 的数据
    request = Request('http://lishi.tianqi.com/' + city + '/' + str(year) + str(month) + '.html')    # 创建请求
    request.add_header('User–Agent', 'Mozilla/5.0 (Windows NT 10.0; WOW64)AppleWebKit/537.36
            (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36')        # 添加请求头
    response = urlopen(request)                                            # 获取服务器响应
    return response.read().decode('gbk')
dates, highs, lows = [], [], []        # 定义 3 个 list 列表作为展示的数据
city,year,months = ('shanghai','2018',['01', '02', '03', '04', '05', '06', '07','08', '09', '10', '11', '12'])
prev_day = datetime(2017, 12, 31)
for month in months:     # 循环读取每个月的天气数据
    html = get_html(city, year, month)
    text = "".join(html.split())        # 将 html 响应拼起来
    patten = re.compile('<divclass="tqtongji2">(.*?)</div><divstyle="clear:both">')
    table = re.findall(patten, text)  # 定义正则表达式，获得天气信息
    patten1 = re.compile('<ul>(.*?)</ul>')
    uls = re.findall(patten1, table[0])
    for ul in uls:
        patten2 = re.compile('<li>(.*?)</li>')          # 定义解析天气信息的正则表达式
        lis = re.findall(patten2, ul)
        d_str = re.findall('>(.*?)</a>', lis[0])[0]        # 解析得到日期数据
        try:
            cur_day = datetime.strptime(d_str, '%Y–%m–%d')          # 将日期字符串格式化为日期
            high = int(lis[1])                                      # 解析得到最高气温
            low = int(lis[2])                                       # 解析最低气温
        except ValueError: print(cur_day, ' 数据出现错误 ')
        else:
            diff = cur_day – prev_day          # 计算前、后两天数据的时间差
            if diff != timedelta(days=1):        # 如果前、后两天数据的时间差不是相差一天，数据有问题
                print('%s 之前少了 %d 天的数据 ' % (cur_day, diff.days – 1))
            dates.append(cur_day); highs.append(high); lows.append(low)
            prev_day = cur_day
fig = plt.figure(dpi=128, figsize=(12, 9))                                 # 配置图形
plt.plot(dates, highs, c='red', label=' 最高气温 ', alpha=0.5, linewidth = 2.0)# 绘制最高气温的折线
plt.plot(dates, lows, c='blue', label=' 最低气温 ', alpha=0.5, linewidth = 2.0)# 绘制最低气温的折线
plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)          # 填充颜色
plt.title(" 上海 %s 年最高气温和最低气温 " % year)                          # 设置标题
fig.autofmt_xdate()                                                        # 绘制斜着的日期标签
plt.xlabel(" 日期 "); plt.ylabel(" 气温（℃）")                             # 为两条坐标轴设置名称
plt.legend()                                                               # 显示图例
ax = plt.gca()
ax.spines['right'].set_color('none')        # 设置右边坐标轴线的颜色（设置为 none 表示不显示）
ax.spines['top'].set_color('none')          # 设置顶部坐标轴线的颜色（设置为 none 表示不显示）
plt.show()
```

Python 爬虫框架（Scrapy）

使用 Python 内置的 urllib 和 re 模块实现网络爬虫的核心工作：

- **向 URL 发送请求，获取服务器响应内容。**这是所有网络爬虫都需要做的通用工作。通用工作由爬虫框架来实现，可以提供更稳定的性能，开发效率更高。
- **提取页面中我们感兴趣的信息。**这不是通用的！每个项目感兴趣的信息都可能有所不同，使用正则表达式提取信息是非常低效的，使用 XPath 提取信息的效率要高得多。
- **识别响应页面中的链接信息。**使用正则表达式效率太低，使用 XPath 会更高效。
- **多线程管理：**这是通用的，应该由框架来完成。

Scrapy 是一个专业的、高效的爬虫框架，它使用专业的 Twisted 包（基于事件驱动的网络引擎包）高效地处理网络通信，使用 lxml（专业的 XML 处理包）、cssselect 高效地提取 HTML 页面的有效信息，同时它也提供了有效的线程管理。安装 Scrapy：pip install scrapy，安装后，Scrapy 位于 lib\site-packages 下。

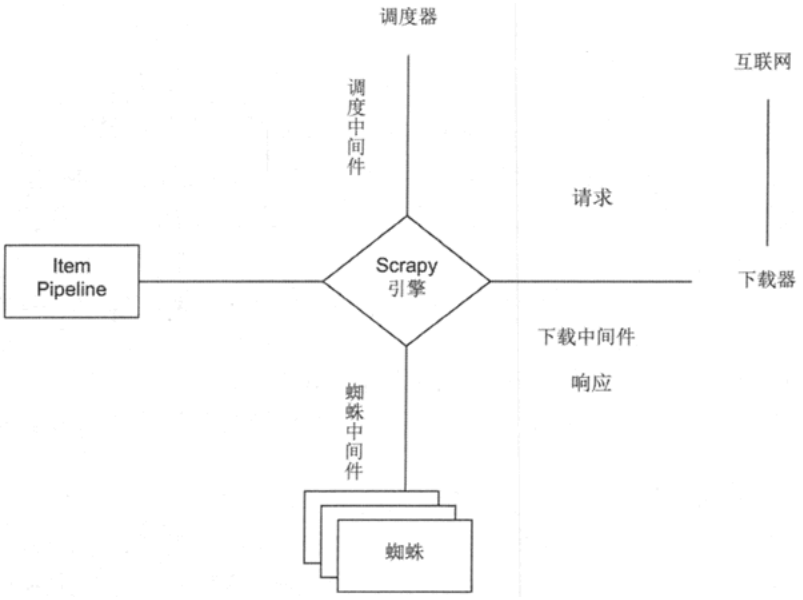
Scrapy 项目创建

在使用 Scrapy 开发爬虫时，通常需要创建一个 Scrapy 项目：scrapy startproject NewsSpider。即创建一个 NewsSpider 的爬虫项目，项目的文件结构如下：

