第 3 章

函数及调用规则

函数是把具有独立功能的代码段组织成一个模块封装起来，使用时调用即可得到封装的功能。在任何计算机语言中函数一般分为系统函数和自定义函数两大类，前者是系统提供的，后者是用户自行编写的。函数的作用是提高编写效率以及代码的重用性。Python 不仅提供了各类数据类型的计算、处理函数，还提供了类型转换、日期、时间处理及辅助运算的其他函数。

3.1 转换函数及使用

Python 的转换函数包括整数到 ASCII 码、不同进制转换和类型转换。

3.1.1 ASCII 码、进制转换函数及应用案例

ASCII 码及进制转换函数如表 3-1 所示。

表 3-1 ASCII 码及进制转换函数

函 数	功 能 描 述
chr(x)	将一个 ASCII 整数转换为一个字符
ord(x)	将一个字符转换为它的 ASCII 码
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串
bin(x)	将一个整数转换为一个二进制字符串
int(x,[base])	将其他进制转换为十进制整数，base 为进制标识，默认为十进制
bool(x)	x=0 返回假(False)，任何其他 x 值都返回真(True)

对于 bool(x)函数，有以下说明。

(1) 对于数值数据，参数为非零值的 bool()函数返回值均为真(True)，只有参数为 0 才返回假(False)。例如：

```
print(bool(12))
print(bool(0))
```

运行结果为

```
True
False
```

(2) 对于字符串数据,参数为没有值的字符串(也就是 None 或空字符串)时返回 False,否则返回 True。例如:

```
print(bool(' '), bool(None))
print(bool('hello'), bool('a'))
```

运行结果为

```
False, False
True,True
```

(3) 参数为空的列表、字典和元组时返回 False,否则返回 True。例如:

```
a = [];b = ();c = {}
print(bool(a),bool(b),bool(c))
a.append(1)
print(bool(a),bool(b),bool(c))
```

运行结果为

```
False False False
True False False
```

(4) 可用 bool()函数判断一个值是否已经被设置。例如:

```
x = input('请输入一个数值:')
print( bool(x.strip()))
```

运行程序,提示输入数值后直接按 Enter 键,结果为 False; 再次运行程序,输入 4,结果为 True。

【例 3-1】 ASCII 码及进制转换的使用。

```
print(ord('b'))                          #字符转换为 ASCII 码
print(ch(100))                           #ASCII 码转换为字符
print(int('0b1111011', 2))               #二进制转换为十进制
print(bin(18))                           #十进制转换为二进制
print(int('011',8))                      #八进制转换为十进制
print(oct(30))                           #十进制转换为八进制
print(int('0x12',16))                    #十六进制转换为十进制
print(hex(87))                           #十进制转换为十六进制
print(bool(100))                         #转换为布尔型
print(ord('D'))                          #转换为 ASCII 码
print(float(123))                        #转换为浮点数
```

运行结果为

```
98
d
123
```

```
0b10010
9
0o36
18
0x57
True
68
123.0
```

3.1.2　类型转换函数及应用案例

常用的类型转换函数如表 3-2 所示。

表 3-2　常用的类型转换函数

函　　数	功 能 描 述
int(x)	将 x 转换为一个整数
float(x)	将 x 转换为一个浮点数
complex(real [,imag])	创建一个复数,real 为实部数据,imag 为虚部数据
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效 Python 表达式,并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	将序列 s 转换为可变集合
dict(d)	创建一个字典,d 必须是一个序列 (key,value)元组
frozenset(s)	将序列 s 转换为不可变集合

例如：

```
a = '01234'
b = list(a)
c = tuple(a)
print(b,c)
```

运行结果为

```
['0', '1', '2', '3', '4'] ('0', '1', '2', '3', '4')
```

【例 3-2】 类型转换函数的使用。

```
import math
a = 5.6 + 1.58j
print(math.pi)
print(int(math.pi))                          #将 π 转换为整数
print(complex(math.pi))                      #将 π 转换为复数
print('实部 = ',a.real,'虚部 = ',a.imag)        #获取实部和虚部
list = list(range(10))                       #生成 0～9 的列表
print(tuple(list))                           #将列表转换为元组
print(frozenset(list))                       #将列表转换为不可变集合
```

运行结果为

```
3.141592653589793
3
(3.141592653589793+0j)
实部= 5.6 虚部= 1.58
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

3.2 其他函数及使用

Python的其他函数可帮助完成各种数据操作,包括字符分割、判断、获取对象地址、返回模块变量和函数名等操作。其他函数如表3-3所示。

表3-3 其他函数

函 数	功 能 描 述
split()	将一个字符串分割成多个字符串
map ()	根据提供的函数对指定序列进行映射
divmod()	将除数和余数运算结果结合起来,返回一个包含商和余数的元组
all()	判断列表、元组及字典的所有元素是否有一个为空,是则返回False,否则返回True
any()	判断列表、元组及字典的某个元素是否为空,是则返回True,否则返回False
id()	用于获取对象的内存地址
zip()	返回对象中对应的元素打包为元组的列表。若元素个数不一致,则列表长度为最短的对象个数
dir()	返回模块中定义的所有模块、变量和函数

【例3-3】 其他函数的使用(1)。

```
def square(x):                              #计算平方数函数
    return x ** 2
map(square, [1,2,3,4,5])                    #返回迭代器
print(list(map(square, [1,2,3,4,5])))       #使用list()函数转换为列表
print(divmod(76,12))                        #返回商和余数
print(type(10.78),type('Python'))           #判断类型
list1 = ['a', 'b', 'c', 'd']
print(all(list1))
list2 = ['a', 'b', '', 'd']
print(all(list2))
print(any(list1))
print(any([0, '', False]))
```

运行结果为

```
[1, 4, 9, 16, 25]
(6, 4)
<class 'float'> <class 'str'>
True
```

```
False
True
False
```

【例 3-4】 其他函数的使用(2)。

```
str1 = [1,2,3,4]
str2 = ['北京','天津','南京','四川']
str3 = ['烤鸭','包子','鸭血粉丝','腊肠']
print(list(zip(str1,str2,str3)))
print(id(str1))
print(dir())
```

运行结果为

