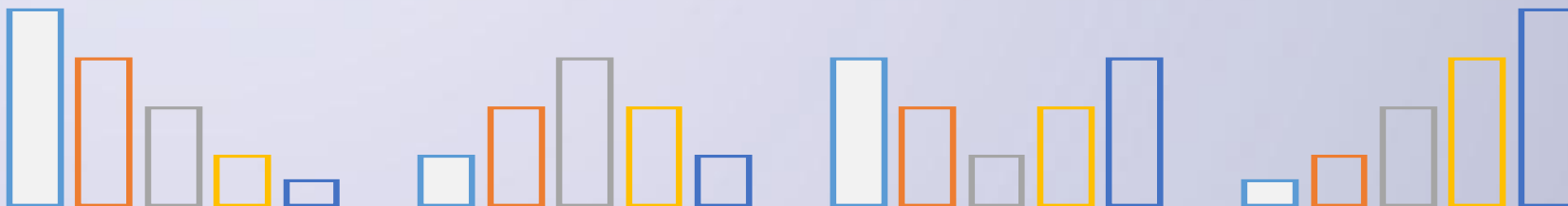
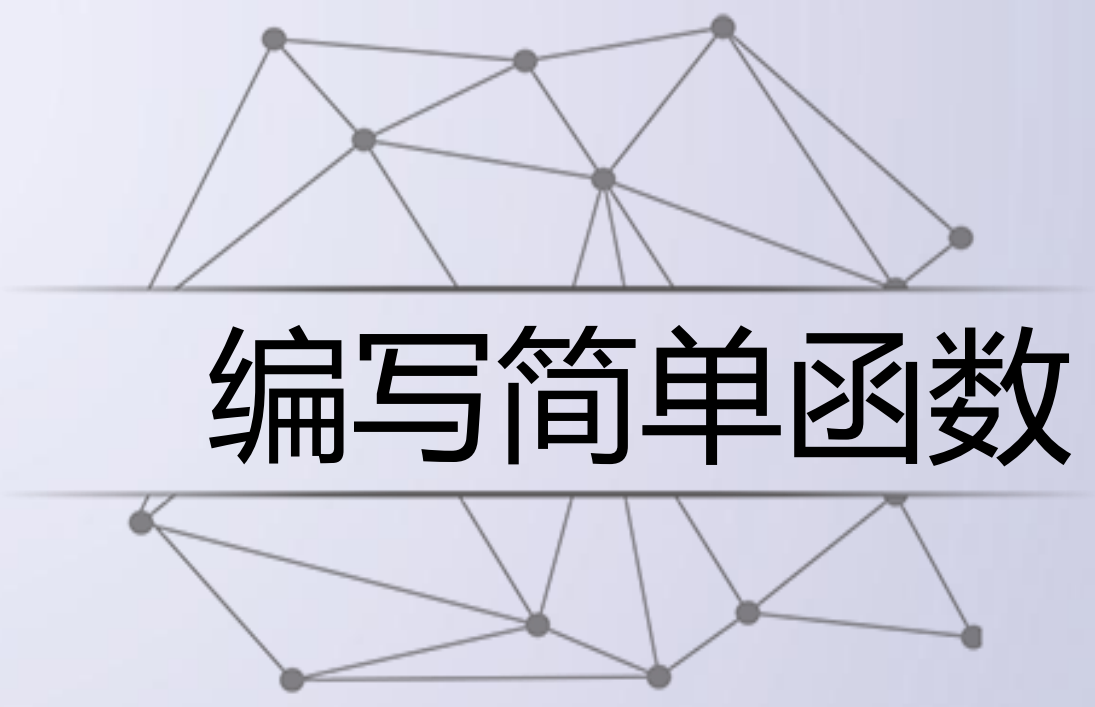


# 第3章 编写一个金融计算器



# 内容

- 编写简单的函数
- 生成自制的模块
- 列表数据类型



# 编写简单函数

# 函数的定义

- 函数是一段具有特定功能的、可重用的语句组，用函数名来表示，并通过函数名完成功能调用。
- 函数也可以看作是一段具有名字的子程序，可以在需要的地方调用执行，不需要在每个执行地方重复编写这些语句。
- 每次使用函数可以提供不同的参数作为输入，以实现对不同数据的处理；函数执行后，还可以反馈相应的处理结果。

# 编写函数语法结构

Python定义一个函数使用def保留字，语法形式如下：

```
def <函数名>(<参数列表>):
```

```
    <函数体>
```

```
    return <返回值列表>
```

