第5章

视频讲解

组合数据和数据结构

Python内置若干组合数据类型，包括序列数据类型、字典和集合等。组合数据类型可以实现复杂数据的处理。Python的标准库模块提供了若干对象和函数，用于实现各种通用数据结构和算法。

5.1 数据结构基础

5.1.1 数据结构的定义

著名的计算机科学家尼克劳斯·沃思(Nikiklaus Wirth)指出：程序＝算法＋数据结构。算法是执行特定任务的方法，数据结构是一种存储数据的方式，有助于求解特定的问题。选择恰当的数据结构来实现算法可以带来更高的运行或者存储效率。

数据(data)是能够被计算机处理的对象集合。数据由数据元素(date element)组成，数据元素包含数据项(data item)。

例如，在学生档案管理系统中，每位学生信息是数据的基本单位，称之为数据元素，也称为元素(element)、结点(node)或者记录(record)。

组成数据元素的项称之为数据项，例如，学生信息由学号、姓名、性别、出生年月、专业等组成。数据项是数据的最小标识单位，又称为字段(field)或者域(field)。

数据结构是计算机存储和组织数据的方式，即相互之间存在一种或多种特定关系的数据元素的集合。数据结构通常由三个部分组成：数据的逻辑结构、数据的物理结构和数据的运算结构。

1. 数据的逻辑结构

数据的逻辑结构反映数据元素之间的逻辑关系。数据的逻辑结构主要包括线性结构(一对一的关系)、树形结构(一对多的关系)、图形结构(多对多的关系)、集合等。

2. 数据的物理结构

数据的物理结构反映数据的逻辑结构在计算机存储空间的存放形式，即数据结构在计算机中的表示。其具体的实现方法包括顺序、链接、索引、散列等多种形式。一种数据结构可以由一种或者多种物理存储结构实现。

3. 数据的运算结构

数据的运算结构反映在数据的逻辑结构上定义的操作算法，例如检索、插入、删除、更新和排序等。

5.1.2　数据的逻辑结构

数据的逻辑结构反映数据元素之间的逻辑关系，与它们在计算机中的存储位置无关。

数据结构可以表示为 DS=(D, R)，其中 DS 表示数据结构，D 表示数据集合，R 表示关系(relation)集合。

例如，三口之家的成员的数据逻辑结构可以表示如下：

```
DS = (D, R)
D = {父亲, 母亲, 孩子}
R = {(父亲, 母亲), (父亲, 孩子), (母亲, 孩子)}
```

