第3章 数据结构

数据结构研究的内容是如何按一定的逻辑结构把数据组织起来，并选择适当的存储表示方法把逻辑结构组织好的数据存储到计算机的存储器，从而更有效地处理数据，提高数据运算效率。数据的运算是定义在数据的逻辑结构上的操作，一般有以下几种常用运算。

(1) 检索，也称查询、搜索，就是在数据结构里查找满足一定条件的节点。一般是给定一个某字段的值，找具有该字段值的节点。

(2) 插入，也称增加，往数据结构中增加新的节点。

(3) 删除，把指定的节点从数据结构中去掉。

(4) 更新，也称修改，改变指定节点的一个或多个字段的值。

(5) 排序，把节点按某种指定的顺序重新排列，如递增或递减。

总结起来就是查、增、删、改和排序。Python 语言内置对象提供了一些数据结构，也称序列结构，包括列表、元组、字典和集合，如图 3.1 所示。从是否有序角度可将序列分为无序序列和有序序列，字典和集合都属于无序序列，列表和元组属于有序序列；从是否可变的角度可将序列分为不可变序列和可变序列，元组属于不可变序列，字典、集合和列表都属于可变序列。本章主要讨论这些结构是如何有效地组织数据，有哪些重要的方法等。在 Python 中这些数据结构都是对象，因此这里不再称其为运算或者函数，而是按照面向对象程序设计的习惯称其为方法。

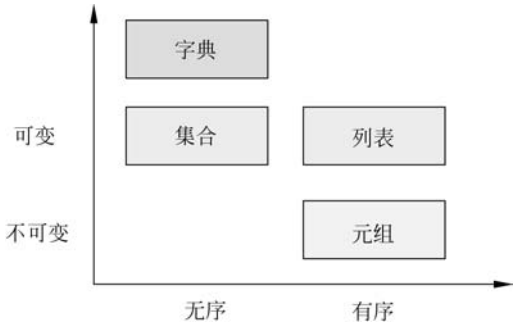


图 3.1 序列分类

3.1 列 表

列表(list)是最重要的 Python 内置对象之一，是包含若干元素的有序连续内存空间。当列表增加或删除元素时，列表对象自动进行内存的扩展或收缩，从而保证相邻元素之间没

有缝隙。Python 列表的这个内存自动管理功能可以大幅度减少程序员的负担，但插入和删除非尾部元素时涉及列表中大量元素的移动，会严重影响效率。

在非尾部位置插入和删除元素时会改变该位置后面的元素在列表中的索引，这对于某些操作可能会导致意外的错误结果。所以，除非确实有必要，否则应尽量从列表尾部进行元素的追加与删除操作。

在形式上，列表的所有元素放在一对方括号[]中，相邻元素之间使用逗号分隔。在 Python 中，同一个列表中元素的数据类型可以各不相同，可以同时包含整数、实数、字符串等基本类型的元素，也可以包含列表、元组、字典、集合、函数以及其他任意对象。如果只有一对方括号而没有任何元素则表示空列表。列表的示例参见程序 3.1。

```
#程序 3.1  列表
[10, 20, 30, 40]                          #整数列表
['white', 'red', 'pink']                  #字符串列表
['python', 3.14, 15, [11, 12]]            #不同类型数据列表
[['lst1', 21,7], ['lst2', 200,3]]         #嵌套列表
[{3}, {4:6}, (6, 7, 8)]                   #序列结构列表
```

Python 采用基于值的自动内存管理模式，变量并不直接存储值，而是存储值的引用或内存地址，这也是 Python 中变量可以随时改变类型的重要原因。同理，Python 列表中的元素也是值的引用，所以列表中各元素可以是不同类型的数据。

需要注意的是，列表的功能虽然非常强大，但是负担也比较重，开销较大，在实际开发中，最好根据实际的问题选择一种合适的数据类型，要尽量避免过多使用列表。

3.1.1 列表操作

1. 列表创建与删除

使用“=”直接将一个列表赋值给变量即可创建列表对象，示例参见程序 3.2。

```
#程序 3.2  列表变量
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

也可以使用 list()函数把元组、range 对象、字符串、字典、集合或其他可迭代对象转换为列表。当一个列表不再使用时，可以使用 del 命令将其删除，这一点适用于所有类型的 Python 对象。列表操作示例参见程序 3.3。

```
#程序 3.3  列表操作
list((3,5,7,9,11))                    #将元组转换为列表
list(range(1, 10, 2))                 #将 range 对象转换为列表
list('hello world')                   #将字符串转换为列表
list({3,7,5})                         #将集合转换为列表
list({'a':3, 'b':9, 'c':6})           #将字典的"键"转换为列表
list({'a':3, 'b':9, 'c':6}.items())   #将字典的"键:值"对转换为列表
x = list()                            #创建空列表
```

```
9   x = [1, 2, 3]
10  del x                          #删除列表对象
```

2. 列表元素访问

创建列表之后，可以使用整数作为下标来访问其中的元素。其中，0 表示第 1 个元素，1 表示第 2 个元素，2 表示第 3 个元素，以此类推；列表还支持使用负整数作为下标，其中－1 表示最后 1 个元素，－2 表示倒数第 2 个元素，－3 表示倒数第 3 个元素，以此类推。这和第 2 章所讲的字符串索引是类似的，示例参见程序 3.4。

```
1   #程序 3.4  列表元素访问
2   x = list('hello')                  #字符串转换成列表
3   print(x[0])                        #打印列表第一个元素
4   print(x[1:3])
5   print(x[-1])                       #打印列表最后一个元素
```

输出结果：

```
h
['e', 'l']
o
```

3.1.2 列表常用方法

列表的常用方法如表 3.1 所示，主要分为向列表插入新的元素、更新列表中元素的值、删除列表中的元素。

表 3.1 列表的常用方法及描述

方　法	描　述
append(x)	将 x 追加至列表尾部
extend(L)	将列表 L 中所有元素追加至列表尾部
insert(index, x)	在列表 index 位置处插入 x，该位置后面的所有元素后移并且在列表中的索引加 1。如果 index 为正数且大于列表长度则在列表尾部追加 x，如果 index 为负数且小于列表长度的相反数则在列表头部插入元素 x
remove(x)	在列表中删除第一个值为 x 的元素，该元素之后所有元素前移并且索引减 1，如果列表中不存在 x 则抛出异常
pop([index])	删除并返回列表中下标为 index 的元素，如果不指定 index 则默认为－1，弹出最后一个元素；如果弹出中间位置的元素则后面的元素索引减 1；如果 index 不是[－L，L]区间上的整数则抛出异常
clear()	清空列表，删除列表中所有元素，保留列表对象
index(x)	返回列表中第一个值为 x 的元素的索引，若不存在值为 x 的元素则抛出异常
count(x)	返回 x 在列表中的出现次数
reverse()	对列表所有元素进行原地逆序，首尾交换
sort(key=None, reverse=False)	对列表中的元素进行原地排序，key 用来指定排序规则，reverse 为 False 表示升序，True 表示降序
copy()	返回列表的浅复制

1. append()、insert()和 extend()方法

append()方法用于向列表尾部追加一个元素，insert()方法用于向列表任意指定位置插入一个元素，extend()方法用于将另一个列表中的所有元素追加至当前列表的尾部。这 3 个方法都属于原地操作，不影响列表对象在内存中的起始地址，示例参见程序 3.5。