```
[(1, '北京', '烤鸭'), (2, '天津', '包子'), (3, '南京', '鸭血粉丝'), (4, '四川', '腊肠')]
2432647943040
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'str1', 'str2', 'str3']
```

3.2.1 split()函数的使用

split()函数用于分割字符串,语法格式为

```
split(sep, num)                          #sep 为分割符,num 为分割次数
```

(1) 省略 sep 时,默认用空格、\n、\t 分割字符串。例如:

```
str = "Line1 - abcdef1 \nLine1 - abc1 \nLine1 - abcd";
print(str.split())
```

运行结果为

```
['Line1 - abcdef1', 'Line1 - abc1', 'Line1 - abcd']
```

(2) 指定 sep 时,按 sep 的值分割字符串。例如:

```
str = "Line1 - abcdef \nLine1 - abc1 \nLine1 - abcd";
print(str.split("1",2))                  #按 1 分割两次
```

运行结果为

```
['Line', ' - abcdef \nLine', ' - abc1 \nLine1 - abcd']
```

(3) 当分割符在字符串首部或尾部时,需要注意结果前后多了空字符(当省略 sep 时,没有影响)。例如:

```
str = "abc define\napplea"
print(str.split("a"))                    #首部和尾部出现分割符用空格
```

运行结果为

```
['', 'bc define\n', 'pple', '']
```

(4) 当分割符连续出现多次时,分割符所在处用空格替代。例如:

```
str = "aabcaaadefine\nappleaa"
print(str.split("a"))
```

运行结果为

```
['', '', 'bc', '', '', 'define\n', 'pple', '', '']
```

(5) 多次分割,获取需要的结果。例如:

```
str = "http://smbu.edu.cn/"
print(str.split("//")[1].split("/")[0].split("."))
```

运行结果为

```
['smbu', 'edu', 'cn']
```

3.2.2 map()函数的使用

map()函数用于对指定序列进行映射,语法格式为

```
map(function, iterable)              # function 为函数,iterable 为一个或多个序列
```

(1) 将元组转换为整数列表。例如:

```
map(int, (1,2,3))
```

运行结果为

```
[1, 2, 3]
```

(2) 将字符串转换为整数列表。例如:

```
map(int, '1234')
```

运行结果为

```
[1, 2, 3, 4]
```

(3) 提取字典的 key,并将结果存放在一个列表中。例如:

```
map(int, {1:2,2:3,3:4})
```

运行结果为

```
[1, 2, 3]
```

(4) 将字符串转换为元组,并将结果以列表的形式返回。例如:

```
map(tuple, 'agdf')
```

运行结果为

```
[('a',), ('g',), ('d',), ('f',)]
```

(5) 函数依次作用在列表的每个元素上,不改变原列表,返回一个新列表。例如:

```
def double(x):
  return x * 2
list1 = list(range(1,10,2))
list2 = list(map(double,list1))
print(list1)
print(list2)
```

运行结果为

```
[1, 3, 5, 7, 9]
[2, 6, 10, 14, 18]
```

【例 3-5】 将 city = ["beijing", "shanghai", "shenzhen", "nanjing"] 列表中的字符串首字母转换为大写。

```
def to_title(str):
    return str.title()                    #title()函数转换首字母为大写
city = ["beijing", "shanghai", "shenzhen", "nanjing"]
dict1 = map(to_title, city)               #调用函数转换映射成列表
print(list(dict1))
```

运行结果为

```
['Beijing', 'Shanghai', 'Shenzhen', 'Nanjing']
```

结论:map()函数作用于一个可迭代对象,它可应用于这个可迭代对象的每个元素。

3.2.3　split()与 map()函数联合使用

若用一条命令将多个数据赋给不同变量,可联合使用 split()和 map()函数,实现多个数据的输入。

(1) 输入 3 个整数,分别赋值给 a,b,c,使用空格隔开,代码如下。

```
a, b, c = map(int, input('请输入方程的系数 a,b,c:').split())
print("a = ",a,",b = ",b,",c = ",c)
```

运行结果为

```
请输入方程的系数 a,b,c:3 4 5
a = 3,b = 4,c = 5
```

(2) 输入 3 个浮点数,分别赋值给 a,b,c,使用逗号隔开,代码如下。

```
a, b, c = map(float, input('请输入方程的系数 a,b,c:').split(','))
print("a = ",a,",b = ",b,",c = ",c)
```

运行结果为

```
请输入方程的系数 a,b,c:3.56,7.89,2.61
a = 3.56,b = 7.89,c = 2.61
```

【例 3-6】 将列表 l1 和 l2 形成字典，l1 为键，l2 为值。

```
l1 = [1,2,3,4,5]
l2 = ['北京','上海','深圳','广州','南京']
dict1 = dict(zip(l1,l2))                    #使用 zip()函数将 l2 插入 l1 中打包成字典
print(dict1)
#或使用以下语句实现
dict2 = dict(map(lambda key, value: [key, value], l1, l2))     #l1 为键,l2 为值
print(dict2)
```

运行结果为

```
{1: '北京', 2: '上海', 3: '深圳', 4: '广州', 5: '南京'}
{1: '北京', 2: '上海', 3: '深圳', 4: '广州', 5: '南京'}
```

3.3 时间和日期函数及使用

Python 程序能用多种方式处理时间和日期，转换日期格式是一个常见的功能。Python 提供了 datetime(表示日期和时间的结合)、time(时间)和 calendar(日历)模块；此外，还提供了用于格式化日期和时间的函数。时间间隔是以秒为单位的浮点数。每个时间戳都以 1970 年 1 月 1 日 0 时(历元)经过了多长时间来表示。当获取日期或时间时，需要导入以下模块。

```
import datetime                    #引入日期和时间模块
import time                        #引入时间模块
import calendar                    #引入日历模块
```

3.3.1 datetime 模块函数及应用案例

datetime(时间、日期)模块函数如表 3-4 所示。

表 3-4 datetime(时间、日期)模块函数

函 数	功 能 描 述
datetime.today()	返回一个表示当前本地日期的 datetime 对象
datetime.now()	返回一个表示当前本地日期和时间的 datetime 对象
datetime.date	提供 year、month、day 属性
datetime.datetime	提供 year、month、day、hour、minute、second 属性，用于将对象从字符串转换为 datetime 对象
datetime.strptime(date_string, format)	将格式字符串转换为 datetime 对象
datetime.timedelta	对象表示一个时间长度，即两个日期或时间的差值
datetime.strftime	用来获得当前时间，可以将时间格式化为字符串

【例 3-7】 datetime 模块函数的使用。