```
NewsSpider
    scrapy.cfg                项目的总配置文件，通常无须修改。
    NewsSpider                项目的 Python 模块，程序将从此处导入 Python 代码。
        item.py               定义项目用到的 Item 类（一个 DTO，数据传输对象）。
        middlewares.py
        pipelines.py          项目的管道文件，负责处理爬取到的信息。
        setting.py            项目的配置文件，进行项目相关配置。
        spiders               存放项目所需的蜘蛛，蜘蛛负责抓取项目感兴趣的信息。
            __init__.py
            __pycache__
        __pycache__
```

为了更好地理解 Scrapy 项目中各组件的作用，给出 Scrapy 概览图如下。Scrapy 包含如下核心组件：

- **调度器：**该组件由 Scrapy 框架实现，负责调用下载中间件从网络上下载资源。
- **下载器：**该组件由 Scrapy 框架实现，负责从网络上下载数据，下载得到的数据会由 Scrapy 引擎自动交给蜘蛛。
- **蜘蛛：**该组件由开发者实现，负责从下载数据中提取有效信息。提取到的信息会由 Scrapy 引擎以 Item 对象的形式转交给 Pipeline。
- **Pipeline：**该组件由开发者实现，接收到 Item 对象（包含蜘蛛提取的信息）后，将这些信息写入文件或数据库中。



scrapy 提供了 startproject（创建项目）、fetch（从指定 URL 获取响应）、genspider（生成蜘蛛）、shell（启动交互式控制台）、version（查看 Scrapy 版本）等常用的子命令。可以直接输入 scrapy 来查看该命令所支持的全部子命令。

Scrapy 调试工具（shell）

以获得 http://news.baidu.com/guoji 页面的“即时新闻”为例：

- **使用 shell 抓取页面**

  Scrapy 提供的 shell 调试工具来抓取该页面中的信息。使用 shell 命令抓取 http://news.baidu.com/guoji：

  ① scrapy shell http://news.baidu.com/guoji

  或 ② scrapy shell –s USER_AGENT='Chrome/63.0.3239.132' http://news.baidu.com/guoji

  » **抓取错误**：当响应中出现 403 错误（response <403 http://news.baidu.com/guoji>），说明网站开启了“防爬虫”，需要在 shell 命令中添加 User-Agent 值，让 Scrapy 伪装成浏览器（User-Agent 值可从“调试控制台”的 Network 中获得），如②所示。

  » **抓取正确**：[s] request <GET http://news.baidu.com/guoji>

  [s] response <200 http://news.baidu.com/guoji>

- **使用 XPath 提取信息**

  查看页面源代码，通过 response 的 xpath() 方法来获取 XPath 匹配的节点。