```
#程序 3.5  列表 append()、insert()、extend()方法
x = [1, 2, 3]
print(id(x))                    #查看对象的内存地址
x.append(4)                     #在尾部追加元素
x.insert(0, 10)                 #在指定位置插入元素
x.extend([5, 6, 7])             #在尾部追加多个元素
print(x)
print(id(x))                    #列表在内存中的地址不变
```

输出结果：

```
2616696590792
[10, 1, 2, 3, 4, 5, 6, 7]
2616696590792
```

2. pop()、remove()和 clear()方法

pop()方法用于删除并返回指定位置(默认是最后一个)上的元素；remove()方法用于删除列表中第一个值与指定值相等的元素；clear()方法用于清空列表中的所有元素。这 3 个方法也属于原地操作，示例参见程序 3.6。还可以使用 del 命令删除列表中指定位置的元素，同样也属于原地操作。

```
#程序 3.6  列表 pop()、remove()、clear()方法
x = [1, 2, 3, 4, 5, 6, 7]
x.pop()                         #弹出并返回尾部元素
x.pop(0)                        #弹出并返回指定位置的元素
print(x)
x.clear()                       #删除所有元素
x = [1, 2, 1, 1, 10]
x.remove(2)                     #删除首个值为 2 的元素
del x[3]                        #删除指定位置上的元素
print(x)
```

输出结果：

```
[2, 3, 4, 5, 6]
[1, 1, 1]
```

3. count()和 index()方法

列表方法 count()用于返回列表中指定元素出现的次数；index()方法用于返回指定元素在列表中首次出现的位置，如果该元素不在列表中则抛出异常，示例参见程序 3.7。

```
#程序 3.7  列表 count()、index()方法
x = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
print(x.count(3))                    #元素 3 在列表 x 中的出现次数
print(x.count(5))                    #不存在,返回 0
print(x.index(2))                    #元素 2 在列表 x 中首次出现的索引
print(x.index(5))                    #列表 x 中没有 5,抛出异常
```

输出结果：

```
3
0
1
ValueError: 5 is not in list
```

4. sort()和 reverse()方法

列表对象的 sort()方法用于按照指定的规则对所有元素进行排序；reverse()方法用于将列表所有元素逆序或翻转,示例参见程序 3.8。

```
#程序 3.8  列表 sort()、reverse()方法
import random
x = list(range(11))                          #包含 11 个整数的列表
random.shuffle(x)                            #把列表 x 中的元素随机乱序
print('随机排序:',x)
x.sort()                                     #按默认规则排序
print('默认排序:',x)
x.reverse()                                  #把所有元素翻转或逆序
print('逆序排序:',x)
#按转换成字符串以后的长#度,降序排列
x.sort(key = lambda item:len(str(item)), reverse = True)
print('指定关键字排序:',x)
```

输出结果：

```
随机排序: [4, 2, 7, 10, 8, 3, 0, 6, 5, 1, 9]
默认排序: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
逆序排序: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
指定关键字排序: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

3.1.3 列表支持的运算符

1. 加法

加法运算符“+”也可以实现列表增加元素的目的,但不属于原地操作,而是返回新列表,涉及大量元素的复制,效率非常低。使用复合赋值运算符“+=”实现列表追加元素时属于原地操作,与 append()方法一样高效,示例参见程序 3.9。

```
#程序 3.9  列表 + 运算
x = [1, 2, 3]
print(id(x))
```

```
x = x + [4]                    #连接两个列表
print(id(x))                   #内存地址发生改变
x += [5]                       #为列表追加元素
print(x)
print(id(x))                   #内存地址不变
```

输出结果：

```
2431893524936
2431894803656
[1, 2, 3, 4, 5]
2431894803656
```

2. 乘法

乘法运算符“*”可以用于列表和整数相乘，表示序列重复，返回新列表。运算符“*=”也可以用于列表元素重复，属于原地操作。列表*运算示例参见程序 3.10。

```
#程序 3.10  列表*运算
x = [1, 2, 3]
print(id(x))
x = x * 2                              #元素重复，返回新列表
print(x)
print(id(x))                           #地址发生改变
x *= 2                                 #元素重复，原地进行
print(x)
print(id(x))                           #地址不变
```

输出结果：

```
2050672185800
[1, 2, 3, 1, 2, 3]
2050672186312
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
2050672186312
```

3. 成员测试运算符 in

成员测试运算符 in 可用于测试列表中是否包含某个元素，查询时间随着列表长度的增加而线性增加，而同样的操作对于集合而言则是常数级的，示例参见程序 3.11。

```
#程序 3.11  列表 in 运算
print(3 in [1, 2, 3])
print(3 in [1, 2, '3'])
```

输出结果：

```
True
False
```

3.1.4 内置函数对列表的操作

(1) max()函数和 min()函数分别用于返回列表中所有元素中的最大值和最小值。

(2) sum()函数用于返回列表中的所有元素之和。

(3) len()函数用于返回列表中的元素个数,zip()函数用于将多个列表中的元素重新组合为元组并返回包含这些元组的 zip 对象。

(4) enumerate()函数返回包含若干下标和值的迭代对象。

(5) map()函数把函数映射到列表上的每个元素,filter()函数根据指定函数的返回值对列表元素进行过滤。

(6) all()函数用来测试列表中是否所有元素都等价于 True,any()函数用来测试列表中是否有等价于 True 的元素。

(7) 标准库 functools 中的 reduce()函数以及标准库 itertools 中的 compress()、groupby()、dropwhile()等大量函数也可以对列表进行操作。

内置函数对列表的操作参见程序 3.12。

```
#程序 3.12  内置函数对列表的操作
import random
x = list(range(5))                          #生成列表
random.seed(100)                            #固定种子值可以让每次序列相同
random.shuffle(x)                           #打乱列表中元素顺序
print('x = ',x)
print('x 的最大值:',max(x))                  #返回最大值
print('x 的最小值:',min(x))                  #返回最小值
print('x 的和:',sum(x))                      #返回和
print('x 的长度:',len(x))                    #返回长度
y = filter(lambda x:x % 2 == 0,x)           #保留偶数
print('过滤 x 的奇数,保留偶数:',list(y))
print('两个序列组合:',list(zip(x,[1] * 5)))  #多列表元素重新组合
z = map(lambda x:x + 10,x)                  #通过 lambda()函数实现每个元素加 10
print('z = ',list(z))
print('枚举列表元素:',list(enumerate(x)))    #枚举列表元素
```

输出结果:

```
x = [2, 0, 4, 3, 1]
x 的最大值: 4
x 的最小值: 0
x 的和: 10
x 的长度: 5
过滤 x 的奇数,保留偶数: [2, 0, 4]
两个序列组合: [(2, 1), (0, 1), (4, 1), (3, 1), (1, 1)]
z = [12, 10, 14, 13, 11]
枚举列表元素: [(0, 2), (1, 0), (2, 4), (3, 3), (4, 1)]
```

3.1.5 列表推导式

列表推导式使用非常简洁的方式快速生成满足特定需求的列表,代码具有非常强的可读性。列表推导式语法形式为:

```
[expression for expr1 in sequence1 if condition1
             for expr2 in sequence2 if condition2
             for expr3 in sequence3 if condition3
             …
             for exprN in sequenceN if conditionN]
```