```
import datetime
t1 = datetime.datetime.now()
```

```
print("当前的日期和时间是 {}" .format(t1))
print("当前的年份是 {}" .format(t1.year),'年')
print("当前的月份是 {}" .format(t1.month),'月')
print("当前的日期是 {}" .format(t1.day),'日')
print("当前小时数是{}" .format(t1.hour),'时')
print("当前分钟数是 {}".format(t1.minute),'分')
print("当前秒数是 {}" .format(t1.second),'秒')
print("今天日期是:{}-{}-{}" .format(t1.year,t1.month,t1.day ) )
print("当前时间是:{}:{}:{}" .format(t1.hour,t1.minute,t1.second ) )
```

运行结果为

```
当前的日期和时间是 2022-11-01 15:44:50.040323
当前的年份是 2022 年
当前的月份是 11 月
当前的日期是  1 日
当前小时数是 15 时
当前分钟数是 44 分
当前秒数是  50 秒
今天日期是: 2022-11-1
当前时间是: 15:44:50
```

3.3.2　time 模块函数

time(时间)模块函数如表 3-5 所示。

表 3-5　time(时间)模块函数

函　　数	功 能 描 述
time.altzone	返回西欧夏令时启用地区时间
time.asctime([tupletime])	接受时间元组并返回一个可读的时间日期形式字符串
time.clock()	以浮点数计算秒数返回当前 CPU 时间,衡量不同程序的耗时
time.ctime([secs])	相当于 asctime(localtime(secs)),未传入参数时相当于 asctime()
time.gmtime([secs])	接收时间戳(历元后经过的浮点秒数)并返回西欧天文时间元组 t 注：t.tm_isdst 始终为 0
time.localtime([secs])	接收时间戳(历元后经过的浮点秒数)并返回当地时间元组 t 注：t.tm_isdst 可取 0 或 1,取决于当地当时是否为夏令时
time.mktime(tupletime)	接收时间元组并返回时间戳(历元后经过的浮点秒数)
time.sleep(secs)	推迟调用线程的运行,secs 为秒数
time.strftime(fmt[,tupletime])	接收时间元组并返回可读字符串的当地时间,格式由 fmt 决定
time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')	根据 fmt 的格式把一个时间字符串解析为时间元组
time.time()	返回当前时间的时间戳(历元后经过的浮点秒数)
time.tzset()	根据环境变量 TZ 重新初始化时间相关设置

3.3.3　calendar 模块函数及应用案例

calendar 模块函数如表 3-6 所示。

表 3-6 calendar 模块函数

函 数	功 能 描 述
calendar.calendar(year,w=2,i=1,c=6)	返回字符串格式年历,3 个月一行,间隔距离为 c。每日宽度间隔为 w 字符。每行长度为 21×w+18+2×c。i 为每星期行数
calendar.firstweekday()	返回当前每周起始日期,默认情况下,首次载入 calendar 模块时返回 0,即星期一
calendar.isleap(year)	闰年返回 True,否则为 False
calendar.leapdays(y1,y2)	返回在 y1 与 y2 两年之间的闰年总数
calendar.month(year,month,w=2,l=1)	返回字符串格式的 year 年 month 月日历,两行标题,一周一行。每日宽度间隔为 w 字符。每行的长度为 7×w+6。l 为每星期的行数
calendar.monthcalendar(year,month)	返回整数的单层嵌套列表。每个子列表装载代表一个星期的整数。year 年 month 月外的日期都设为 0;范围内的日都由该月第几日表示,从 1 开始
calendar.setfirstweekday(firstweekday)	指定一周的第 1 天,firstweekday=0 表示星期一,firstweekday=1 表示星期二,…,firstweekday=6 表示星期六;也可以通过常量 MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY 和 SUNDAY 设置,如 calendar.setfirstweekday(calendar.SATURDAY)

要输出今天是星期几,可先获取年、月、日的整数,再使用 calendar 模块函数即可获得。方法是

```
import datetime
import calendar
y = int(datetime.datetime.now().strftime("%y"))
m = int(datetime.datetime.now().strftime("%m"))
d = int(datetime.datetime.now().strftime("%d"))
print("今天是星期: ",calendar.weekday(y,m,d) + 1)
```

也可使用以下方法获取。

```
Week = (datetime.date.today().weekday()) + 1
print("现在的时间是: ",Week)
```

【例 3-8】 输出当前日期、时间和星期数。

```
import calendar
import datetime
print("今天的日期是:",datetime.date.today())
print("现在的时间是:",datetime.datetime.now().strftime("%H时%M分%S秒"))
y = int(datetime.datetime.now().strftime("%Y"))
m = int(datetime.datetime.now().strftime("%m"))
d = int(datetime.datetime.now().strftime("%d"))
print("今天是星期:",calendar.weekday(y,m,d) + 1)
```

运行结果为

```
今天的日期是: 2022-11-01
现在的时间是: 15时48分50秒
今天是星期: 2
```

【例 3-9】 time 和 calendar 模块函数的使用。

```
import time
import calendar
ticks = time.time()
print("当前时间戳为:", ticks)
localtime = time.localtime(time.time())
print("无格式获取本地时间 :", localtime)
localtime = time.asctime( time.localtime(time.time()) )
print("有格式获取本地时间为 :", localtime)
print(time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()))
print(time.strftime("%a %b %d %H:%M:%S %Y", time.localtime()))
cal = calendar.month(2022, 10)
print("以下输出 2022 年 10 月份的日历:")
print(cal)
```

运行结果为