```
<div class="hd"><h3>即时新闻<span class="dec">INSTANT NEWS</span></h3></div>
<div class="bd">
<div id="instant-news">
<ul class="ulist mix-ulist">
<li><a href="http://baijiahao.baidu.com/s?id=1643072551967809502" class="title" target="_blank" mon="col=
<li><a href="http://baijiahao.baidu.com/s?id=1643072524532090148" class="title" target="_blank" mon="col=
<li><a href="http://baijiahao.baidu.com/s?id=1643072409604886635" class="title" target="_blank" mon="col=
<li><a href="http://baijiahao.baidu.com/s?id=1643072384585229721" class="title" target="_blank" mon="col=
<li><a href="http://baijiahao.baidu.com/s?id=1643072314613981586" class="title" target="_blank" mon="col=
<li><a href="http://baijiahao.baidu.com/s?id=1643072266818200589" class="title" target="_blank" mon="col=
<li><a href="http://baijiahao.baidu.com/s?id=1643071564208373754" class="title" target="_blank" mon="col=
<li><a href="http://baijiahao.baidu.com/s?id=1643072025676611422" class="title" target="_blank" mon="col=
</ul>
</div>
</div>
</div>
```

response.xpath ('//div[@id="instant-news"]/ul/li/a/text()').extract()

```
[' 中国留学生携防弹衣入境美国 被注销签证并遣返 ',
 ' 普京强调推进叙利亚问题政治解决进程 ',
 ' 巴西总统发言人：欢迎外国援助亚马孙灭火 不反对与法 ..',
 ' 普京陪埃尔多安看航展 当场掏钱请吃冰激凌 ',
 ' 劫匪西装革履假扮顾客，持枪洗劫珠宝店，掳走近 300..',
 ' 人民日报钟声：美国贸易霸凌将世界经济拖入险境 ',
 ' 警方确认 TMZ 发布的音频就是考辛斯案报案人提供的音 ..',
 ' 伊朗总统：美国重返伊核协议是伊美对话的前提 ']
```

Scrapy 爬虫项目开发过程

基于 Scrapy 项目开发爬虫大致需要如下几个步骤：

1. **定义 Item 类**。该类仅仅用于定义项目需要爬取的 N 个属性。

```
# 项目目录 \NewsSpider\items.py
import scrapy
class NewsspiderItem(scrapy.Item):
    title=scrapy.Field()    # 文章标题
    url=scrapy.Field()      # 文章链接
```

2. **编写 Spider 类**。应该将该 Spider 类文件放在 spiders 目录下。这一步是爬虫开发的关键，需要使用 XPath 或 CSS 选择器来提取 HTML 页面中感兴趣的信息。Spider 提供了 scrapy genspider 命令用来创建 Spider。（项目目录中执行）scrapy genspider instant_position "news.baidu.com/guoji"，创建 instant_position 文件，首页地址：news.baidu.com/guoji。

```
# 项目目录 \NewsSpider\spiders\instant_position.py
import scrapy
from NewsSpider.items import NewsspiderItem
class InstantPositionSpider(scrapy.Spider):
    name = 'instant_position'                          # 定义该 Spider 的名字
    allowed_domains = ['baidu.com']                    # 定义该 Spider 允许爬取的域名
    start_urls = ['http://news.baidu.com/guoji/']      # 定义该 Spider 爬取的首页列表
    def parse(self, response):
        # 遍历页面上所有 //div[@id="instant-news"]/ul/li 节点
        for instant_news in response.xpath('//div[@id="instant-news"]/ul/li'):
            item = NewsspiderItem()
            item['title'] = instant_news.xpath('./a/text()').extract_first()  # 获得标题
            item['url'] = instant_news.xpath('./a/@href').extract_first()  # 获得地址
            yield item          # 将 item 返回给 Scrapy 引擎
        # 若存在下一页
        new_links = response.xpath('// 下一页的节点 /@href').extract()
        if new_links and len(new_links) > 0:
            new_link = new_links[0]             # 获取下一页的链接
            yield scrapy.Request(" 下一页地址前缀 " + new_link, callback=self.parse) # 请求下一页
```

3. **编写 pipelines.py 文件**。该文件负责将所爬取的数据写入文件或数据库中。