数据的逻辑结构可以分为线性结构和非线性结构。

线性结构中的元素结点具有线性关系。如果从数据结构的语言来描述，线性结构具有以下特点。

(1) 线性结构是非空集。

(2) 线性结构有且仅有一个开始结点和一个终端结点。

(3) 线性结构所有结点都最多只有一个直接前趋结点和一个直接后继结点。

在实际应用中，线性表、队列、栈等数据结构属于线性结构。

非线性结构中的各元素结点之间具有多个对应关系。如果从数据结构的语言来描述，非线性结构具有以下特点。

(1) 非线性结构是非空集。

(2) 非线性结构的一个结点可能有多个直接前趋结点和多个直接后继结点。

在实际应用中，数组、广义表、树结构和图结构等数据结构属于非线性结构。

常用的数据逻辑结构包括如下几种方式，如图 5-1 所示。

(1) 集合：数据结构中的元素之间除了“同属一个集合”的相互关系外，别无其他关系。

(2) 线性结构：数据结构中的元素存在一对一的相互关系。

(3) 树形结构：数据结构中的元素存在一对多的相互关系。

(4) 图形结构：数据结构中的元素存在多对多的相互关系。

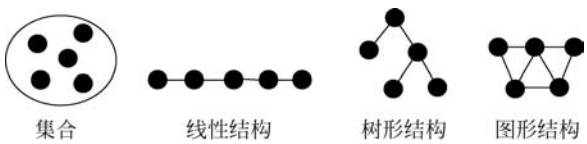


图 5-1　常用的数据逻辑结构

5.1.3　数据的物理结构

数据的物理结构是指逻辑结构在计算机存储空间的存放形式。数据的物理结构是数据

结构在计算机中的实现方式，它包括数据元素的机内表示和关系的机内表示。

实现逻辑数据结构的常用方法包括顺序、链接、索引、散列等。一种逻辑数据结构可以表示成一种或者多种物理存储结构。

数据元素通常称为结点，在计算机内部表示为二进制位(bit)的位串。

数据元素之间的关系在计算机内部的存储结构通常有两种方式：顺序存储结构和链式存储结构。顺序映像借助元素在存储器中的相对位置来表示数据元素之间的逻辑关系。非顺序映像借助指示元素存储位置的指针来表示数据元素之间的逻辑关系。

5.1.4 常用算法

设计和实现数据结构的目的在于通过算法更有效地处理数据。数据的算法基于数据的逻辑结构，但具体实现要在物理存储结构上进行。

基于数据结构的常用算法包括以下几种。

(1) 检索：在数据结构中查找满足给定条件的结点。

(2) 插入：在数据结构中增加新的结点。

(3) 删除：从数据结构中删除指定结点。

(4) 更新：改变指定结点的一个或者多个字段的值。

(5) 排序：按某种指定的顺序重新排列结点(从而可以提高其他算法的操作效率)。

5.2 常用的数据结构

5.2.1 线性表

线性表(linear list)是最基本的一种数据结构，是具有相同特性的数据元素的有限序列，通常记作$(a_1, a_2, \cdots, a_n)$。其中 a_1 无前趋，a_n 无后继，其他每个元素都有一个前趋和后继。

线性表的物理存储结构主要有两种：顺序存储结构和链式存储结构，前者称为顺序表，后者称为线性链表。

1. 顺序表

顺序存储结构使用一组地址连续的存储单元依次存储线性表的数据元素，以“物理位置相邻”来表示线性表中数据元素间的逻辑关系，用户可以随机存取顺序表中任一元素。

顺序表的存储示意图如图 5-2 所示。

数据元素	a_1	a_2	…	a_n
存储地址	0	1	…	n−1

图 5-2 顺序表的存储示意图

顺序表的优点是查找和访问元素速度快，但插入和删除开销大。

2. 线性链表

线性链表(linked list)是一种数据元素按照链式存储结构进行存储的数据结构，这种存

储结构具有在物理上存在非连续的特点。

线性链表由一系列数据结点构成，每个数据结点包括数据域和指针域两部分。其中，指针域保存了数据结构中下一个元素存放的地址。链表结构中数据元素的逻辑顺序是通过链表中的指针链接次序来实现的。

线性链表的存储示意图如图 5-3 所示。链表的头指向第一个元素的存储位置 1，其中存储第一个元素 a_1，并包含指向下一个元素的指针 3。指针指向 0 时，表示链表结束位置。

	存储序号	数据域	指针域
head=1	1	a_1	3
	2		
	3	a_2	4
	4	a_3	6
	5		
	6	a_4	0

图 5-3　线性链表的存储示意图

线性链表还可以表示为如图 5-4 所示。

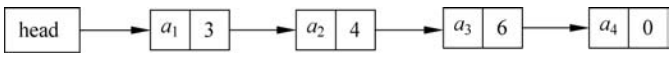


图 5-4　线性链表示意图

线性链表的优点在于插入和删除效率高。其缺点是访问元素时需要遍历链表的所有元素，因而效率欠佳。

除了上面讨论的单向线性链表外，还存在双向线性链表、单向循环线性链表、双向循环线性链表等。

5.2.2　队列

队列(queue)是先进先出(FIFO，First In First Out)的序列，即最先添加(push)的元素，是最先弹出(pop)的元素。

列表可以实现队列，但并不适合。因为从列表的头部移除一个元素，列表中的所有元素都需要移动位置，所以效率不高。用户可以使用 collections 模块中的 deque 对象来删除列表头部的元素。