```
当前时间戳为: 1667215197.0234706
无格式获取本地时间 : time.struct_time(tm_year = 2022, tm_mon = 10, tm_mday = 31, tm_hour = 19,
tm_min = 19, tm_sec = 57, tm_wday = 0, tm_yday = 304, tm_isdst = 0)
有格式获取本地时间为 : Mon Oct 31 19:19:57 2022
2022-10-31 19:19:57
Mon Oct 31 19:19:57 2022
以下输出 2022 年 10 月份的日历:
    October 2022
Mo  Tu  We  Th  Fr  Sa  Su
                    1   2
 3   4   5   6   7   8   9
10  11  12  13  14  15  16
17  18  19  20  21  22  23
24  25  26  27  28  29  30
31
```

【例 3-10】 改变时间的输出方法。

```
import datetime
now = datetime.datetime.now()
time_date = now.strftime('%Y-%m-%d %H:%M:%S')
print('原始时间:\t\t\t\t{}'.format(time_date))
add_info = datetime.timedelta(days = 1, hours = 2, minutes = 3, seconds = 4)
time_date1 = datetime.datetime.strptime(time_date, '%Y-%m-%d %H:%M:%S')
add_end = add_info + time_date1
print('加上 1 天 2 个小时 3 分钟 4 秒后:\t{}'.format(add_end))
add_info1 = datetime.timedelta(days = -1, hours = -2, minutes = -3, seconds = -4)
dec_end = time_date1 + add_info1
print('减去 1 天 2 个小时 3 分钟 4 秒后:\t{}'.format(dec_end))
```

运行结果为

```
原始时间:                          2022-10-31 19:24:42
加上 1 天 2 个小时 3 分钟 4 秒后:   2022-11-01 21:27:46
```

```
减去 1 天 2 个小时 3 分钟 4 秒后:  2022-10-30 17:21:38
```

【例 3-11】 输出指定的日历。

```
import calendar
#输入指定年月
yy = int(input("输入年份: "))
mm = int(input("输入月份: "))
#显示日历
print(calendar.month(yy,mm))
```

运行结果为

```
输入年份: 2023
输入月份: 1
    January 2023
Mo  Tu  We  Th  Fr  Sa  Su
                        1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19  20  21  22
23  24  25  26  27  28  29
30  31
```

3.4 匿名函数

匿名函数是一种通过单个语句生成函数的方式,使用 lambda 关键字实现,所以又称为 lambda 函数,适合完成简单函数计算。lambda 函数不仅减少了代码的冗余,且不用命名函数也可快速实现函数功能。lambda 函数使代码的可读性更强,程序看起来更加简洁。

3.4.1 lambda 函数的使用规则

lambda 函数是 Python 独有的功能,其使用规则如下。

(1) Python 使用 lambda 关键字创建匿名函数。lambda 函数的主体是一个表达式,而不是一个代码块,仅能在 lambda 表达式中封装有限的逻辑,可替代简单的函数功能。

(2) lambda 函数拥有自己的命名空间,且不能访问参数列表之外或全局命名空间内的参数。

(3) lambda 函数只能写一行,却不等同于 C 或 C++语言的内联函数,目的是调用函数时不占用栈内存,增加运行效率。

lambda 函数的语法格式为

```
lambda [变量 1 [,变量 2,...,变量 n]]:表达式
```

例如:

```
def square1(n):
    return n ** 2
print(square1(10))
```

可写成

```
square1 = lambda n: n ** 2
print(square1(10))
```

将实参直接调用，可写成

```
print((lambda x: x ** 2)(10))
```

以上 3 组代码运行结果均为 100。

若多个变量使用 lambda 函数，例如：

```
mul = lambda x,y,z:x * y * z
print(mul(2,5,8))
```

输出结果为

```
80
```

3.4.2　lambda 函数的应用案例

【例 3-12】 根据字符串中不同字符的数量，对一个字符串列表进行排序，将字符数最多的字符串放在最后。

```
str = ['father', 'mather', 'sister', 'brother', 'uncle']
str.sort(key = lambda x:len(list(x)))
print(str)
```

运行结果为

```
['uncle', 'father', 'mather', 'sister', 'brother']
```

【例 3-13】 使用 range(10)产生列表，要求：

(1) 将列表按照元素与 5 的距离升序排列；

(2) 将列表各元素进行乘 2 操作并输出。

```
list1 = list(range(10))
list2 = sorted(list1,key = lambda x:abs(5 - x))        #按照元素与 5 的距离排序
print(list2)
list3 = map(lambda x:x * 2,list1)                      #将列表元素分别乘 2 操作
print(list(list3))
```

运行结果为

```
[5, 4, 6, 3, 7, 2, 8, 1, 9, 0]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

说明：lambda 和 def 关键字声明的函数不同，lambda 函数对象自身并没有一个显式的__name__属性，这是 lambda 函数被称为匿名函数的一个原因。

【例 3-14】 office 列表由 5 个单独的字典组成，在字典组成的列表中获取不同的 key 值。

```
office = [
{'姓名':'张三','电话分机':'3222','办公室':101},
{'姓名':'李四','电话分机':'4621','办公室':102},
{'姓名':'王五','电话分机':'5214','办公室':103},
{'姓名':'赵六','电话分机':'7666','办公室':104},
{'姓名':'胡七','电话分机':'8668','办公室':105}
]
staff_name = list(map(lambda name:name['姓名'],office))
staff_num = list(map(lambda num:num['办公室'],office))
print(staff_name)
print(staff_num)
```

运行结果为

```
['张三', '李四', '王五', '赵六', '胡七']
[101, 102, 103, 104, 105]
```

说明：当每个字典的 key 名称都一样时，可以通过 list[key]来获取。在字典组成的列表中无法直接通过 office[姓名]获取姓名值，此时需要用到 map()和 lambda 函数组合才能获取。即 map()函数用作接收一个函数，获取列表序列 office 中的 name['姓名']元素。同理，num['办公室']的获取方法也是如此。

3.5 函数调用

Python 函数是经过组织、可重复使用的、用来实现单一或相关联功能的代码段，用于提高应用模块效率和代码的重复利用率。用户自定义函数是根据自己需求功能定义的函数。无论哪种函数，其基本规则如下。

(1) 函数代码块以 def 关键词开头，后接函数标识符名称和括号“()”。任何传入参数和自变量必须放在括号中。括号之间用于定义参数。

(2) 函数的第 1 行语句可以选择性地使用文档字符串，常用于存放函数说明。

(3) 函数内容以冒号起始，并且缩进。

(4) return [表达式] 用于结束函数，可返回一个值给调用方，省略 return 相当于返回 None。

3.5.1 自定义函数

自定义函数的语法格式为

```
def 函数名( 形参列表 ):
    "函数_文档说明字符串"
    实现特定功能的语句代码
    return [返回值]
```