列表推导式在逻辑上等价于一个循环语句，只是形式上更加简洁，示例参见程序 3.13。

```
1  #程序 3.13  列表推导式
2  lst1 = [x * x for x in range(10)]
3  print(lst1)
4  lst2 = []
5  for x in range(10):                    #和第 2 行的列表推导式等价
6      lst2.append(x ** 2)                #列表追加元素
7  print(lst2)
```

输出结果：

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

1. 实现嵌套列表的平铺

程序 3.14 中列表推导式中有两个循环。第一个循环可以看作外循环，执行得慢；第二个循环可以看作内循环，执行得快。

```
1  #程序 3.14  带二重循环的列表推导式
2  vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
3  print([num for elem in vec for num in elem])
4  result = []
5  for elem in vec:                       #和第 3 行的列表推导式等价
6      for num in elem:
7          result.append(num)
8  print(result)
```

输出结果：

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. 使用列表推导式实现矩阵转置

使用列表推导式实现矩阵转置，示例参见程序 3.15。

```
1  #程序 3.15  矩阵转置
2  matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
3  print(matrix)
4  print([[row[i] for row in matrix] for i in range(4)])      #列表推导式实现矩阵转置
```

输出结果：

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

3.1.6 切片操作

在形式上，切片使用两个冒号分隔的 3 个数字来完成，[start:end:step]，

第一个数字 start 表示切片开始位置，默认为 0；

第二个数字 end 表示切片截止（但不包含）位置（默认为列表长度）；

第三个数字 step 表示切片的步长（默认为 1）。

当 start 为 0 时可以省略，当 end 为列表长度时可以省略，当 step 为 1 时可以省略，省略步长时还可以同时省略最后一个冒号。

当 step 为负整数时，表示反向切片，这时 start 应该在 end 的右侧。

1. 使用切片获取列表部分元素

使用切片可以返回列表中部分元素组成的新列表。与使用索引作为下标访问列表元素的方法不同，切片操作不会因为下标越界而抛出异常，而是简单地在列表尾部截断或者返回一个空列表，代码具有更强的健壮性，示例参见程序 3.16。

```
1  #程序 3.16  列表切片
2  lst = list(range(10))
3  print(lst[::])                #返回包含原列表中所有元素的新列表
4  print(lst[::-1])              #返回包含原列表中所有元素的逆序列表
5  print(lst[::2])               #隔一个取一个，获取偶数位置的元素
6  print(lst[1::2])              #隔一个取一个，获取奇数位置的元素
7  print(lst[3:6])               #指定切片的开始和结束位置
```

输出结果：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[0, 2, 4, 6, 8]
[1, 3, 5, 7, 9]
[3, 4, 5]
```

2. 使用切片为列表增加元素

可以使用切片操作在列表任意位置插入新元素，不影响列表对象的内存地址，属于原地操作，示例参见程序 3.17。

```
1  #程序 3.17  列表切片增加元素
2  lst = [3, 5, 7]
3  lst[len(lst):] = [9]          #在列表尾部增加元素
4  print(lst)
5  lst[:0] = [1, 2]              #在列表头部插入多个元素
6  print(lst)
7  lst[3:3] = [4]                #在列表中间位置插入元素
8  print(lst)
```

输出结果：

```
[3, 5, 7, 9]
[1, 2, 3, 5, 7, 9]
[1, 2, 3, 4, 5, 7, 9]
```


3. 使用切片替换和修改列表中的元素

使用切片替换和修改列表中的元素，示例参见程序 3.18。

```
1  #程序 3.18  列表切片实现修改元素
2  lst = [3, 5, 7, 9]
3  lst[:3] = [1, 2, 3]              #替换列表元素,等号两边的列表长度相等
4  print(lst)
5  lst[3:] = [4, 5, 6]              #切片连续,等号两边的列表长度可以不相等
6  print(lst)
7  lst[::2] = [0] * 3               #隔一个修改一个
8  print(lst)
9  lst[::2] = ['a', 'b', 'c']       #隔一个修改一个
10 print(lst)
```

输出结果：

```
[1, 2, 3, 9]
[1, 2, 3, 4, 5, 6]
[0, 2, 0, 4, 0, 6]
['a', 2, 'b', 4, 'c', 6]
```

4. 使用切片删除列表中的元素

使用切片删除列表中的元素，示例参见程序 3.19。

```
1  #程序 3.19  列表切片实现删除元素
2  lst = [3, 5, 7, 9]
3  lst[:3] = []                     #删除列表中前 3 个元素
4  print(lst)
```

输出结果：

```
[9]
```

3.2 元　　组

列表的功能虽然很强大，但负担也很重，在很大程度上影响了运行效率。有时候并不需要那么多功能，很希望有个轻量级的列表，元组(tuple)正是这样一种类型。

从形式上，元组的所有元素放在一对圆括号中，元素之间使用逗号分隔，如果元组中只有一个元素则必须在最后增加一个逗号。

3.2.1 元组创建与元素访问

元组和列表有一些相似的地方，支持使用下标访问其元素，支持双向索引等，但与列表不同的是元组是不可变序列，一旦定义，不能改变元组元素的值，示例参见程序 3.20。

```
#程序 3.20  元组创建与访问
x = ()                                  #创建空元组
x = tuple()                             #创建空元组
tuple(range(5))                         #将其他迭代对象转换为元组
x = (1, 2, 3)                           #直接把元组赋值给一个变量
print('x 的类型为:',type(x))             #使用 type()函数查看变量类型
print('x 的第一个元素为:',x[0])          #元组支持使用下标访问特定位置的元素
print('x 的最后一个元素为:',x[-1])       #最后一个元素,元组也支持双向索引
x[1] = 4                                #元组是不可变的,抛出异常
x = (3)                                 #这和 x = 3 是一样的
x = (3,)                                #如果元组中只有一个元素,必须在后面多写一个逗号
```

输出结果：

```
x 的类型为: <class 'tuple'>
x 的第一个元素为: 1
x 的最后一个元素为: 3
TypeError: 'tuple' object does not support item assignment
```

3.2.2 元组与列表的比较

列表和元组都属于有序序列，都支持使用双向索引访问其中的元素，以及使用 count()方法统计指定元素的出现次数和 index()方法获取指定元素的索引，len()、map()、filter()等大量内置函数和“+”“+=”“in”等运算符也都可以作用于列表和元组。

元组属于不可变(immutable)序列，不可以直接修改元组中元素的值，也无法为元组增加或删除元素。

元组没有提供 append()、extend()和 insert()等方法，无法向元组中添加元素；同样，元组也没有 remove()和 pop()方法，也不支持对元组元素进行 del 操作，不能从元组中删除元素，而只能使用 del 命令删除整个元组。

元组也支持切片操作，但是只能通过切片访问元组中的元素，而不允许使用切片修改元组中元素的值，也不支持使用切片操作为元组增加或删除元素。

Python 的内部实现对元组做了大量优化，访问速度比列表更快。如果定义了一系列常量值，主要用途仅是对它们进行遍历或其他类似用途，而不需要对其元素进行任何修改，那么一般建议使用元组而不用列表。

元组在内部实现上不允许修改其元素值，从而使得代码更加安全。例如，调用函数时使用元组传递参数可以防止在函数中修改元组，而使用列表则很难保证这一点。

3.2.3 生成器推导式

生成器推导式(generator expression)的用法与列表推导式非常相似，在形式上生成器推导式使用圆括号(parentheses)作为定界符，而不是列表推导式所使用的方括号(square brackets)。

与列表推导式最大的不同是，生成器推导式的结果是一个生成器对象。生成器对象类似于迭代器对象，具有惰性求值的特点，只在需要时生成新元素，比列表推导式具有更高的

效率，空间占用非常少，尤其适合大数据处理的场合。

使用生成器对象的元素时，可以根据需要将其转化为列表或元组，也可以使用生成器对象的__next__()方法或者内置函数 next()进行遍历，或者直接使用 for 循环遍历其中的元素。不管用哪种方法访问其元素，只能从前往后正向访问每个元素，没有任何方法可以再次访问已访问过的元素，也不支持使用下标访问其中的元素。当所有元素访问结束以后，如果需要重新访问其中的元素，必须重新创建该生成器对象，enumerate、filter、map、zip 等其他迭代器对象也具有同样的特点。

(1) 使用生成器对象__next__()方法或内置函数 next()进行遍历，参见程序 3.21。