# 例题：计算 $PV = \frac{FV}{(1+R)^n}$

其中PV：现值，FV：未来值，R：折现率，n：周期数；

假设在n=1年后获得FV=100元，若年折现率R=10%，  
问：FV的现值是多少？

编写不需要保存的Python函数

```
>>> def pv_f(fv, r, n):  
    return fv/(1+r)**n  
  
>>> pv_f(100, 1, 0.1)  
90.9090909090909
```

可以用dir()来检查函数是否存在

# 在编辑器里定义函数

1. 单击File->New File

2. 编程

```
def pv_f(fv, r, n):  
    return fv/(1+r)**n
```

3.单击File->Save (文件名: pvf.py)

4.Run->Run Module

用import激活自己编写的函数（要在同一目录下）

```
>>> from pvf import*
```

```
>>> pv_f(100, 1, 0.1)
```

```
90.9090909090909
```

## 补充：lambda函数

- Python的有33个保留字，其中一个lambda，该保留字用于定义一种特殊的函数——匿名函数，又称lambda函数。
- 匿名函数并非没有名字，而是将函数名作为函数结果返回，如下：

＜函数名＞ = lambda ＜参数列表＞：＜表达式＞

- lambda函数与正常函数一样，等价于下面形式：

```
def ＜函数名＞(＜参数列表＞):
```

```
    return ＜表达式＞
```



# lambda函数实例

简单说，lambda函数用于定义简单的、能够在一行内表示的函数，返回一个函数类型，实例如下。

```
>>>f = lambda x, y : x + y
>>>type(f)
<class 'function'>
>>>f(10, 12)
22
```

# 补充：可选参数和可变数量参数

- 在定义函数时，有些参数可以存在默认值

```
>>>def dup(str, times = 2):  
    print(str*times)  
>>>dup("knock~")  
knock~knock~  
>>>dup("knock~", 4)  
knock~knock~knock~knock~
```

- 在函数定义时，可以设计可变数量参数，通过参数前增加星号 (\*) 实现

```
>>>def vfunc(a, *b):  
    print(type(b))  
    for n in b:  
        a += n  
    return a  
>>>vfunc(1,2,3,4,5)  
<class 'tuple'>  
15
```

## 补充：参数的位置和名称传递

Python提供了按照形参名称输入实参的方式，调用如下：

```
result = func(x2=4, y2=5, z2=6, x1=1, y1=2, z1=3)
```

由于调用函数时指定了参数名称，所以参数之间的顺序可以任意调整。



# 变量的返回值

- `return`语句用来退出函数并将程序返回到函数被调用的位置继续执行。
- `return`语句同时可以将0个、1个或多个函数运算完的结果返回给函数被调用处的变量，例如。

```
>>>def func(a, b):  
    return a*b  
>>>s = func("knock~", 2)  
>>>print(s)  
knock~knock~
```




# 变量的返回值

- 函数可以没有return，此时函数并不返回值。例如：

```
def happy():  
    print("Happy birthday to you!")
```

- 函数也可以用return返回多个值，多个值以元组类型保存，例如：

```
>>>def func(a, b):  
    return b,a  
>>>s = func("knock~", 2)  
>>>print(s, type(s))  
(2, 'knock~') <class 'tuple'>
```



# 生成自制模块

# 将多个函数写在一个.py文件中

表3-1列出了Python金融计算器的主要功能，用到以下符号

- PV: 现值
- FV: 未来值
- R: 每期有效利率（折现率）
- n: 周期数
- C: 永久年金或定期年金的每期支付
- PMT: 永久年金或定期年金的每期支付（同C）
- g: 一个增长型永久年金（定期年金）的增长率
- APR: 年利率
- $R_C$ : 连续复利利率
- m: 每年复利次数