说明：

(1) 函数名是一个符合 Python 语法的标识符，最好能够体现出该函数的功能，不建议使用 x、y、z 等简单字符。例如，解方程函数可命名为 def equation(a,b,c):。

(2) 形参列表：函数可以接收多少个参数，多个参数之间用逗号分隔，没有形参时，必须加括号，后面的冒号不能省略。

(3) 若定义一个没有任何功能的空函数，可以使用 pass 语句作为占位符。

(4) return [返回值]为函数的可选参数，用于设置该函数的返回值，若没有返回值，可省略该语句，不写或只写 return 即可。

例如，自定义函数计算两个数的和，代码如下。

```
def add( a,b ):
    #学习 Python 函数
    c = a + b
    return c
```

若在函数中输出，可写为

```
def add( a,b ):
    #学习 Python 函数
    c = a + b
    print(c)
```

3.5.2　函数调用及应用案例

定义函数只是给了函数一个名称，函数名其实就是指向一个函数对象的引用，可以把函数名赋给一个变量。函数指定了函数中包含的形参(形式参数)、代码块结构。当函数完成以后，不调用是不能执行的，可在本程序中调用或通过另一个函数调用执行，也可以直接在 Python 命令提示符下调用执行，调用时的参数为实际参数，简称为实参。

1. 有参函数的调用

例如，调用两次前面定义的 add()函数。

```
def add( a,b ):
    #学习 Python 函数
    c = a + b
    return c
print(add(3,2))                    #第 1 次调用
print(add(15,7))                   #第 2 次调用
```

若在函数中输出，可写为

```
def add( a,b ):
    #学习 Python 函数
    c = a + b
    print(c)
add(3,2)                           #第 1 次调用
add(15,7)                          #第 2 次调用
```

调用两次后结果为 5 和 22。

2. 无参函数的调用

当函数不需要参数时，直接调用即可。例如：

```
def beijing():
    #学习 Python 无参数函数
    print('北京是中国的首都')
beijing()                          #调用
```

运行结果为

```
北京是中国的首都
```

3. 调用规则

函数调用时，实参和形参结合，其个数和顺序是一一对应的。Python 在实际调用中，允许实参个数少于形参个数，反之则不行。若调用时实参个数少于形参个数，执行时可使用默认值。例如：

```
def student( name, mathematics = 90, Python = 95 ):
    print('姓名:',name)
    print('数学:', mathematics )
    print('Python 语言:', Python)
student('葛明')                    #调用
```

运行结果为

```
姓名: 葛明
数学: 90
Python 语言: 95
```

若形参和实参一致，则覆盖默认值。例如：

```
def student( name, mathematics = 90, Python = 95 ):
    print('姓名:', name )
    print('数学:', mathematics )
    print('Python 语言:', Python)
student('葛明',100,100)
```

运行结果为

```
姓名: 葛明
数学: 100
Python 语言: 100
```

【例 3-15】 输出调用的结果。

```
img1 = [12,34,56,78]
img2 = [1,2,3,4,5]
def displ():
    print(img1)
def modi():
    img1 = img2
    print(img1)
modi()
displ()
```

运行结果为

```
[1, 2, 3, 4, 5]
[12, 34, 56, 78]
```

3.5.3　函数传递及应用案例

在 Python 中，类型属于对象，变量的类型取决于对象的引用。例如：

```
a = [1,2,3]              # a 是 list 类型
a = " 鲁滨逊"            # a 是 string 类型
```

说明：

(1) 变量 a 没有类型，它只是一个对象的引用(一个指针)，可以是列表(list) 类型对象，也可以是字符串(string)类型对象。

(2) Python 中一切都是对象，分为不可变对象和可变对象。其中，字符串(string)、元组(tuple)和数字(number)是不可变对象；而列表(list)、字典(dict)等则是可变对象。

(3) 不可变类型：若 a=15，再赋值 a=10，是改变 a 的值，相当于新生成了 a。

(4) 可变类型：若 list1=[1,2,3,4]，再赋值 list1[2]=5，则是将 list1 列表的第 3 个值更改，list1 没有变，只是其内部的一部分值被修改了。

(5) Python 函数的参数传递规则：不可变类型类似 C++语言的值传递，如整数、字符串、元组；可变类型传递参数需要加 * 符号。

【例 3-16】 使用面积函数计算不同输入值的面积。

```
def area(width, height):
    return width * height
def print_welcome(name):
    print("欢迎", name)
print_welcome("张三峰")
w = eval(input("输入矩形的宽度:"))
h = eval(input("输入矩形的长度:"))
print("宽度=", w, "长度=", h, " 矩形面积是: ", area(w,h))
```

运行结果为

```
欢迎 张三峰
输入矩形的宽度: 15
输入矩形的长度: 26
宽度=15 长度=26  矩形面积是: 390
```

【例 3-17】 计算 $x^2+y/4$。

```
def func(x,y):
  x1 = x**2
  y1 = y/4
  result = x1 + y1
  print("计算结果是: ",str(result))
func(3,5)
```

运行结果为

```
计算结果是:  10.25
```

【例 3-18】 传递可变对象。

```
def changeme(mylist):
    print("修改传入的列表")
    mylist.append([11, 12, 13, 14])
    print("函数内取值: ",mylist)
    return
#调用 changeme()函数
mylist = [10, 20, 30]
changeme(mylist)
print('函数外取值:',mylist)
```

由于传入函数和在末尾添加新内容的对象用的是同一个引用,运行结果为

```
修改传入的列表
函数内取值: [10, 20, 30, [11, 12, 13, 14]]
函数外取值: [10, 20, 30, [11, 12, 13, 14]]
```

3.5.4 函数参数、返回值及应用案例

调用函数时可使用的参数类型包括必备参数、关键字参数、默认参数、不定长参数和匿名函数。

1. 必备参数

必备参数须以正确的顺序传入函数,调用有参函数时必须传入一个参数。

当调用 printme()函数时,必须传入一个参数,否则会出现语法错误。例如,以下代码执行时会出现错误。

```
def printme(str):
    print(str)
    return
printme()              #调用
```