```
1  #程序 3.21  使用生成器对象遍历列表
2  gen = ((i+2)**2 for i in range(10))    #创建生成器对象
3  print(gen)
4  print(tuple(gen))                      #将生成器对象转换为元组
5  print(list(gen))                       #生成器对象已遍历结束，没有元素了
6  gen = ((i+2)**2 for i in range(10))    #重新创建生成器对象
7  print(gen.__next__())                  #使用生成器对象的__next__()方法获取元素
8  print(next(gen))                       #使用函数 next()获取生成器对象中的元素
```

输出结果：

```
<generator object <genexpr> at 0x000001F1FA4738C8>
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
[]
4
9
```

(2) 使用 for 循环语句直接迭代生成器对象中的元素，参见程序 3.22。

```
1  #程序 3.22  for 循环直接迭代生成器对象中的元素
2  gen = ((i+2)**2 for i in range(10))
3  for item in gen:                #使用循环直接遍历生成器对象中的元素
4      print(item, end=' ')
```

输出结果：

```
4 9 16 25 36 49 64 81 100 121
```

(3) 访问过的元素不再存在，参见程序 3.23。

```
1  #程序 3.23  不可再次访问已经访问过的元素
2  x = filter(None, range(10))      #filter 对象也具有类似的特点
3  print(5 in x)
4  print(2 in x)                    #不可再次访问已访问过的元素
5  x = map(str, range(20))          #map 对象也具有类似的特点
6  print('0' in x)
7  print('0' in x)                   #不可再次访问已访问过的元素
```

输出结果：

```
True
False
True
False
```

3.3 字　　典

字典(dictionary)是包含若干“键:值”元素的无序可变序列,字典中的每个元素包含用冒号分隔开的“键”和“值”两部分,表示一种映射或对应关系,也称关联数组。定义字典时,每个元素的“键”和“值”之间用冒号分隔,不同元素之间用逗号分隔,所有的元素放在一对花括号“{}”中。

字典中元素的“键”可以是 Python 中任意不可变数据,如整数、实数、复数、字符串、元组等类型等可哈希数据,但不能使用列表、集合、字典或其他可变类型作为字典的“键”。字典中的“键”不允许重复,而“值”是可以重复的。

3.3.1 字典创建与删除

使用赋值运算符“=”将一个字典赋值给一个变量即可创建一个字典变量,也可以使用内置类 dict 以不同形式创建字典,示例参见程序 3.24。

```
1   #程序 3.24  字典结构的创建
2   x = dict()                                  #空字典
3   print(type(x))                              #查看对象类型
4   x = {}                                      #空字典
5   keys = ['a', 'b', 'c', 'd']
6   values = [1, 2, 3, 4]
7   dictionary = dict(zip(keys, values))        #根据已有数据创建字典
8   print(dictionary)
9   dic = dict(name = 'Dong', age = 39)         #以关键参数的形式创建字典
10  print(dic)
11  #以给定内容为"键",创建"值"为空的字典
12  dic = dict.fromkeys(['name', 'age', 'sex'])
13  print(dic)
```

输出结果:

```
<class 'dict'>
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
{'name': 'Dong', 'age': 39}
{'name': None, 'age': None, 'sex': None}
```

3.3.2 字典元素的访问

字典中的每个元素表示一种映射关系或对应关系,根据提供的“键”作为下标就可以访问对应的“值”,如果字典中不存在这个“键”会抛出异常,示例参见程序 3.25。

```
1  #程序 3.25  字典元素的访问
2  dic = {'age': 39, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
3  print(dic['age'])                #指定的"键"存在,返回对应的"值"
4  print(dic['address'])            #指定的"键"不存在,抛出异常
5  xuehao = {'001':'zhangsan','002':'lisi','003':'wangwu'}
6  print(xuehao.get('001'))         #如果字典中存在该"键"则返回对应#的"值"
7  #指定的"键"不存在时返回指定的默#认值
8  print(xuehao.get('address', 'Not Exists.'))
```

输出结果：

```
39
KeyError: 'address'
zhangsan
Not Exists.
```

字典对象提供了一个 get()方法用来返回指定“键”对应的“值”，并且允许指定该键不存在时返回特定的“值”。程序 3.25 第 8 行就是指定要返回 address 键的值，如果该键不存在则返回字符串“Not Exists”。

使用字典对象的 items()方法可以返回字典的键、值对。使用字典对象的 keys()方法可以返回字典的键。使用字典对象的 values()方法可以返回字典的值。

3.3.3 元素添加、修改与删除

当以指定“键”为下标为字典元素赋值时，有以下两种含义。

(1) 若该“键”存在，则表示修改该“键”对应的值。

(2) 若不存在，则表示添加一个新的“键:值”对，也就是添加一个新元素。

字典元素的添加、修改操作，参见程序 3.26。

```
1  #程序 3.26  字典元素的添加、修改操作
2  dic = {'age': 35, 'name': 'Dong', 'sex': 'male'}
3  dic['age'] = 39                  #修改元素值
4  dic['address'] = 'Jiangning'     #添加新元素
5  print(dic)
6  dic = {'age': 37, 'score': 95, 'name': 'Yang'}
7  #修改'age'键的值,同时添加新元素'address': 'Jiangning'
8  dic.update({'address':'Jiangning', 'age':39})
9  print(dic)
```

输出结果：

```
{'age': 39, 'name': 'Yang', 'sex': 'male', 'address': 'Jiangning'}
{'age': 39, 'score': 95, 'name': 'Yang', 'address': 'Jiangning'}
```

使用字典对象的 update()方法可以将另一个字典的“键:值”一次性全部添加到当前字典对象。如果两个字典中存在相同的“键”，则以另一个字典中的“值”为准对当前字典进行更新。

如果需要删除字典中指定的元素，可以使用 del 命令，也可以使用字典对象的 pop()和 popitem()方法弹出并删除指定的元素，参见程序 3.27。

```
#程序 3.27  字典元素的删除
dic = {'age': 37, 'score': 95, 'name': 'Dong', 'sex': 'male'}
del dic['age']                  #删除字典元素
print(dic)
dic.popitem()                   #弹出一个元素,对空字典会抛出异常
print(dic)
dic.pop('name')                 #弹出指定键对应的元素
print(dic)
```

输出结果：

```
{'score': 95, 'name': 'Dong', 'sex': 'male'}
{'score': 95, 'name': 'Dong'}
{'score': 95}
```

3.3.4 标准库 collections 中与字典有关的类

1. OrderedDict 类

Python 内置字典 dict 是无序的，如果需要一个可以记住元素插入顺序的字典，可以使用 collections.OrderedDict，参见程序 3.28。

