第3章 控制语句与自定义函数

学习目标

- 布尔变量的概念与使用
- 使用if 、else、elif 语句实现条件判断
- for、while、break、continue循环语句的使用
- 了解zip 和enumerate函数，以及不同的迭代方式
- 使用列表推导创建循环列表
- 函数的创建与参数传递方式

3.1.1 条件——布尔变量

- 在Python中，布尔值True用来表示真，False用来表示假。
- 下列值也会被解释器看作是假：None、()、{}、[]、“”、0等，即所有类型的数字0、空序列、空字典都为假，此外的一切都被解释为真：

```
>>> if (1):
        print('It\'s True.' )


It's True.
>>> bool('python')
True
>>> True == 1
True
>>> False == 0
True
>>> True + False + 42
43
```

3.1.2 条件语句

- if语句
  - if语句是简单的条件语句，如果if后的条件判断为真，则执行冒号后的语句块，若条件为假则不执行：

```
student_ID = {'Alice' : 1230, 'Bob' : 3789}
name = input('enter the name:')
if (name in student_ID):
    print(('%s\'s ID is %s') % (name, student_ID[name]))

enter the name:Alice
Alice's ID is 1230
```

  - 上述代码中没有“>>>”，因为当代码中使用条件、循环等语句时，代码会变长，为了更好地编辑代码，最好使用集成开发环境IDE或是保存为.py文件进行管理。

3.1.2 条件语句

- if else语句
  - else语句用于对if语句中判断表达式值为假时进行操作的说明，不能脱离if语句单独存在：

```
student_ID = {'Alice' : 1230, 'Bob' : 3789}
name = input('enter the name:')
if (name in student_ID):
    print(('%s\'s ID is %s') % (name, student_ID[name]))
else:
    print('Name is invalid.')

enter the name:Carol
Name is invalid.
```

3.1.2 条件语句

- if elif语句
  - 有时进行一次条件判断并不够，elif语句就可以用于进行多次判断，必须与if语句同时使用：

```
grade = int(input('enter your grades:'))
if (100 >= grade > 85):
    print('Congratulations! Your grades are in the top 30%.')
elif (85 >= grade > 70):
    print('Your grades are above average.')
elif (70 >= grade >= 0):
    print('Your grades are below average.')
else:
    print('error')
```

```
enter your grades:72
Your grades are above average.
```

3.1.2 条件语句

- 嵌套
  - 有时在if语句判断为真时也要继续进行条件判断，将条件更加细化，这就可以通过嵌套的形式来实现：

```
grade = int(input('enter your grades:'))
if (100 >= grade >= 0):
    if (grade > 85):
        print('Congratulations! Your grades are in the top 30%.')
    elif (grade > 70):
        print('Your grades are above average.')
    else:
        print('Your grades are below average.')
else:
    print('error')

enter your grades:72
Your grades are above average.
```

3.1.2 条件语句

- assert语句
  - 有时要求某些条件必须为真，此时可使用关键字assert进行断言，相当于在程序中设置检查点：

```
>>> grade = -1
>>> assert (100 >= grade >= 0)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    assert (100 >= grade >= 0)
AssertionError
```

3.2.1 循环——循环语句

- 除了条件判断，有时也要将同一段代码重复执行多次，就需要用到循环。
- while循环
  - while循环通过判断while后的表达式的值的真假来控制循环的进行与结束，只要表达式的值为真，就执行冒号后的语句块，若表达式的值为假，则循环停止：

```
>>> i = 0
>>> while (2 * i + 1 < 10):
        print(2 * i + 1)
        i += 1


1
3
5
7
9
```

3.2.1 循环——循环语句

- for循环
  - 在对序列或其他可迭代对象中每个元素循环执行相同代码时，for循环更合适：

```
>>> words = ['this', 'is', 'a', 'book']
>>> for word in words:
        print(word)


this
is
a
book

>>> new=[]
>>> for i in 'python':
        new.append(i)


>>> print(new)
['p', 'y', 't', 'h', 'o', 'n']
```