注意：C、R和n具有相同的频率，保持一致。

# 表3-1

|  |  |
|--|--|
| $FV = PV(1 + R)^n$   | $PV = \frac{FV}{(1 + R)^n}$                      |
| $PV(\text{永久年金}) = \frac{C}{R}$  | 假设第 1 次现金流量发生在第 1 期的结尾                           |
| $PV(\text{增长型永久年金}) = \frac{c}{R - g}$                                   | 假设第 1 次现金流量发生在第 1 期的结尾, 并且 $R > g$               |
| $PV(\text{定期年金}) = \frac{PMT}{R} \left[ 1 - \frac{1}{(1 + R)^n} \right]$ | 假设第 1 次现金流量发生在第 1 期的结尾                           |
| $FV(\text{定期年金}) = \frac{PMT}{R} [(1 + R)^n - 1]$                        | 假设第 1 次现金流量发生在第 1 期的结尾                           |
| $PV(\text{前置型永久年金}) = PV(\text{永久年锱金}) * (1 + R)$                        | 前置型: 现金流量发生在每一期的开始                               |
| $PV(\text{前置型定期年金}) = PV(\text{定期年金}) * (1 + R)$                         | $FV(\text{前置型定期年金}) = FV(\text{定期年金}) * (1 + R)$ |



# 表3-1（续）

|   |  |
|---|--|
| $PV(\text{债券}) = PV(\text{每期支付}) + PV(\text{面值})$ | $PV(\text{bond}) = \frac{c}{R} \left[ 1 - \frac{1}{(1+R)^n} \right] + \frac{FV}{(1+R)^n}$  |
| $EAR = \left( 1 + \frac{APR}{m} \right)^m - 1$    | <ul style="list-style-type: none"> <li>• <math>EAR</math>: 有效年利率</li> <li>• <math>APR</math>: 年利率</li> <li>• <math>m</math>: 每年复利次数</li> </ul> |
| 把一个年利率转换为另一个年利率                                   |  |
| 例如, 已知 $APR_1$ 、 $m_1$ 和 $m_2$ , 求出 $APR_2$       | $\left( 1 + \frac{APR_1}{m_1} \right)^{m_1} = \left( 1 + \frac{APR_2}{m_2} \right)^{m_2}$  |
| 把一个有效年利率转换为另一个有效年利率                               | $\left( 1 + R_{\text{eff}, m_1} \right)^{m_1} = \left( 1 + R_{\text{eff}, m_2} \right)^{m_2}$  |
| 把一个年利率转换为连续复利利率 $R_c$                             | $R_c = m * \ln \left( 1 + \frac{APR}{m} \right)$   |
| 把连续复利利率 $R_c$ 转换为年利率                              | $APR = m * \left( e^{\frac{R_c}{m}} - 1 \right)$   |

# 两种注释方法

- 用#来标识注释，每一行语句中#之后的字符就是注释，是单行注释，不会对语句的执行产生影响

```
>>> fv = 100 # this is comment
```

```
>>> fv
```

```
100
```

- 多行注释：用一对重复3次的双引号(即""")

```
"""
```

不执行

```
"""
```

- 在用help()函数时，注意只有def的第一行下的注释会显示出来，如果在注释前增加任何代码，就不会显示注释里面的内容

# 例题分析

```
>>> def pv_f(fv,r,n):  
    """  
  
    Objective: estimate present value  
    fv: future value  
    r : discount periodic rate  
    n : number of periods  
    formula :  $fv/(1+r)**n$   
    e. g. ,  
    >>>pv_f(100,0.1,1)  
    90.9090909090909  
    >>>pv_f(r=0.1, fv=100, n=1)  
    90.9090909090909  
    >>>pv_f(n=1, fv=100, r=0.1)  
    90.9090909090909  
    """  
  
    return fv/(1+r)**n
```

```
>>> help(pv_f)  
Help on function pv_f in module  
__main__:  
  
pv_f(fv, r, n)  
    Objective: estimate present value  
    fv: future value  
    r : discount periodic rate  
    n : number of periods  
    formula :  $fv/(1+r)**n$   
    e. g. ,  
    >>>pv_f(100,0.1,1)  
    90.9090909090909  
    >>>pv_f(r=0.1, fv=100, n=1)  
    90.9090909090909  
    >>>pv_f(n=1, fv=100, r=0.1)  
    90.9090909090909
```

# 增长型永久年金的现值:

- 公式:

$$PV(\text{永久年金}) = \frac{C}{R-g}$$

- 条件函数: `if()`

- 编程:

```
def pv_growing_perpetuity(c,r,g):  
    if(r<g):  
        print("r<g !!!!")  
    else:  
        return(c/(r-g))
```

```
>>> pv_growing_perpetuity(10,0.1,0.08)  
499.99999999999999  
>>> pv_growing_perpetuity(10,0.1,0.12)  
r<g !!!!
```

# 净现值和净现值法则

自编函数npv\_f.py

```
def npv_f(rate, cashflows):  
    total = 0.0  
    for i, cashflow in  
enumerate(cashflows):  
        total += cashflow / (1 +  
rate)**i  
    return total
```