```
#程序 3.28  有序字典 OrderedDict 类
import collections
x = collections.OrderedDict()       #有序字典
x['c'] = 8
x['a'] = 3
x['b'] = 5
print(x)                            #按插入的顺序输出
```

输出结果：

```
OrderedDict([('c', 8), ('a', 3), ('b', 5)])
```

2. defaultdict 类

标准库 collections 中的默认字典，在访问一个不存在的键时，不会抛出异常，会创建一个新的键，值默认为 0。程序 3.29 中首先利用随机选择函数从 0～9 字符串中选择 100 次，然后利用默认字典统计字符串出现的次数。

```
#程序 3.29  默认字典 defaultdict 类
import string
import random
from collections import defaultdict
x = string.digits                               #取数字
y = [random.choice(x) for i in range(100)]      #从 x 中选择 100 次作为列表元素
z = ''.join(y)                                  #列表中的元素用空格
frequences = defaultdict(int)                   #所有值默认为 0
for item in z:
    frequences[item] += 1                       #修改每个字符的频次
print(frequences.items())
```

输出结果：

```
dict_items([('2', 9), ('3', 8), ('9', 12), ('7', 11), ('8', 12), ('4', 7), ('1', 7), ('0', 16),
('6', 7), ('5', 11)])
```

3. Counter 类

对于频次统计的问题，使用 collections 模块的 Counter 类可以更加快速地实现统计频率的功能，并且能够延伸给出更多的统计信息功能。例如，查找出现次数最多的元素，参见程序 3.30。

```
1  #程序 3.30  计数 Counter 类
2  import random
3  import string
4  from collections import Counter
5  x = string.digits
6  y = [random.choice(x) for i in range(100)]
7  z = ''.join(y)
8  frequences = Counter(z)
9  print(frequences)                    #以字典形式返回，默认以值从大到小排序
10 print(frequences.items())            #返回以项作为元素的列表
11 print(frequences.most_common(1))     #返回出现次数最多的 1 个字符及其频率
12 print(frequences.most_common(3))     #返回出现次数最多的前 3 个字符及其频率
```

输出结果：

```
Counter({'5': 17, '1': 15, '2': 15, '6': 11, '7': 10, '0': 9, '4': 9, '8': 6, '3': 5, '9': 3})
dict_items([('1', 15), ('3', 5), ('5', 17), ('0', 9), ('8', 6), ('2', 15), ('4', 9), ('7', 10),
('6', 11), ('9', 3)])
[('5', 17)]
[('5', 17), ('1', 15), ('2', 15)]
```

3.4 集　　合

集合(set)属于 Python 无序可变序列，使用一对花括号作为定界符，元素之间使用逗号分隔，同一个集合内的每个元素都是唯一的，元素之间不允许重复。

集合中只能包含数字、字符串、元组等不可变类型(或者说可哈希)的数据，而不能包含列表、字典、集合等可变类型的数据。

3.4.1 集合对象的创建与删除

直接将集合赋值给变量即可创建一个集合对象，也可以使用函数 set()函数将列表、元组、字符串、range 对象等其他可迭代对象转换为集合。如果原来的数据中存在重复元素，则在转换为集合的时候只保留一个；如果原序列或迭代对象中有不可哈希的值，则无法转换成为集合，抛出异常。集合对象的创建与删除，参见程序 3.31。

```
#程序 3.31  集合的创建与删除
x = set()                                    #空集合
s = {3, 5}                                   #创建集合对象
print(type(s))
s1 = set(range(8, 14))                       #把 range 对象转换为集合
print(s1)
s2 = set([0, 1, 2, 3, 0, 1, 2, 3, 7, 8])     #转换时自动去掉重复元素
print(s2)
del s1                                       #删除集合 s1
print(s1)                                    #抛出异常
```

输出结果：

```
<class 'set'>
{8, 9, 10, 11, 12, 13}
{0, 1, 2, 3, 7, 8}
NameError: name 's1' is not defined
```

3.4.2 集合操作与运算

1. 集合元素增加与删除

使用集合对象的 add()方法可以增加新元素，如果该元素已存在则忽略该操作，不抛出异常；update()方法用于合并另外一个集合中的元素到当前集合中，并自动去除重复元素，参见程序 3.32。

```
#程序 3.32  集合元素的增加,修改
s = {1, 2, 3}
s.add(3)                    #添加元素,重复元素自动忽略
print(s)
s.update({3,4})             #更新当前字典,自动忽略重复的元素
print(s)
```

输出结果：

```
{1, 2, 3}
{1, 2, 3, 4}
```

pop()方法用于随机删除并返回集合中的一个元素，如果集合为空则抛出异常；remove()方法用于删除集合中的元素，如果指定元素不存在则抛出异常；discard()方法用于从集合中删除一个特定元素，如果元素不在集合中则忽略该操作；clear()方法清空集合删除所有元素，参见程序 3.33。

```
#程序 3.33  集合元素的删除
s={1,2,3,4}
s.discard(5)                #删除元素,不存在则忽略该操作
print('discard ',s)
print('pop',s.pop())        #删除并返回一个元素
s.remove(5)                 #删除元素,不存在就抛出异常
```

输出结果：

```
discard  {1, 2, 3, 4}
pop 1
KeyError: 5
```

程序 3.33 中抛出异常的语句在最后一行，前面的正常执行。如果存在异常的语句在前面，则该语句后面的就不执行。这主要是由于 Python 脚本是解释执行，遇到异常就直接退出。如果需要执行抛出异常的语句后面的内容，则需要添加异常处理。

2. 集合运算

集合虽然是 Python 语言的数据结构，它和数学上的集合一样可以做并、交、差运算，参见程序 3.34。

```
1   #程序 3.34  集合运算
2   a_set = set([4, 5, 6, 7])
3   b_set = {0, 1, 2, 3, 4}
4   print('并集 1 ',a_set | b_set)                              #并集
5   print('并集 2 ',a_set.union(b_set))
6   print('交集 1 ',a_set & b_set)                              #交集
7   print('交集 2 ',a_set.intersection(b_set))
8   print('差集 1 ',a_set.difference(b_set))                    #差集
9   print('差集 2 ',a_set - b_set)
10  print('对称差集 1 ',a_set.symmetric_difference(b_set))      #对称差集
11  print('对称差集 2 ',a_set ^ b_set)
```

输出结果：

```
并集 1 {0, 1, 2, 3, 4, 5, 6, 7}
并集 2 {0, 1, 2, 3, 4, 5, 6, 7}
交集 1 {4}
交集 2 {4}
差集 1 {5, 6, 7}
差集 2 {5, 6, 7}
对称差集 1 {0, 1, 2, 3, 5, 6, 7}
对称差集 2 {0, 1, 2, 3, 5, 6, 7}
```

集合也可以进行关系运算，只不过不是用来比较大小而是判断集合之间的包含关系、子集等，参见程序 3.35。

```
1   #程序 3.35  集合的关系运算
2   x = {1, 2, 3}
3   y = {1, 2, 5}
4   z = {1, 2, 3, 4}
5   print(x < y)                    #比较集合大小/包含关系
6   print(x < z)                    #真子集
7   print(y < z)
8   print({1, 2, 3} <= {1, 2, 3})#子集
```

输出结果：