3.2.1 循环——循环语句

- for循环
  - range函数为Python内置的范围函数，常用于迭代某范围的数字，它包含下限0，但不包含上限，默认步长为1，下限、上限、步长可依次设置值：

```
>>> for i in range(10):
        print(i)


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

```
>>> for i in range(1,10,2):
        print(i)


1
3
5
7
9
```

3.2.1 循环——循环语句

- for循环
  - for语句也可以用在对字典中键或值的循环中：

```
>>> consumer = {'Bob' : [39, 'services', 'university.degree'],'Alice' : [30, 'blue-
collar', 'basic.9y'],'Carol' : [25, 'services', 'basic.9y']}
>>> job=[]
>>> for key in consumer:
        job.append(consumer[key][1])


>>> print(job)
['services', 'blue-collar', 'services']
```

  - 由于字典中的项没有顺序的概念，即在循环中每一项都会被处理，但处理的顺序不确定，所以对于顺序很重要或有特殊要求的情况，可以结合列表进行处理。

3.2.1 循环——循环语句

- 循环的嵌套
  - 循环也可以进行嵌套：

```
>>> consumer = {'Bob' : ['yes', 39, 'university.degree', 'services'],'Alice' : ['bl
ue-collar', 'yes', 30, 'basic.9y'],'Carol' : ['services', 'no', 'basic.9y', 25]}
>>> bought = []
>>> for key in consumer:
        for i in consumer[key]:
            if (i == 'yes'):
                bought.append((key,consumer[key]))


>>> print(bought)
[('Bob', ['yes', 39, 'university.degree', 'services']), ('Alice', ['blue-collar', '
yes', 30, 'basic.9y'])]
```

3.2.1 循环——循环语句

- break与continue结束循环
  - break语句结束整个循环，而continue语句只结束当前的迭代、进行下一轮迭代，但不结束整个循环：

```
>>> for i in range(5):
        if i == 2:
            break
        print(i)


0
1
```

```
>>> for i in range(5):
        if i == 2:
            continue
        print(i)


0
1
3
4
```

3.2.1 循环——循环语句

- for else语句
  - for else语句指在进行完for循环后，执行else语句中的代码块：

```
x=[1,2,3,4]
for i in x:
    print(i)
else:
    print('no bug')
```

```
1
2
3
4
no bug
```

3.2.1 循环——循环语句

- for else语句
  - for else语句与break语句联合使用时会有更大的作用：

```
x = [1, 2, 'bug', 4]
for i in x:
    if i == 'bug':
        print('find bug')
        break
    else:
        print('no bug')
```

```
no bug
no bug
find bug
```

3.2.1 循环——循环语句

- while True/break语句
  - while True的条件一直为真，循环会一直进行，通过添加if/break语句可以在满足条件时结束循环：

```
employee = []
while True:
    new = input('添加员工姓名（输入end停止）:')
    if (new == 'end'):
        break
    employee.append(new)
print(employee)

添加员工姓名（输入end停止）:Alice
添加员工姓名（输入end停止）:Bob
添加员工姓名（输入end停止）:end
['Alice', 'Bob']
```

3.2.2 迭代方式

- 并行迭代
  - 并行迭代指同时循环迭代两个甚至多个序列：

```
>>> names = ['Alice', 'Bob', 'Carol']
>>> ages = [30, 39, 25]
>>> for i in range(len(names)):
        print('%s is %s years old.' % (names[i], ages[i]))


Alice is 30 years old.
Bob is 39 years old.
Carol is 25 years old.
```

3.2.2 迭代方式

- 并行迭代
  - zip函数也可以用于并行迭代，该函数将两个或多个序列“压缩”在一起，返回一个由元组构成的列表：

```
>>> names = ['Alice', 'Bob', 'Carol']
>>> ages = [30, 39, 25]
>>> jobs = ['blue-collar', 'services', 'services']
>>> for name, age, job in zip(names, ages, jobs):
        print('%s:%s years old, %s' % (name, age, job))