5.2.3　栈

栈(stack)是后进先出(LIFO，Last In First Out)的线性表，即最后入队的元素，是最先出队的元素。

向列表最后位置添加元素和从最后位置移除元素非常方便和高效，故使用列表可以快捷高效地实现栈。list.append()方法对应于入栈操作(push)；list.pop()对应于出栈操作(pop)。

5.2.4　树

1. 树的定义

树(tree)是典型的非线性结构，由 n(n>=1)个有限结点组成一个具有层次关系的集

合，其形状像一棵倒挂的树。客观世界存在许多树状逻辑关系，例如部门组织结构、文件系统结构等。树的一般形式如图 5-5 所示。

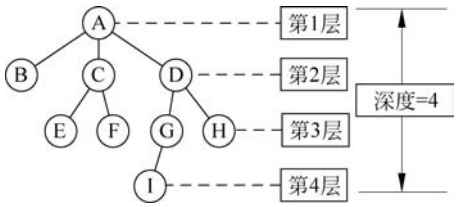


图 5-5　树的一般形式

树具有以下的特点。

（1）每个结点有零个或多个子结点。

（2）没有父结点的结点称为根结点（root）。

（3）每一个非根结点有且只有一个父结点。

（4）除了根结点外，每个子结点可以分为多个不相交的子树。

2. 树的相关术语

树有以下的常用术语。

（1）结点：每个元素称为结点。

（2）根结点：没有父结点的结点称为根结点。

（3）结点的度（degree）：一个结点含有的子结点个数称为该结点的度。

（4）叶结点或终端结点（leaf）：度为 0 的结点称为叶结点。

（5）非终端结点或分支结点（branch）：度不为 0 的结点。

（6）双亲结点或父结点（parent）：若一个结点含有子结点，则这个结点称为其子结点的父结点。

（7）孩子结点或子结点（child）：一个结点含有的子树的根结点称为该结点的子结点。

（8）兄弟结点（sibling）：具有相同父结点的结点互称为兄弟结点。

（9）树的度：一棵树中，最大的结点的度称为树的度。

（10）结点的层次（level）：从根开始定义起，根为第 1 层，根的子结点为第 2 层，以此类推。

（11）树的高度或深度（depth）：树中结点的最大层次。

（12）堂兄弟结点：双亲在同一层的结点互为堂兄弟结点。

（13）结点的祖先：从根到该结点所经分支上的所有结点。

（14）子孙：以某结点为根的子树中任一结点都称为该结点的子孙。

（15）森林：m（m>=0）棵互不相交的树的集合称为森林。

（16）空树。空集合也是树，称为空树。空树中没有结点。

（17）无序树：树中任意结点的子结点之间没有顺序关系，这种树称为无序树，也称为自由树。

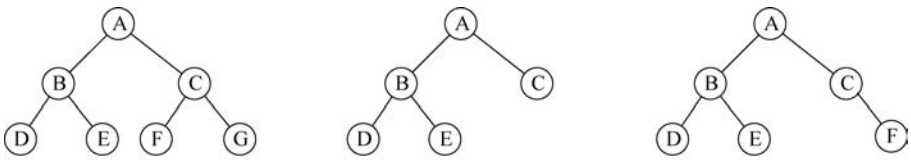
（18）有序树：树中任意结点的子结点之间有顺序关系，这种树称为有序树。

（19）二叉树：每个结点最多含有两个子树的树称为二叉树。

(20) 满二叉树：如果一棵二叉树只有深度为 0 的结点和深度为 2 的结点，并且度为 0 的结点在同一层上，则这棵二叉树为满二叉树。即，一棵深度为 k 且有 2^k-1 个结点的二叉树称为满二叉树。满二叉树每一层的结点个数都达到了最大值，即满二叉树的第 i 层上有 2^{i-1} 个结点。如图 5-6(a)所示是一棵深度为 3 的满二叉树。