运行错误为

```
TypeError:printme() missing 1 required positional argument: 'str'
```

2. 关键字参数

关键字参数和函数调用关系紧密。函数调用时使用关键字参数确定传入的参数值,允许函数调用时参数顺序与声明时不一致,因为 Python 解释器能够用参数名匹配参数值。例如:

```
def printme( str ):
   print("打印任何传入的字符串")
   print (str)
   return
```

```
#调用 printme 函数
printme( str = "定义我的字符串")
```

运行结果为

```
打印任何传入的字符串
定义我的字符串
```

调用时也可不指定关键字参数。例如：

```
def printinfo(name, age):
#"打印任何传入的字符串"
    print("Name: ", name)
    print("Age: ", age)
    return
#调用 printinfo 函数
printinfo(age = 50,name = "Lucy")
```

运行结果为

```
Name: Lucy
Age: 50
```

【例 3-19】 实参和形参个数不同时的函数调用。

```
def calu(x = 4, y = 1, z = 10):
    return(x ** y * z)
print(calu(2,3))              #调用函数并输出
```

运行结果为

```
80
```

说明：调用的实参覆盖了原参数 x=4,y=1。

3. 默认参数

调用函数时,参数的值若没有传入,则被认为采用默认值。

例如,以下代码中如果 age 参数没有被传入值,则输出默认的 age 值。

```
def printinfo(name, age = 35):
    #"打印任何传入的字符串"
    print("Name: ", name)
    print("Age: ", age)
    return
#调用 printinfo ()函数
printinfo(age = 50, name = "Lucy")
printinfo(name = "lucy")
```

运行结果为

```
Name: Lucy
Age: 50
Name: lucy
Age: 35
```

4. 不定长参数

若不能确定传递的参数个数,称为不定长参数,可以用 * 和 ** 来实现。加了 * 符号的参数会以元组(tuple)形式传入。例如:

```
def function( * args):
    print(args)
function(12,35,65)
```

运行结果为

```
(12, 35, 65)
```

加了 ** 符号的参数会以字典形式传入。例如:

```
def function( ** kwargs):
    print(kwargs)
function(a = 12,b = 35,c = 65)
```

运行结果为

```
{'a': 12, 'b': 35, 'c': 65}
```

说明:

(1) 这里传入的参数键值是成对出现的。

(2) 当一个星号(*)和两个星号(**)同时出现时,一个星号必须在两个星号前面。例如:

```
def function( * args, ** kwargs):
    print(args)
    print(kwargs)
```

【例 3-20】 未命名变量参数的使用。

```
def fun(a, b, * args):
    print(a)
    print(b)
    print(args)
    print(" = " * 9)
    ret = a + b
    ret1 = args
    return ret,ret1
print(fun(1,2,3,4,5))
```

运行结果为

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

【例 3-21】　使用列表和字典的可变参数。

```
def fun(a, b, *args, **kwargs):
    print(a)
    print(b)
    print(args)
    print(kwargs)
fun(1, 2, 3, 4, name = "hello", age = 20)
```

运行结果为

```
1
2
(3, 4)
{'name': 'hello', 'age': 20}
```

【例 3-22】　使用元组和字典作为形式参数。

```
def fun(a, b, *args, **kwargs):
    print(a)
    print(b)
    print(args)
    print(kwargs)
tup = (11,22,33)
dic = {"name":"hello", "age":20}
fun(1, 2, *tup, **dic)
```

运行结果为

```
1
2
(11, 22, 33)
{'name': 'hello', 'age': 20}
```

5. return 语句

return[表达式]是返回函数,不带参数值的 return 语句相当于返回 None。若函数中出现 return,表示这个函数运行到这里就结束了,后面不管有多少语句都不会再执行。

【例 3-23】　return 语句返回一个值。

```
def function():
    print("Apple")
    return "Banana"
    print("bb")
print(function())
```

运行结果为

```
Apple
Banana
```

说明：Python 函数可以以元组的方式返回多个值。例如：

```
def fun(a,b):
    return a,b,a+b
print(fun(10,20))
```

运行结果为

```
(10 20 30)
```

【例 3-24】 使用 return 语句返回多个值。

```
def sum_mul(arg1, arg2):
    #返回两个参数的和
    total = arg1 + arg2
    multiple = arg1 * arg2
    print("相加结果: ", total)
    print("相乘结果: ", multiple)
    return total,multiple
    print(arg1)
print("返回值:",sum_mul(5, 8))            #调用 sum
```

运行结果为

```
相加结果: 13
相乘结果: 40
返回值: (13, 40)
```

3.6 嵌套函数

函数体内又包含另外一个函数的完整定义,称为嵌套定义。C 语言不允许嵌套定义,只允许嵌套调用;Python 既允许嵌套定义,也允许嵌套调用。

3.6.1 嵌套定义

嵌套定义的语法格式为

```
def 函数名 1( 参数列表 )
    def 函数名 2( 参数列表 )
    …
        return
    return
```

例如,定义 fun1()函数,内嵌 fun2()函数,fun2()函数又内嵌 fun3()函数,方法如下。

```
def fun1():
  def fun2():
    print('函数 fun2')
    def fun3():
      print('函数 fun3')
  print('函数 fun1')
```

3.6.2　嵌套调用及案例

函数的嵌套调用是指在调用一个函数的过程中，又调用了其他函数。例如，在调用 fun() 函数的过程中调用 fun1()函数，代码如下。

```
def fun():
  char = '函数变量'
  def fun1():
    print('嵌套函数中输出:',char)
  fun1()
fun()
```

运行结果为

```
嵌套函数中输出: 函数变量
```

（1）简单的嵌套函数。

【例 3-25】 使用简单嵌套函数。

```
def py():
    print("Python")
    def mat():
        print("MATLAB")
    mat()
py()
```

运行结果为

```
Python
MATLAB
```

（2）可以使用 return 语句返回嵌套的函数值。

【例 3-26】 使用嵌套函数中的 return 语句输出返回值。

```
def mul(factor):
    def mul2(number):
        return number * factor
    return mul2
print('输出结果为:',mul(3)(3))
```