Alice:30 years old, blue-collar
Bob:39 years old, services
Carol:25 years old, services
```

  - 对于不等长的多个序列，zip以最短的序列的长度为准进行匹配。

3.2.2 迭代方式

- 编号迭代
  - 有时在迭代时需要获取当前迭代对象的索引:

```
>>> employee = [('Alice', 'A部门', 4000), ('Bob', 'A部门', 3500),('Carol', 'B部门',
5000), ('Dave', 'A部门', 2500)]
>>> A_employee = []
>>> for i in employee:
        if (i[1] == 'A部门'):
                A_employee.append((employee.index(i), i[0]))

>>> print(A_employee)
[(0, 'Alice'), (1, 'Bob'), (3, 'Dave')]
```

3.2.2 迭代方式

- 编号迭代
  - 也可以使用内置的enumerate函数：

```
>>> employee = [('Alice', 'A部门', 4000), ('Bob', 'A部门', 3500),('Carol', 'B部门',
5000), ('Dave', 'A部门', 2500)]
>>> A_employee = []
>>> for index,i in enumerate(employee):
        if (i[1] == 'A部门'):
                A_employee.append((index, i[0]))


>>> print(A_employee)
[(0, 'Alice'), (1, 'Bob'), (3, 'Dave')]
```

3.2.2 迭代方式

- 翻转与排序迭代
  - 在迭代中可以使用reversed和sorted函数实现序列的前后翻转或排序，它们修改对象本身，而是返回翻转或排序后的列表或可迭代对象：

```
>>> sorted('python')
['h', 'n', 'o', 'p', 't', 'y']
>>> reversed('python')
<reversed object at 0x032FE0B0>
>>> list(reversed('python'))
['n', 'o', 'h', 't', 'y', 'p']
```

3.2.3 排序

- 常用的排序算法有冒泡排序、快速排序、直接排序等，这里以冒泡排序（由小至大，共n个元素）为例：
  - 从序列最左端的第一个元素开始，将其与相邻元素进行比较，若相邻元素大于该元素则保持不变，同时操作元素替换为相邻元素，否则交换两者的位置。依次进行比较与调整，第一轮结束后，列表中最大的元素位于最右端。
  - 对序列前n-1个元素在进行①中的操作，整个冒泡排序共进行n-1轮比较。

```
def bubble_sort(L):
    length = len(L)
    for x in range(1, length):
        for i in range(0, length-x):
            if L[i] > L[i+1]:
                temp = L[i]
                L[i] = L[i+1]
                L[i+1] = temp
    return L
number=[2, 9, 3, 17, 8, 5]
bubble_sort(number)
print(number)
```

```
[2, 3, 5, 8, 9, 17]
```

3.3.1 列表推导式

- 列表推导式可遍历for后的可迭代对象，并对每个元素按照for前的表达式进行运算，以运算结果为新列表的元素，并最终返回新列表：

```
>>> [x ** 2 for x in range(5)]
[0, 1, 4, 9, 16]
```

- 列表推导式中也可以添加条件语句或进行循环嵌套：

```
>>> [x ** 2 for x in range(10) if (x % 2 == 0)]
[0, 4, 16, 36, 64]

>>> [x * y for x in range(1, 3) for y in range(1, 4)]
[1, 2, 3, 2, 4, 6]
```

3.3.2 其他语句

- pass语句
  - pass语句可以在的代码中做占位符使用。
  - 如在if语句条件筛选时，有时会想要仅对某个条件不执行任何操作。但在Python中空代码块是非法的，解决这一问题可以在空代码块的位置加上pass语句：

```
name = 'Alice'
if (name == 'Bob'):
    print('Welcome!')
elif (name == 'Alice'):
    pass
else:
    print('Access Denied')
```

3.3.2 其他语句

- del语句
  - del语句用于移除对象：