```
False
True
False
True
```

3.5 序列封包与解包

把多个值赋给一个变量时，Python 会自动地把多个值封装成元组，称为序列封包。把一个序列（列表、元组、字符串等）直接赋给多个变量，此时会把序列中的各个元素依次赋值给每个变量，但是元素的个数需要和变量个数相同，这称为序列解包。

1. 序列封包

序列封包可参见程序 3.36。

```
1  #程序 3.36  序列封包
2  a = 1,2,3              # 序列封包，把多个值赋给一个变量
3  print(a)
4  print(type(a))
5  print(a[1:3])
```

输出结果：

```
(1, 2, 3)
<class 'tuple'>
(2, 3)
```

2. 序列解包

在序列解包时，如果只想解出部分元素，可以在变量的左边加“*”，该变量就会变成列表，保存多个元素，参见程序 3.37。

```
1  #程序 3.37  序列解包
2  a = 'hello'
3  b,c,*d = a             #字符串解包
4  print(b,c,d)
5  b,c,d = a              #元素个数与变量个数不相等时，解包会报错
6  print(b)
```

输出结果：

```
h e ['l', 'l', 'o']
ValueError: too many values to unpack (expected 2)
```

3.6 NumPy 库中的 array 结构

除 Python 内置的数据结构之外，还有一个最为重要、使用最为频繁的数据结构——NumPy 库中的 array 结构，称为数组。下面简要介绍其操作，主要是一些数学运算。

1. 生成数组

生成数组的代码可参见程序 3.38。

```
#程序 3.38  生成数组
import numpy as np
print(np.array([1,2,3,4,5]))   #将 Python 列表转换为数组
print(np.array(range(5)))      #将 Python 的 range 对象转换为数组
print(np.array([[1,2,3],[4,5,6]]))
print(np.linspace(0,10,11))    #生成等差数组
print(np.logspace(0,10,11))    #生成对数数组
print(np.zeros((3,3)))         #生成全 0 二维数组
print(np.ones((2,4)))          #生成全 1 二维数组
print(np.identity(3))          #生成全单位矩阵(二维数组)
print(np.empty((3,2)))         #生成空数组,元素值不确定
```

输出结果：

```
[1 2 3 4 5]
[0 1 2 3 4]
[[1 2 3]
 [4 5 6]]
[ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]
[1.e+00 1.e+01 1.e+02 1.e+03 1.e+04 1.e+05 1.e+06 1.e+07 1.e+08 1.e+09
 1.e+10]
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[5.11798224e-307 1.37961370e-306]
 [1.24610383e-306 1.78020169e-306]
 [1.78020984e-306 8.34454050e-308]]
```

2. 数组运算

数组与数值的运算是一次性作用在数组的每个元素上，不需要人为地编写循环对数组元素进行运算，而且这种方式速度比循环操作还要快。和内置列表不同的一点是，数组与数值相乘是数组元素与数值做乘法运算，而列表与数值相乘则是重复列表若干次，参见程序 3.39。

```
#程序 3.39  数组运算 1
import numpy as np
x = np.array([1,2,3,4,5])
print(x + 2)                  #数组一次性与数值相加,避免写循环对元素相加
print(x * 2)                  #数组与数值相乘
print(x/2)                    #数组与数值相除
print(x//2)                   #数组与数值整除
print(x ** 3)                 #幂运算
print(x % 3)                  #取余数
```

输出结果：

```
[3 4 5 6 7]
[ 2  4  6  8 10]
[0.5 1.  1.5 2.  2.5]
[0 1 1 2 2]
[  1   8  27  64 125]
[1 2 0 1 2]
```

3. 数组的矩阵运算

从数学角度看，一维数组是向量，也可以认为是特殊的矩阵，二维数组是矩阵，更高维度的数组称为张量。程序 3.40 演示常用的矩阵转置、点乘(内积)运算。

```
1   #程序 3.40  数组的矩阵运算
2   import numpy as np
3   x = np.array([[1,2,3],[2,4,5]])
4   y = np.array([[1,2,3],[3,4,5],[4,5,6]])
5   print(x.dot(y))              #矩阵相乘
6   print(x.T)                   #矩阵转置
7   print(x.transpose())         #矩阵转置
8   x1 = np.array([1,2,4])
9   y1 = np.array([3,4,5])
10  print(x1.dot(y1))            #向量内积
```

输出结果：

```
[[19 25 31]
 [34 45 56]]
[[1 2]
 [2 4]
 [3 5]]
[[1 2]
 [2 4]
 [3 5]]
31
```

4. 数组元素访问

数组元素的访问和其他语言定义的数组相似，用第一个索引取行，第二个索引取列，如果更高维度的数组则会有更多索引，另外还支持切片，取数组的某些行某些列。数组元素的访问可参见程序 3.41。

```
1   #程序 3.41  数组元素访问
2   import numpy as np
3   x = np.array([[1,2,3],[3,4,5],[4,5,6]])
4   print(x[0])              #访问数组第 0 行
5   print(x[1][2])           #访问数组第 1 行 2 列(索引从 0 开始)
6   print(x[:,0])            #访问数组所有行第 0 列
7   print(x[1,1:3])          #访问第 1 行第 1 至 3 列(不包括第 3 列)
```

输出结果：

```
[1 2 3]
5
[1 3 4]
[4 5]
```

5. 广播

广播机制是数组特有的，参与运算的两个数组大小不一样，在运算之前会各自扩展（复制）至相同行相同列，再进行运算，参见程序 3.42。

```
1  #程序 3.42  数组元素访问
2  import numpy as np
3  x = np.array([1,3,5,7,9])
4  y = np.array([10,20,30,40,50])[:,np.newaxis] #增加一个轴(维度),变成只有一列的矩阵
5  print('y:\n',y)
6  print('x + y:\n',x + y)
7  print('x * y:\n',x * y)
```

输出结果：