- 循环for
- a+=b
- enumerate()

- 净现值(NPV)：所有收益和成本的现值的差
- 净现值法则  
$$\begin{cases} \text{if } NPV(\text{project}) > 0, \text{accept} \\ \text{if } NPV(\text{project}) < 0, \text{reject} \end{cases}$$

问题背景：投资一个起始资金100为期5年的项目。在未来5年每年年底的现金流分别为20, 40, 50, 20, 10。投资折现率是每年5%，是否该投资该项目？

# 内部收益率和内部收益率法则

```
def IRR_f(cashflows, iterations=100):  
    rate=1.0  
    investment=cashflows[0]  
    for i in range(1, iterations+1):  
        rate*=(1-  
npv_f(rate, cashflows)/investment)  
    return rate
```

- 循环for
- range()
- 调用函数npv\_f

- 内部收益率(IRR)：是使得净现值为零的折现率

- 净现值法则

$\begin{cases} \text{if } IRR(\text{project}) > R_{\text{capital}}, \text{accept} \\ \text{if } IRR(\text{project}) < R_{\text{capital}}, \text{reject} \end{cases}$

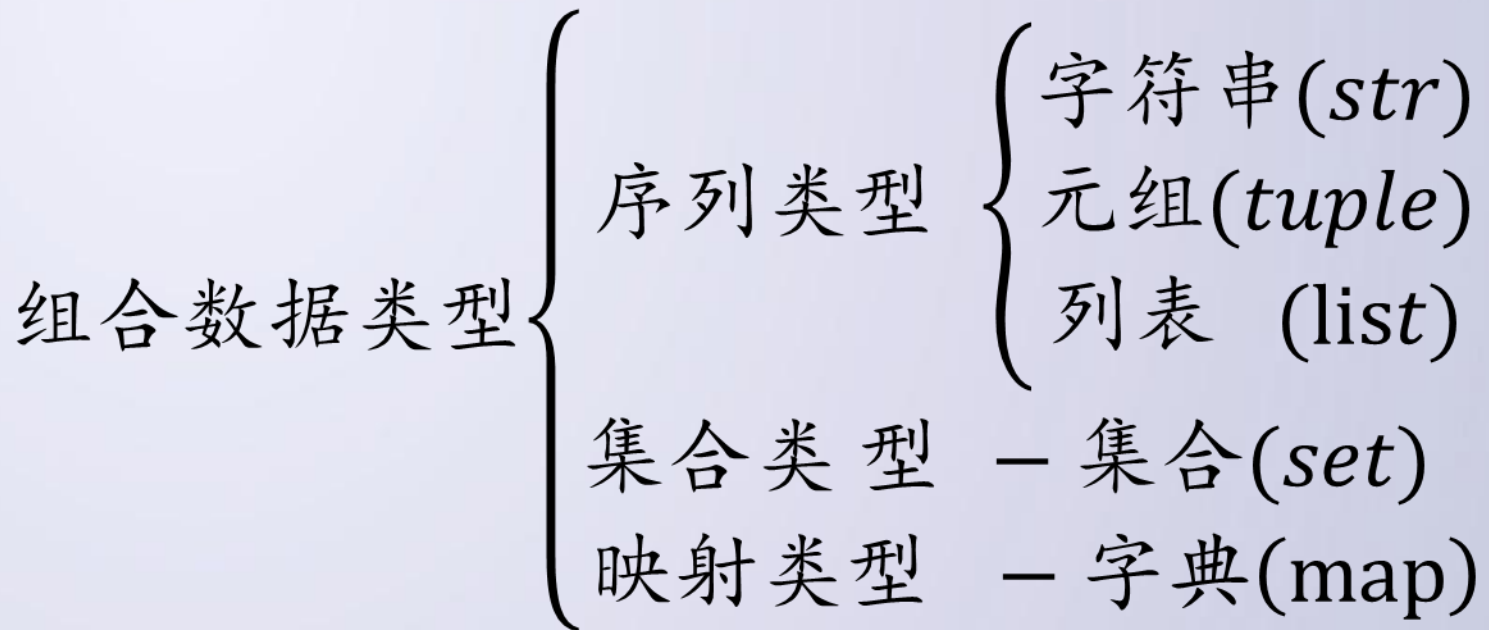
如果项目的内部收益率比资本成本大，就接收这个项目，否则拒绝。



# 列表数据类型



# 组合数据类型





# 序列类型

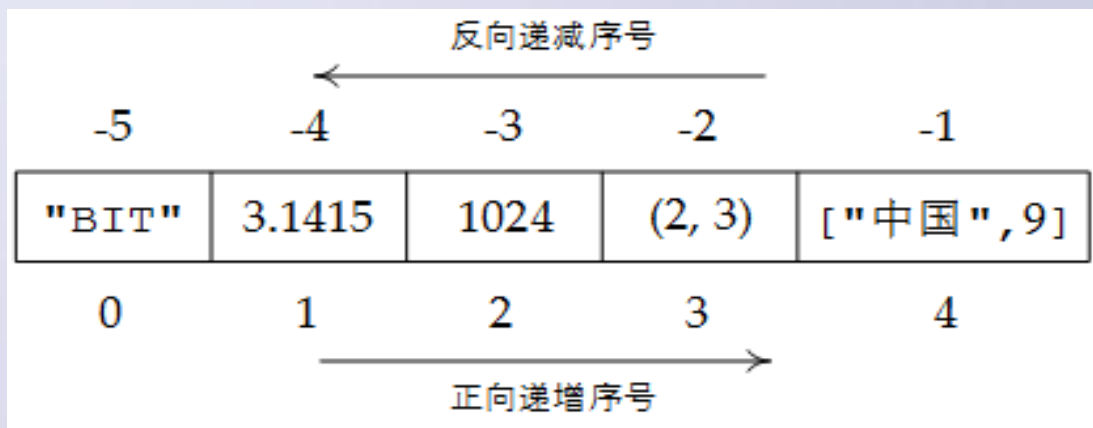
- 序列类型是一个元素向量，元素之间存在先后关系，通过序号访问，元素之间不排他。
- 集合类型是一个元素集合，元素之间无序，相同元素在集合中唯一存在。
- 映射类型是“键-值”数据项的组合，每个元素是一个键值对，表示为(key, value)。

# 序列类型

- 序列类型是一维元素向量，元素之间存在先后关系，通过序号访问。
- 当需要访问序列中某特定值时，只需要通过下标标出即可
- 由于元素之间存在顺序关系，所以序列中可以存在相同数值但位置不同的元素。
- 序列类型支持成员关系操作符（in）、长度计算函数（len()）、分片（[]），元素本身也可以是序列类型。
- Python语言中有很多数据类型都是序列类型，其中比较重要的是：str（字符串）、tuple（元组）和list（列表）