```
# 项目目录 \NewsSpider\pipelines.py
class NewsspiderPipeline(object):
    def process_item(self, item, spider):
        print(" 标题 :" , item['title'])
        print("URL:" , item['url'])
```

4. **修改 settings.py 文件**。如增加 User-Agent 头。

```
# 项目目录 \NewsSpider\settings.py
BOT_NAME = 'NewsSpider'
SPIDER_MODULES = ['NewsSpider.spiders']
NEWSPIDER_MODULE = 'NewsSpider.spiders'
ROBOTSTXT_OBEY = True
# 配置默认的请求头
DEFAULT_REQUEST_HEADERS = {
    "User-Agent" : "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:61.0) Gecko/20100101 Firefox/61.0",
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
}
# 配置使用 Pipeline
ITEM_PIPELINES = {
    'NewsSpider.pipelines.NewsspiderPipeline': 300,
}
```

5. **执行 Spider**。（项目目录中执行）scrapy crawl instant_position。

scrapy 爬虫数据保存到 MySQL 数据库

在 MySQL 的 python 数据库中执行如下 SQL 语句来创建 instant_inf 数据表：

```
CREATE TABLE instant_inf (
  id INT (11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR (255),
  url VARCHAR (255)
)
```

修改 pipelines.py 文件

```
# 项目目录 \NewsSpider\pipelines.py
import mysql.connector            # 导入访问 MySQL 的模块
class NewsspiderPipeline(object):
    def __init__(self):      # 定义构造器，初始化要写入的文件
        self.conn = mysql.connector.connect(user='root', password='rootpassword',
            host='localhost', port='3306', database='python', use_unicode=True)
        self.cur = self.conn.cursor()
    def close_spider(self, spider):          # 重写 close_spider 回调方法，用于关闭数据库资源
        print('---------- 关闭数据库资源 ----------')
        self.cur.close()      # 关闭游标
        self.conn.close()     # 关闭连接
    def process_item(self, item, spider):
        self.cur.execute("INSERT INTO instant_inf VALUES(null, %s, %s,)", (item['title'], item['url']))
        self.conn.commit()
```

Scrapy 突破反爬虫机制

有些网站做了一些“反爬虫”处理，其网页内容不是静态的，而是使用 JavaScript 动态加载的，此时的爬虫程序也需要做相应的改进。

- **使用 shell 调试工具分析目标站点**（以 https://unsplash.com/ 为例）

  由于该网页是“瀑布流”设计（所谓“瀑布流”设计，就是网页没有传统的分页按钮，而是让用户通过滚动条来实现分页，当用户向下拖动滚动条时，程序会动态载入新的分页），当我们在 Firefox 中拖动滚动条时，可以在 Firefox 的调试控制台中看到再次向 https://unsplash.com/napi/photos?page=N&per_page=N 发送了请求，只是 page 参数发生了改变。可见，为了不断地加载新的图片，程序只要不断地向该 URL 发送请求，并改变 page 参数即可。

- **使用 Scrapy 爬取高清图片**（程序略）

Scrapy 反爬虫常见解决方案

- **IP 地址验证**　有些网站会使用 IP 地址验证进行反爬虫处理，程序会检查客户端的 IP 地址，如果发现同一个 IP 地址的客户端频繁地请求数据，该网站就会判断该客户端是爬虫程序。

  针对这种情况，可以让 Scrapy 不断地随机更换代理服务器的 IP 地址，这样就可以欺骗目标网站了。自定义一个下载中间件，让该下载中间件随机更换代理服务器即可：

  1. 打开 Scrapy 项目下的 middlewares.py 文件，在该文件中增加定义如下类：

```
class RandomProxyMiddleware (object) :
    def process request (self, request, spider):   # 动态设置代理服务器的 IP 地址
        #get_random_proxy() 函数随机返回代理服务器的 IP 地址和端口
        request.meta["proxy"] = get_random_proxy()
```

  2. 通过 settings.py 文件设置启用自定义的下载中间件。在 settings.py 文件中增加如下配置代码：