```
y:
 [[10]
 [20]
 [30]
 [40]
 [50]]
x + y:
 [[11 13 15 17 19]
 [21 23 25 27 29]
 [31 33 35 37 39]
 [41 43 45 47 49]
 [51 53 55 57 59]]
x * y:
 [[ 10  30  50  70  90]
 [ 20  60 100 140 180]
 [ 30  90 150 210 270]
 [ 40 120 200 280 360]
 [ 50 150 250 350 450]]
```

3.7 机器学习中的变量分布

数据结构在机器学习算法中典型的应用是组织样本数据。如果样本的特征是一维数据，利用列表就可以组织样本；如果样本的特征是二维或者更高维数据，可以利用 NumPy 包中的数组结构 array 组织样本数据；一维数据也可以利用 array 来组织，方便进行各种矩阵运算。接下来的程序中变量组织就是利用 array 这种结构进行。

每个变量分布是指随机变量的分布。2.6 节已经提到，在机器学习中，用随机变量来描述训练样本，一个样本对应一个随机变量。随机变量取每个值的概率就是随机变量的分布。

如果变量的取值是离散的，则称分布律；如果变量的取值是连续的，则称分布密度。

变量的分布对机器学习的训练模型非常重要，假设变量服从什么样的分布就会推导出对应的训练模型。例如，假设样本之间独立，每个样本都服从两点分布，就能推导出 Logistic 分类器。样本之间独立，并且每个样本都服从同样的分布，称为独立同分布。假设样本之间独立同分布，服从高斯分布则能推导出线性回归模型的损失函数如 1.7.2 小节中式(1.3)的形式。在 6.5 节给出了公式的推导，这里借此说明随机变量的分布对机器学习来说非常重要。

3.7.1 两点分布

两点分布又称伯努利分布，即随机变量的取值只有两种情况。一个二分类问题中，标签值可以取 0 对应某一类，也可以取 1 对应另外一类。两点分布正好可以用来描述一个二分类中一个样本要么取 0，要么取 1。从这点出发，可以推导 Logistic 线性分类器。这种分布比较简单，不做程序演示。

3.7.2 高斯分布


视频 7

高斯分布也称正态分布，若一维随机变量 x 服从一个位置参数为 μ、尺度参数为 σ 的概率分布，且其概率密度函数为：

$$p(x)=\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{3.1}$$

二维及高维随机变量的高斯分布是一维高斯分布的推广，公式(3.1)中的 x、μ 变成了高维向量，σ^2 也变成了协方差矩阵，σ 变成协方差矩阵行列式的开方。

$$p(x)=\frac{1}{(2\pi)^{D/2}\mid\Sigma\mid^{1/2}}\exp(-1/2(x-\mu)^{\mathrm{T}}\Sigma^{-1}(x-\mu)) \tag{3.2}$$

图 3.2 是程序 3.43 生成的一维标准正态分布，$\mu=0$，$\sigma=1$。第 8 行取区间[$\mu-3\sigma$，$\mu+3\sigma$]平均分布的 51 个点作为随机变量的 x 的值；第 9 行是计算这些随机变量高斯分布的概率值；第 10 行是根据随机变量的值和对应的概率绘图；第 13 行设置的字体可以有多种，如黑体、宋体、楷体等。

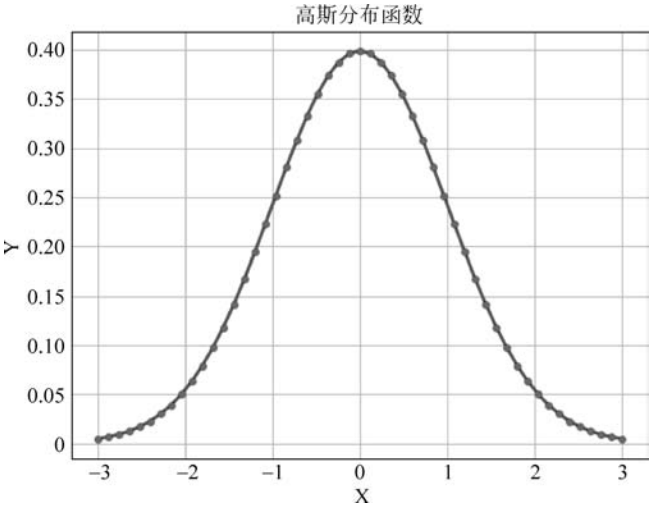


图 3.2 一维标准正态分布

```
#程序 3.43  一维标准正态分布
import math
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
mu = 0                          #均值
sigma = 1                       #方差
x = np.linspace(mu - 3 * sigma, mu + 3 * sigma, 51)
y = np.exp(-(x-mu) ** 2/(2 * sigma ** 2))/(math.sqrt(2 * math.pi) * sigma)
plt.plot(x, y, 'r-', x, y, 'go', linewidth=2, markersize=4)   #画图
plt.xlabel('X')
plt.ylabel('Y')
mpl.rcParams['font.sans-serif'] = 'SimHei'   #FangSong/黑体 FangSong/KaiTi 也行
mpl.rcParams['axes.unicode_minus'] = False
plt.title(u'高斯分布函数', fontsize=10)
plt.grid(True)
plt.show()
```

二维标准正态分布的三维图形如图 3.3 所示，由程序 3.44 生成。其中，第 6 行是调用 NumPy 模块中的函数生成二维随机变量的值，这些值是区间[－3，3]内平均分布的 100 个点；第 7 行输出的形状显示返回的 $\boldsymbol{x}$ 是一个列向量；第 8 行输出的形状显示返回的 $\boldsymbol{y}$ 是一个行向量；第 9 行是计算二维随机变量高斯分布的概率；第 11 行是准备绘制三维图形；第 12 行利用 surface()函数绘制三维图形。

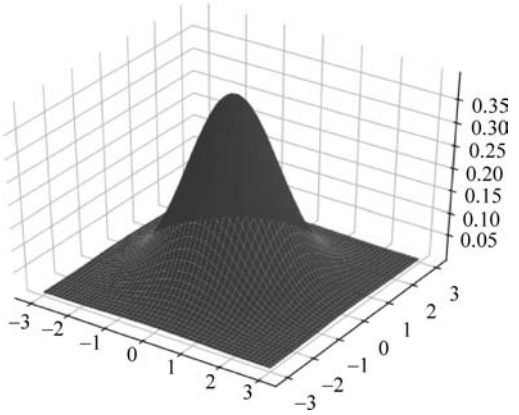


图 3.3　二维标准正态分布的三维图形

```
#程序 3.44  二维标准正态分布
import math
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
x, y = np.ogrid[-3:3:100j, -3:3:100j]                    #生成二维平面上的坐标点
print(x.shape)
print(y.shape)
z = np.exp(-(x**2 + y**2)/2) / math.sqrt(2*math.pi) #二元高斯分布
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z)
plt.show()
```

输出结果：

```
(100, 1)
(1, 100)
```

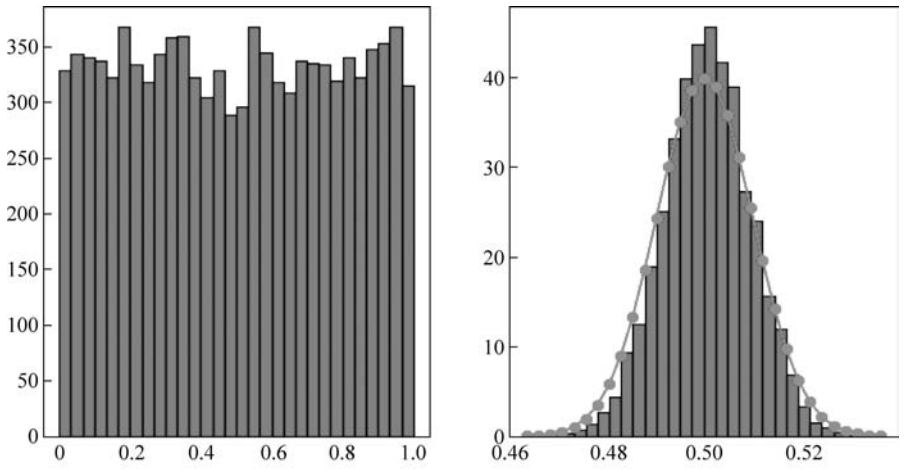
3.7.3 中心极限定理


视频 8

中心极限定理是随机变量序列部分和的分布渐近于正态分布的定理，是统计机器学习误差分析的理论基础。一些随机现象受到许多相互独立的随机因素的影响，如果每个因素所产生的影响都很微小，总的影响可以看作是服从正态分布的。在线性回归模型中，预测值与真值之间的误差就可以认为是服从 $\varepsilon \sim N(0,\sigma^2)$ 的正态分布，从而推导出损失函数。下面通过程序 3.45 验证服从均匀分布的随机变量的和满足中心极限定理，对于服从其他分布的随机变量，读者可以自己验证。