```
>>> x = [1, 2, 3, 'python']
>>> del x[1]
>>> x
[1, 3, 'python']
```

- eval语句
  - eval语句用于计算以字符串形式书写的python表达式：

```
>>> eval('list(\'python\')')
['p', 'y', 't', 'h', 'o', 'n']
```

3.4.1 函数的创建

- Python中除了内置函数外，也可以自定义函数。
- def语句
  - 用于定义函数：

```
>>> def hello(name):
        print('Hello,%s!' % name)
        return 0
```

  - 可以像调用内置函数一样调用自定义函数：

```
>>> hello('python')
Hello,python!
0
```

3.4.1 函数的创建

- return语句
  - 用于从函数中返回值：

```
>>> def fibs(n):
        results = [0, 1]
        for i in range(n - 2):
            results.append(results[-2] + results[-1])
        return results

>>> fibs(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

  - 若return语句后不加返回值，则该函数的返回值为None，此时return语句只是起到结束函数的作用。

3.4.1 函数的创建

- return语句
  - 函数也可以返回多个值，若用一个变量去接收返回值，则该变量的类型就是元组：

```
>>> def fibs(n):
        results = [0, 1]
        for i in range(n - 2):
            results.append(results[-2] + results[-1])
        return results, results[-1]

>>> x1 = fibs(10)
>>> print(x1)
([0, 1, 1, 2, 3, 5, 8, 13, 21, 34], 34)
```

  - 若用与元素数量相等的变量去接收返回值，则各变量与元组中的各元素一一匹配：

```
>>> x2, x3 = fibs(10)
>>> print(x2, x3)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34] 34
```

  - 也可以选择性接收返回值：

```
>>> x4 = fibs(10)[1]
>>> print(x4)
34
```

3.4.1 函数的创建

- 记录函数
  - 可以通过在函数中添加文档字符串进行说明，文档字符串会作为函数的一部分被储存：

```
>>> def fibs(n):
        'Calculate the Fibonacci sequence of length x'
        results = [0, 1]
        for i in range(n - 2):
                results.append(results[-2] + results[-1])
        return results
```

  - 正常调用函数时并不会显示文档，需要通过以下方式访问：

```
>>> print(fibs.__doc__)
Calculate the Fibonacci sequence of length x
```

  - __doc__ 是函数的属性，doc两边为双下划线，表明它是一个特殊属性。

3.4.2 参数

- def语句中函数名后括号中的变量是函数的形式参数，调用函数时提供的值为实际参数，也叫参数。
- 参数可以来自于函数外的变量。对于字符串、数字、元组等不可变的数据类型来说，调用函数时函数获得的参数相当于外界的变量将值输入函数中，函数内部对参数进行的操作并不会影响外界的变量：

```
>>> def change(x, y):
        x = y
        return x

>>> m = 5
>>> n = 10
>>> m_change = change(m, n)
>>> print(m, m_change)
5 10
```

3.4.2 参数

- 对于列表等可变数据类型来说，调用函数时传入的参数并非值本身，而是相应的储存地址，所以在函数内部改变了参数时，相应的函数外的变量的值也会随之变化：

```
>>> def change(x, y):
        x[0] = y[0]
        return x

>>> m = [5, 10]
>>> n = [10, 10]
>>> m_change = change(m, n)
>>> print(m, m_change)
[10, 10] [10, 10]
```

3.4.2 参数

- 关键字参数
  - 前面提到的参数为位置参数，它们通过位置相互区分：

```
>>> def introduce1(name, age):
        return '%s is %s years old.' % (name, age)

>>> def introduce2(age, name):
        return '%s is %s years old.' % (name, age)

>>> print(introduce1('Alice', 25))
Alice is 25 years old.
>>> print(introduce2(25, 'Alice'))
Alice is 25 years old.
```

  - 关键字参数通过参数名来区分，与顺序无关，但应注意参数名与值的对应：

```
>>> print(introduce1(age = 25, name = 'Alice'))
Alice is 25 years old.
```