# 序列类型

- 元组是包含0个或多个数据项的不可变序列类型。元组生成后是固定的，其中任何数据项不能替换或删除，所有数据项包含在一对圆括号内。
- 列表则是一个可以修改数据项的序列类型，使用也最灵活



# 序列类型

## 序列类型有12个通用的操作符和函数

| 操作符                                     | 描述                              |
|---|---------------------------------|
| <code>x in s</code>                     | 如果x是s的元素，返回True，否则返回False       |
| <code>x not in s</code>                 | 如果x不是s的元素，返回True，否则返回False      |
| <code>s + t</code>                      | 连接s和t                           |
| <code>s * n</code> 或 <code>n * s</code> | 将序列s复制n次                        |
| <code>s[i]</code>                       | 索引，返回序列的第i个元素                   |
| <code>s[i: j]</code>                    | 分片，返回包含序列s第i到j个元素的子序列（不包含第j个元素） |
| <code>s[i: j: k]</code>                 | 步骤分片，返回包含序列s第i到j个元素以j为步数的子序列    |
| <code>len(s)</code>                     | 序列s的元素个数（长度）                    |
| <code>min(s)</code>                     | 序列s中的最小元素                       |
| <code>max(s)</code>                     | 序列s中的最大元素                       |
| <code>s.index(x[, i[, j]])</code>       | 序列s中从i开始到j位置中第一次出现元素x的位置        |
| <code>s.count(x)</code>                 | 序列s中出现x的总次数                     |

# 列表类型

列表（list）是包含0个或多个对象引用的有序序列，属于序列类型。与元组不同，列表的长度和内容都是可变的，可自由对列表中数据项进行增加、删除或替换。列表没有长度限制，元素类型可以不同，使用非常灵活。