```
#程序 3.45  均匀分布的中心极限定理
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from scipy.stats import norm
x = np.random.random(10000)                              #生成均匀分布的随机变量
plt.subplot(121)
plt.hist(x,30,edgecolor='k',color='g')                   #直方图
matplotlib.rcParams['font.sans-serif'] = 'simhei'
plt.title('均匀分布的随机变量',fontsize=11)
for i in range(1000):                                    #生成 1000 个随机变量并求和
    x += np.random.random(10000)
plt.subplot(122)
gx = plt.hist(x/1000,30,edgecolor='k',color='g',density=True)[1]
gy = norm.pdf(gx,loc=0.5,scale=0.01)                     #与自动生成的高斯分布做对比
plt.plot(gx,gy,'r-o')
plt.title('1000 个均匀分布的随机变量和的分布',fontsize=11)
plt.show()
```

图 3.4 是程序 3.45 输出的图形，从图 3.4(b)中的直方图可以看出，均分分布的随机变量的和的分布集中在 0.5 两边的区域内，与生成的正态分布非常近似。



(a) 均匀分布的随机变量的直方图　　(b) 1000个均匀分布的随机变量和的分布图

图 3.4　均匀分布的随机变量和 1000 个均匀分布的随机变量和的分布图

3.8　实　　验

1. 实验目的

(1) 掌握列表、元组、字典、集合四种数据结构的创建、删除、更新等方法。

(2) 掌握列表、元组、字典、集合四种数据结构的区别和联系。

(3) 掌握序列解包和封包的形式及用法。

2. 实验内容

(1) 编写 Python 程序计算无理数圆周率 π 和自然对数的底 e。这两个看似毫不相干的无理数却可以通过欧拉公式统一起来。欧拉公式堪称数学上最完美的公式，也有人称式(3.3)是“上帝创造的公式”。

$$e^{i\pi}+1=0 \tag{3.3}$$

计算圆周率有多种方法，这里给出两种，即蒙特卡罗方法和级数求解。蒙特卡罗方法也称随机方法。假设有一个边长为 2 的正方形，则正方形的面积为 4，内切圆的面积就是 π，如图 3.5 所示。利用随机方法生成二维坐标点，坐标点的坐标值在 0 到＋1 之间，统计坐标点离圆心距离为 1 的点的个数，除以总共生成的点数，再乘以 4，就近似圆周率 π。

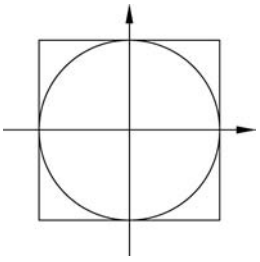


图 3.5　蒙特卡罗方法计算圆周率

使用蒙特卡罗方法计算圆周率，参考程序 3.46。

```
1   #程序 3.46  蒙特卡罗方法计算圆周率
2   times = 100000
3   for i in range(times):
4       x = random()                    #默认生成[0,1)内的随机数
5       y = random()
```

```
6        if x * x + y * y <= 1:
7            counts += 1
8    print('落入坐标轴第一象限的点数:',counts)
9    print('圆周率近似值:',4.0 * counts/times)
```

利用公式求圆周率：

$$\pi^2 = 6 \times \sum_{i=1}^{n} 1/i^2 \tag{3.4}$$

利用公式求解圆周率，参见程序 3.47。

```
1    #程序 3.47  依公式(3.4)求解圆周率
2    a = list(range(1,1000000))
3    b = [1/x ** 2 for x in a]                #列表推导式求公式(3.4)中的一项
4    print((6 * sum(b)) ** 0.5)
```

自然对数的底的计算公式：

$$e = 1 + 1 + \frac{1}{2!} + \cdots + \frac{1}{n!} \tag{3.5}$$

自然对数底求解，参见程序 3.48。其中，第 5 行 reduce()函数内的第一个参数是一个函数，这里用 lambda 表达式来代替。后面章节还会讨论 lambda 表达式，它本身就是一个没有名字的函数。第 5 行的 lambda 表达式实现的是乘积运算。

```
1    #程序 3.48  依公式(3.5)计算自然对数的底
2    from functools import reduce
3    a = []
4    for i in range(2,10):
5        a.append(1/reduce(lambda x,y:x * y,list(range(1,i))))
6    print(sum(a) + 1)
```

(2) 理解圆周率和自然对数底的计算公式，尝试利用其他函数或方法编写替代的程序求解。

(3) 参考教材 3.7 节编写 Python 程序，画出服从泊松分布的随机变量的概率分布图，及其和的分布图形，验证中心极限定理。

(4) 根据每一步的结果写出实验报告。

本章小结

Python 数据结构也是 Python 的内置对象，数据结构是对所处理的数据做有效的组织，在此基础上可以方便处理。Python 主要提供 list、tuple、dict、set 等重要的数据结构。本章给出这几种数据结构的创建、删除、访问以及常用的方法，在编写程序时要善于利用这些数据结构解决实际问题。

习　　题

一、选择题

1. 关于列表，下面描述不正确的是(　　)。

A. 元素类型可以不同　　B. 长度没有限制

C. 必须按顺序插入元素　　D. 支持 in 运算符

2. 下列方法仅适用于列表，而不适用于字符串的是(　　)。

A. count()　　B. sort()　　C. find()　　D. index()

3. 下列程序的输出结果是(　　)。

```
a = [10,20,30]
print(a * 2)
```

A. [10,20,30,10,20,30]　　B. [20,40,60]

C. [11,22,33]　　D. [10,20,30]

4. 表达式(12,34,56)+(78)的结果是(　　)。

A. (12,34,56,(78))　　B. (12,34,56,78)

C. [12,34,56,78]　　D. 程序出错

5. 下列程序的输出结果是(　　)。

```
sum = 0
for i in range(10):
    sum += i
print(sum)
```

A. 0　　B. 45　　C. 10　　D. 55

6. 关于元组数据结构，下面描述正确的是(　　)。

A. 支持 in 运算符　　B. 所有元素数据类型必须相同

C. 插入的新元素放在最后　　D. 元组不支持切片操作

7. 元组和列表都支持的方法是(　　)。

A. extend()　　B. append()　　C. index()　　D. remove()

8. 在字典中查找一个键和查找一个值，哪个速度快？(　　)

A. 同样快　　B. 值　　C. 键　　D. 无法比较

9. 下列语句的执行结果为(　　)。

```
{1,2,3}&{3,4,5}
```

A. {3}　　B. {1,2,3,4,5}　　C. {1,2,3,3,4,5}　　D. 程序出错

10. 下列语句哪个不能创建一个字典？(　　)

A. {}　　B. dict(zip([1,2,3],[4,5,6]))

C. dict([(1,4),(2,5),(3,6)])　　D. {1,2,3}

二、填空题

1. 下列程序输出结果是_____。

```
a = [10,20,30]
b = a
b[1] = 40
print(a[1])
```

2. 下列程序输出结果是_____。

```
[n * n for n inrange(6) if n * n % 2 = = 1]
```

三、程序与简答题

1. 为什么尽量从列表的尾部进行元素的增加与删除操作?

2. range()函数返回的是什么结构?

3. 编写程序生成1000个0～100之间的随机整数,统计每个元素出现次数。提示:借助字典结构。

4. 写出使用列表推导式生成包含10个数字5的列表的语句。

5. 假设有一个列表a,要求从a中每3个元素取1个,将取到的元素组成新的列表b,试写出语句。

6. 编写程序生成20个随机数的列表,将前10个元素升序排列,后10个元素降序排列,输出结果。

7. 列表对象的sort()方法用来对列表元素进行原地排序,该函数返回值是什么结构?