3.4.2 参数

- 关键字参数
  - 关键字参数可以在定义函数时给定默认值，调用函数时提供参数就不是必须条件了：

```
>>> def introduce(name = 'John', age = 20):
        return '%s is %s years old.' % (name, age)

>>> print(introduce())
John is 20 years old.
>>> print(introduce(age = 25))
John is 25 years old.
```

  - 关键字参数与位置参数可以联合使用，但必须把位置参数放在前面：

```
>>> def introduce(name, age, more = 'That\'s all I know.'):
        return '%s is %s years old. %s' % (name, age, more)

>>> print(introduce('John', 20))
John is 20 years old. That's all I know.
```

3.4.2 参数

- 收集参数
  - 通过在参数前面加*，可以实现接收任意多个参数、并将所有值“收集”在一个元组中：

```
>>> def func(*x):
        return x

>>> print(func(1, 2, 3))
(1, 2, 3)
```

  - 这种带星号的参数可以与普通参数联合使用，此时需要将普通参数放在前面：

```
>>> def func(first, *x):
        return x

>>> print(func(1, 2, 3, 4))
(2, 3, 4)
```

3.4.2 参数

- 收集参数
  - 若普通参数first后不再提供任何供收集的参数，则* x为空元组：

```
>>> print(func(1))
()
```

  - *也可以执行相反的操作，在调用函数时将元组中的每个元素依序作为函数的参数：

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

>>> a = (1, 2)
>>> print(add(*a))
3
```

3.4.3 作用域

- 调用函数时，新的命名空间就创建了，函数内代码块的操作均在该命名空间中进行，不影响函数外的变量。相比于函数外的作用域，该命名空间为内部作用域，函数内的变量为局部变量（local variable）。
- 在函数中对与全局变量名相同的局部变量操作不会影响全局变量：

```
>>> x = 5
>>> def func(a):
        x = a + 5
        return x

>>> print(func(x))
10
>>> print(x)
5
```

3.4.4 递归

- 递归指的就是函数对自身的调用。
- 如定义一个求参数n的阶乘的函数，可以使用循环或使用递归实现：

```
>>> def factorial1(n):
        results = 1
        for i in range(1, n+1):
                results *= i
        return results

>>> print(factorial1(5))
120
```

```
>>> def factorial2(n):
        results = 1
        if (n == 1):
                return results
        else:
                results = n * factorial2(n-1)
        return results

>>> print(factorial1(5))
120
```

3.4.4 递归

- 二分查找法
  - 二分查找的原理是将一个有序序列一分为二，判断所查找的元素在哪一部分，再将该部分继续分为平均的两部分，重复这一查找操作，直到找到目标元素：

```
def search(sequence, number, left, right):
    if (left == right):
        if (number == sequence[right]):
            return right
        else:
            return "Not Found!"
    else:
        middle = (left + right) // 2
    if (number > sequence[middle]):
        return search(sequence, number, middle+1, right)
    else:
        return search(sequence, number, left, middle)
sequence = [7, 5, 9, 2, 6, 11, 8, 20]
sequence.sort()
print(search(sequence, 11, 0, len(sequence)-1))
print(search(sequence, 12, 0, len(sequence)-1))
```

```
6
Not Found!
```

3.4.4 递归

- 快速排序
  - 快速排序原理是先任选序列中的一个数作为关键数据将所有比它大的放在它后面，将所有比它小的放在它前面，以这个找到位置的数为分界，分别对前面和后面两个子序列中的第一个元素进行重复操作，如此迭代，完成最终排序：

```
def quick_sort(sequence):
    if len(sequence) < 2:
        return sequence
    else:
        pivot = sequence[0]
        left = [i for i in sequence[1:] if i < pivot]
        right = [i for i in sequence[1:] if i >= pivot]
        return quick_sort(left) + [pivot] + quick_sort(right)

sequence = [7, 5, 9, 2, 6, 11, 8, 20]
print(quick_sort(sequence))
```

```
[2, 5, 6, 7, 8, 9, 11, 20]
```