运行结果为

```
输出结果为: 9
```

（3）内部函数定义的变量只在内部有效，包括其嵌套的内部函数，对外部函数无效。

【例 3-27】 使用外部函数无效的嵌套。

```
def fun():
  def fun1():
      x = 2022                                  #嵌套内部 fun1()函数中定义变量
      print('在 fun1()函数中:x = ', x)          #嵌套内部 fun1()函数中输出
```

```
        def fun2():
            print('在 fun2()函数中:x = ', x)          #嵌套内部 fun2()函数中输出
        fun2()
    fun1()
    #print(x)      #这里输出 x 无效                  #嵌套外部函数中输出出错
fun()                                               #缺少该句则无法输出结果
```

运行结果为

```
在 fun1()函数中: x = 2022
在 fun2()函数中: x = 2022
```

fun()函数是 fun1()函数的外部函数,而 fun1()函数又是 fun2()函数的外部函数。在 3 个函数中仅定义了一个变量 x,并且该变量是在 fun1()函数中定义的,分别在 fun1()函数和它的内部 fun2()函数中输出该变量的值。如果不调用 fun()函数,就无法输出 x 的值,且在 fun()函数外输出 x 也是无效的。

(4) 内部函数可以引用外部函数定义的变量,但是外部函数不能引用内部函数定义的变量。

例如,外部函数 fun()定义了一个变量 x,然后又在内部函数 fun1()中定义了一个变量 y,分别在内部函数和外部函数中引用这两个变量。

```
def fun():
    x = 60
    def fun1():
        y = 5
        print('在内部函数 fun1()中输出 y = {}, y = {}'.format(x,y))      #能输出
    fun1()
    print('在外部函数 fun2()中输出 y = {}, y = {}'.format(x,y))          #出错,不能输出 y
fun()
```

程序在引用内部变量时出错了,因为内部函数定义的变量只对这个函数本身及其内部函数可见,而对其外部函数不可见,所以在外部函数 fun()看来,变量 y 尚未定义。

(5) 嵌套的层次输出。

【例 3-28】 使用逐层嵌套函数输出。

```
x = 10
def outer():
    x = 1
    def inner():
        x = 2
        print("x1 = ",x)
    inner()
    print("x2 = ",x)
outer()
print("x3 = ",x)
```

运行结果为

```
x1 = 2
x2 = 1
x3 = 10
```

【例 3-29】 使用常规嵌套。

```
def add1(x, y):
    return x + y
def add2(x):
    def add(y):
        return x * y
    return add
g = add2(2)
print(add1(2, 3))
print(add2(2)(3))
print(g(3))
```

运行结果为

```
5
6
6
```

3.7 递归函数

在一个函数体内调用它自身，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。函数内部可以调用其他函数，当然在函数内部也可以调用自己。调用函数自身要设置正确的返回条件，其特点如下。

(1) 函数内部的代码是相同的，只是针对参数不同，处理的结果不同；

(2) 当参数满足某个条件时，函数不再执行，通常称为递归的出口，否则会出现死循环。

说明：Python 默认递归深度为 100 层(Python 限制)，使用递归函数的优点是逻辑简单清晰，缺点是过深的递归调用会导致栈溢出，且占用内存比较大。

例如，已知有一个数列 fn，各元素取值为 fn(0)＝1，fn(1)＝4，…，fn(n) ＝ 2fn(n－1)＋fn(n－2)，其中 n 为大于或等于 2 的整数，求 fn(10)的值。

可以使用递归程序定义一个 fn(n)函数，用于计算 fn(10)的值。fn(10)等于 2×fn(9)＋fn(8)，其中 fn(9)又等于 2×fn(8)＋fn(7)，以此类推，最终会计算到 fn(2)等于 2×fn(1)＋fn(0)，即 fn(2)是可计算的，这样递归带来的隐式循环就能自动结束。顺着这个递推回去，最后就可以得到 fn(10)的值。

对于递归的过程，当一个函数不断地调用它自身时，必须在某个时刻函数的返回值是确定的，即不再调用它自身；否则，这种递归就变成了无穷递归，类似于死循环。一般递归函数定义规则如下。

```
def sum_number(num):
    print(num)
     #递归的出口,当参数满足某个条件时,不再执行函数
     if num == 1:
         return
    #自己调用自己
     sum_number(num - 1)
```

说明：递归程序一般与循环联用，具体使用方法见 4.6 节案例。

3.8 局部变量与全局变量

局部变量与全局变量的定义范围不同，作用域也不同。

3.8.1 局部变量及应用案例

局部变量只能在其被声明的函数内部访问，Python 默认任何在函数内赋值的变量和函数的形式参数都是局部变量。例如，在 sub()函数内定义的 a、b 变量均是局部变量。

【例 3-30】 局部变量的修改。

```
def fun1():
    x = 10
    print('fun1 中的局部变量 x =',x)
    x = 100
    print('fun1 修改后的局部变量 x =',x)
def fun2():
    x = 20
    print('fun2 中的局部变量 x =',x)
fun1()
fun2()
```

运行结果为

```
fun1 中的局部变量 x = 10
fun1 修改后的局部变量 x = 100
fun2 中的局部变量 x = 20
```

【例 3-31】 局部变量的返回值。

```
def sub(a):
    b = a ** 2
    print('a = ',a,',b = ',b)
    return b
print('返回值:',sub(10))                    #调用函数即可输出返回值
#print(a,b)                                 #由于 a,b 均是局部变量,无法使用,这里出错
```

运行结果为

```
a = 10, b = 100
返回值: 100
```

3.8.2　全局变量及应用案例

全局变量可以在整个程序范围内访问，Python 规定在函数外赋值的变量为全局变量。

1. 全局变量的使用

【例 3-32】 全局变量的使用。

```
x = 300
def fun1():
    print("fun1 内变量 x = ",x)
def fun2():
    print("fun2 内变量 x = ",x)
fun1()
fun2()
print("函数外变量 x = ",x)
```