```
DOWNLOADER MIDDLEWARES = {
    'ZhipinSpider.middlewares.RandomProxyMiddleware': 543,     # 配置自定义的下载中间件
}
```

- **禁用 Cookie** 有些网站可以通过跟踪 Cookie 来识别是否是同一个客户端。如果同一个客户端在单位时间内的请求过于频繁，则基本可以断定这个客户端不是正常用户，很有可能是程序操作（比如爬虫），此时目标网站就可以禁用该客户端的访问。

  针对这种情况，可以让 Scrapy 禁用 Cookie(Scrapy 不需要登录时才可禁用 Cookie）。在 settings.py 文件中取消如下代码的注释即可禁用 Cookie：

```
COOKIES_ENABLED = False
```

- **违反爬虫规则文件** 在很多 Web 站点目录下都会提供一个 robots.txt 文件，在该文件中制定了一系列爬虫规则。

  为了让爬虫程序违反爬虫规则文件的限制，强行爬取站点信息，可以在 settings 文件中取消如下代码的注释来违反站点制定的爬虫规则：

```
ROBOTSTXT OBEY = False  # 指定不遵守爬虫规则
```

- **限制访问频率** 当同一个 IP 地址、同一个客户端访问目标网站过于频繁时（正常用户的访问速度没那么快），其很可能会被当成机器程序（比如爬虫）禁止访问。

  为了更好地模拟正常用户的访问速度，可以限制 Scrapy 的访问频率。在 settings 文件中取消如下代码的注释即可限制 Scrapy 的访问频率：

```
AUTOTHROTTLE ENABLED = True      # 开启访问频率限制
AUTOTHROTTLE START DELAY = 5     # 设置访问开始的延迟
AUTOTHROTTLE MAX DELAY = 60      # 设置访问之间的最大延迟
# 设置 Scrapy 并行发给每台远程服务器的请求数量
AUTOTHROTTLE TARGET CONCURRENCY= 1.0
DOWNLOAD DELAY = 3          # 设置下载之后的自动延迟
```

- **图形验证码** 有些网站为了防止机器程序访问，会做一些很“变态”的设计，它会记录同一个客户端、同一个 IP 地址的访问次数，只要达到一定的访问次数（不管你是正常用户，还是机器程序），目标网站就会弹出一个图形验证码让你输入，只有成功输入了图形验证码才能继续访问。

  为了让机器识别这些图形验证码，通常有两种解决方式：

  - 使用 PIL、Libsvrn 等库自己开发程序来识别图形验证码。这种方式具有最大的灵活性，只是需要开发人员自己编码实现。
  - 通过第三方识别。有不少图形验证码的在线识别网站，它们的识别率基本可以做到 90% 以上。但是识别率高的在线识别网站通常都要收费，而免费的往往识别率不高，不如自己写程序来识别。

Scrapy Selenium 整合（启动浏览器并登陆）

某些网站要求用户必须先登录，然后才能获取网络数据，这样爬虫程序将无法随意爬取数据。为了登录该网站，通常有两种做法：

- 直接用爬虫程序向网站的登录处理程序提交请求，将用户名、密码、验证码等作为请求参数，登录成功后记录登录后的 Cookie 数据。**优点**是完全由自己来控制程序，因此爬虫效率高、灵活性好；**缺点**是编程麻烦，尤其是当目标网站有非常强的反爬虫机制时，爬虫开发人员要花费大量的时间来处理。
- 使用真正的浏览器来模拟登录，然后记录浏览器登录之后的 Cookie 数据。**优点**是简单、易用，而且几乎可以轻松登录所有网站（因为本来就是用浏览器登录的，正常用户怎么访问，爬虫启动的浏览器也怎么访问）；**缺点**是需要启动浏览器，用浏览器加载页面，因此效率较低。

在使用 Scrapy 开发爬虫程序时，经常会整合 Selenium 来启动浏览器登录。

Selenium 本身与爬虫并没有多大的关系，主要是作为 Web 应用的自动化测试工具来使用的。Selenium 可以驱动浏览器对 Web 应用进行测试，就像真正的用户在使用浏览器测试 Web 应用一样。后来的爬虫程序正是借助于 Selenium 的这个功能来驱动浏览器登录 Web 应用的。