# 列表类型

由于列表属于序列类型，所以列表也支持成员关系操作符（`in`）、长度计算函数（`len()`）、分片（`[]`）。列表可以同时使用正向递增序号和反向递减序号，可以采用标准的比较操作符（`<`、`<=`、`==`、`!=`、`>=`、`>`）进行比较，列表的比较实际上是单个数据项的逐个比较。

# 列表类型

列表用中括号（[]）表示，也可以通过list()函数将元组或字符串转化成列表。直接使用list()函数会返回一个空列表。

```
>>>ls = [425, "BIT", [10, "CS"], 425]
>>>ls
[425, 'BIT', [10, 'CS'], 425]
>>>ls[2][-1][0]
'C'
>>>list((425, "BIT", [10, "CS"], 425))
[425, 'BIT', [10, 'CS'], 425]
>>>list("中国是一个伟大的国家")
['中', '国', '是', '一', '个', '伟', '大', '的', '国', '家']
>>>list()
[]
```



# 列表类型

与整数和字符串不同，列表要处理一组数据，因此，列表必须通过显式的数据赋值才能生成，简单将一个列表赋值给另一个列表不会生成新的列表对象。

```
>>>ls = [425, "BIT", 1024]  #用数据赋值产生列表ls
>>>lt = ls                  #lt是ls所对应数据的引用，lt并不包含真实数据
>>>ls[0] = 0
>>>lt
[0, 'BIT', 1024]
```



# 列表类型的操作

| 函数或方法  | 描述   |
|--|--|
| <code>ls[i] = x</code>                             | 替换列表ls第i数据项为x                              |
| <code>ls[i: j] = lt</code>                         | 用列表lt替换列表ls中第i到j项数据（不含第j项，下同）              |
| <code>ls[i: j: k] = lt</code>                      | 用列表lt替换列表ls中第i到j以k为步的数据                    |
| <code>del ls[i: j]</code>                          | 删除列表ls第i到j项数据，等价于 <code>ls[i: j]=[]</code> |
| <code>del ls[i: j: k]</code>                       | 删除列表ls第i到j以k为步的数据                          |
| <code>ls += lt</code> 或 <code>ls.extend(lt)</code> | 将列表lt元素增加到列表ls中                            |
| <code>ls *= n</code>                               | 更新列表ls，其元素重复n次                             |
| <code>ls.append(x)</code>                          | 在列表ls最后增加一个元素x                             |
| <code>ls.clear()</code>                            | 删除ls中所有元素                                  |
| <code>ls.copy()</code>                             | 生成一个新列表，复制ls中所有元素                          |
| <code>ls.insert(i, x)</code>                       | 在列表ls第i位置增加元素x                             |
| <code>ls.pop(i)</code>                             | 将列表ls中第i项元素取出并删除该元素                        |
| <code>ls.remove(x)</code>                          | 将列表中出现的第一个元素x删除                            |
| <code>ls.reverse(x)</code>                         | 列表ls中元素反转                                  |

# 列表类型的操作

```
>>>vlist = list(range(5))

>>>vlist

[0, 1, 2, 3, 4]

>>>len(vlist[2:])          #计算从第3个位置开始到结尾的子串长度

3

>>>2 in vlist              #判断2是否在列表vlist中

True

>>>vlist[3]="python"      #修改序号3的元素值和类型

>>>vlist

[0, 1, 2, 'python', 4]

>>>vlist[1:3]=["bit", "computer"]

>>>vlist

[0, 'bit', 'computer', 3, 4]
```

# 列表类型的操作

当使用一个列表改变另一个列表值时，Python不要求两个列表长度一样，但遵循“多增少减”的原则，例子如下：

```
>>>vlist[1:3]=["new_bit", "new_computer", 123]
>>>vlist
[0, 'new_bit', 'new_computer', 123, 'python', 4]
>>>vlist[1:3]=["fewer"]
>>>vlist
[0, 'fewer', 123, 'python', 4]
```

# 列表类型的操作

与元组一样，列表可以通过for...in语句对其元素进行遍历，基本语法结构如下：

```
for    <任意变量名>  in  <列表名>:  
    语句块
```

# 列表类型的操作

```
>>>for e in vlist:  
    print(e, end=" ")  
0 fewer 123 python 4
```

列表是一个十分灵活的数据结构，它具有处理任意长度、混合类型的能力，并提供了丰富的基础操作符和方法。当程序需要使用组合数据类型管理批量数据时，请尽量使用列表类型。

# 小 结

- 编写简单函数
- 生成自制的模块
- 列表数据类型