运行结果为

```
fun1 内变量 x = 300
fun2 内变量 x = 300
函数外变量 x = 300
```

2. 全局变量与局部变量同名的使用

若全局变量与局部变量同名，则在函数内局部变量起作用。

```
total = 0                              #这里 total 是全局变量
def sum(arg1, arg2):                   #定义函数,arg1 和 arg2 均为局部变量
    #返回两个参数的和
    total = arg1 + arg2                #这里 total 是局部变量
    print("函数内是局部变量: ", total)
    return total
#调用 sum()函数
sum(10, 20)
print("函数外是全局变量: ", total)
```

运行结果为

```
函数内是局部变量:  30
函数外是全局变量:  0
```

【例 3-33】 全局变量与局部变量同名的使用。

```
a = 10                                 #全局变量
def fun():
    a = 20                             #局部变量
    b = a * 2                          #局部变量
    print('函数内部 a = ',a,',b = ',b)
fun()
print('函数外部 a = ',a)
```

运行结果为

```
函数内部 a = 20 ,b = 40
函数外部 a = 10
```

3.8.3 命名空间的作用域及应用案例

Python 命名空间由 4 种作用域组成，变量的作用域决定了访问特定变量的范围。

1. 命名空间

命名空间是变量名称的集合，定义在函数内部的变量拥有一个局部作用域，定义在函数外部的变量拥有全局作用域。命名空间是一个包含了变量名称(键)和它们各自相应的对象(值)的字典。

Python 命名空间有 4 种作用域，即局部作用域、全局作用域、外部作用域和内置作用域，它们在变量赋值时就已经生成，程序在解析某个变量名称对应的值时，首先从其所在函数的局部作用域进行查找，若没找到，再查找外部作用域(如果存在)，然后再到全局作用域查找，若还是没找到，就到内置作用域进行查找，如果在 4 个作用域内都没找到，则说明引用了一个未定义的变量，这时 Python 解释器就会报错。

一个变量是通过命名空间来查找使用的，同一个命名空间中，变量名称和字典的键一样，是独立的，不同命名空间内变量名称可重复使用。全局作用域对应于当前模块(或文件)，内置作用域对应于 Python 解释程序(需要系统内部的调用)。一个 Python 表达式可以访问局部作用域和全局作用域里的变量。如果一个局部变量和一个全局变量重名，则局部变量起作用，且会覆盖全局变量。

2. global 语句的使用

(1) 在函数内定义全局变量并赋值，必须使用 global 语句，语法格式为

```
global  变量名
```

例如，若命名一个全局变量 Money，在函数内使用 Money 变量并重新赋值，会出现“UnboundLocalError: local variable 'Money' referenced before assignment”的错误。此时需要添加 global Money 语句，即可在函数内赋值。

【例 3-34】 global 语句的使用。

```
def test(b = 2, a = 4):
    global z
    z += a * b
    return z
z = 20
print(z, test())
```

运行结果为

```
20  28
```

【例 3-35】 全局变量作用域的使用。

```
Money = 2000                              #定义 Money 全局变量
def AddMoney():
    global Money                          #若去掉该句,程序出错
    Money = Money + 22
def NewMoney():
  print('NewMoney 函数的输出 = ',Money)
#调用
print('AddMoney 函数调用前 Money = ',Money)
AddMoney()
print('调用 AddMoney 函数后 Money = ',Money)
NewMoney()
```

运行结果为

```
AddMoney 函数调用前 Money = 2000
调用 AddMoney 函数后 Money = 2022
NewMoney 函数的输出 = 2022
```

(2) 在函数内使用 global 语句修改全局变量。

【例 3-36】 在函数内修改全局变量。

```
a = 10
def fun():
    global a
    a = 20
    b = a * 2
    print('函数内部 a = ',a,',b = ',b)
fun()
print('函数外部 a = ',a)
```

运行结果为

```
函数内部 a = 20 ,b = 40
函数外部 a = 20
```

【例 3-37】 global 语句跳过中间层直接将嵌套作用域内的局部变量变为全局变量。

```
num = 20
def outer():
    num = 10
    def inner():
        global num
        print (num)
        num = 100
        print (num)
    inner()
    print(num)
outer()
print (num)
```

运行结果为

```
20
100
10
100
```

3.9 globals()函数与 locals()函数

globals()函数与 locals()函数可被用来返回全局作用域和局部作用域里的对象名称。如果在函数内部调用 locals()函数,返回的是所有能在该函数中访问的对象名称。如果在函数内部调用 globals()函数,返回的是所有在该函数中能访问的全局对象名称。两个函数的返回类型都是字典,所以它们的名称能用 keys()函数获取。

3.9.1 globals()函数及应用案例

globals()函数是 Python 的内置函数,它可以返回一个包含全局范围内所有变量的字典。该字典中的每个键值对,键为变量名,值为该变量的值。

【例 3-38】 globals()函数的使用。

```
x = 'globals()函数的使用'
y = '全局变量'
def fun_global():
  name = '这里是局部变量'
  area = '只能在函数中使用'
print(globals())
```

运行结果为

```
{...'x': 'globals()函数的使用', 'y': '全局变量'}
```

说明:globals()函数返回的字典中,会默认包含有多个变量(上述结果中已省略),这些都是 Python 主程序内置的。

3.9.2 locals()函数及应用案例

locals()函数也是 Python 的内置函数,通过调用该函数,可以得到一个包含当前作用域内所有变量的字典。

【例 3-39】 locals()函数的使用。

```
x = 'globals()函数的使用'
y = '全局变量'
def fun_global():
    name = '这里是局部变量'
    area = '只能在函数中使用'
    print(locals())
```

```
print(globals())
fun_global()
print("函数外的 locals 为:", locals())
```

运行结果为

```
{...'x': 'globals()函数的使用', 'y': '全局变量', 'fun_global': <function fun_global at
0x0000026E0D103E20>}
{'name': '这里是局部变量', 'area': '只能在函数中使用'}
函数外的 locals 为: {... 'x': 'globals()函数的使用', 'y': '全局变量', 'fun_global': <function
fun_global at 0x0000026E0D103E20>}
```

说明：locals()函数返回的局部变量组成的字典，可以用来访问变量，但无法修改变量的值。在函数内部调用 locals() 函数，会获得包含所有局部变量的字典；而在全局范围内调用 locals()函数，其功能和 globals()函数相同。