(21) 完全二叉树：满二叉树中叶子结点所在的层次中，如果自右向左连续缺少若干叶子结点，这样的得到的二叉树被称为完全二叉树。即，若设二叉树的深度为 k，除第 k 层外，其他各层(1～k－1)的结点数都达到最大个数，而第 k 层所有的结点都连续集中在最左边，这就是一个完全二叉树。如图 5-6(b)所示是一棵完全二叉树。如图 5-6(c)所示是一棵非完全二叉树。满二叉树是完全二叉树的特殊形态。

(22) 哈夫曼树(最优二叉树)：带权路径最短的二叉树称为哈夫曼树或最优二叉树。



(a) 深度为3的满二叉树　(b) 完全二叉树　(c) 非完全二叉树

图 5-6　二叉树示例

二叉树具有以下重要性质。

(1) 二叉树的第 i(i≥1)层上至多有 2^{i-1} 个结点。

(2) 深度为 h(h≥1)的二叉树中最少有 h 个结点，最多含有 2^h-1 个结点。

(3) 对于任意一棵非空二叉树，若有 m 个叶子结点，有 n 个深度为 2 的结点，则必有 m＝n＋1。

(4) 具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor+1$，其中 $\lfloor\ \rfloor$ 表示向下取整。

(5) 一棵有 n 个结点的完全二叉树，如果对树中的结点按照自顶向下、同一层从左向右的顺序进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i(1≤i≤n)的结点有如下特性。

- 若 i＝1，则该结点是二叉树的根，无双亲结点。否则，编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点。
- 若 2＊i＞n，则该结点无左孩子结点。否则，编号为 2＊i 的结点为其左孩子结点。
- 若 2＊i＋1＞n，则该结点无右孩子结点。否则，编号为 2＊i＋1 的结点为其右孩子结点。
- 若结点编号 i 为奇数，i 不等于 1，并且处于右兄弟结点位置，则编号为 i－1 的结点为其左兄弟结点。
- 若结点编号 i 为偶数，i 不等于 n，并且处于左兄弟结点位置，则编号为 i＋1 的结点为其右兄弟结点。
- 结点 i 所在的层次为 $\lfloor \log_2 i \rfloor+1$。

3. 二叉树的遍历

二叉树是 n 个有限元素的集合，该集合或者为空，或者由一个称为根的元素及两个不相交的、被分别称为左子树和右子树的二叉树组成。二叉树是有序树。当集合为空时，称该二

叉树为空二叉树。二叉树具有五种基本形态，如图 5-7 所示。

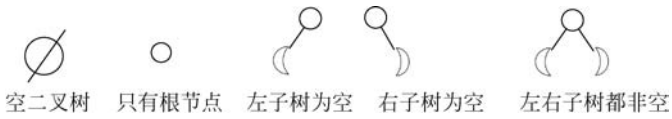


图 5-7　二叉树的五种基本形态

遍历是对树的一种最基本的运算。所谓遍历二叉树，就是按一定的规则和顺序遍历二叉树的所有结点，使每一个结点都被访问一次，而且只被访问一次。

一棵非空的二叉树由根结点、左子树、右子树三个基本部分组成。因此，在任一给定结点上，可以按某种次序执行三个操作。

(1) 访问结点本身(N)。

(2) 遍历该结点的左子树(L)。

(3) 遍历该结点的右子树(R)。

以上三种操作有六种遍历方法：NLR、LNR、LRN、NRL、RNL、RLN。

前三种次序与后三种次序对称，故只讨论前三种次序。

1) NLR

前序遍历(preorder traversal)，又称为先序遍历。若二叉树非空，则依次执行如下操作。

(1) 访问根结点(N)。

(2) 遍历左子树(L)。

(3) 遍历右子树(R)。

2) LNR

中序遍历(inorder traversal)。若二叉树非空，则依次执行如下操作。

(1) 遍历左子树(L)。

(2) 访问根结点(N)。

(3) 遍历右子树(R)。

3) LRN

后序遍历(postorder traversal)。若二叉树非空，则依次执行如下操作。

(1) 遍历左子树(L)。

(2) 遍历右子树(R)。

(3) 访问根结点(N)。

例如，给定如图 5-8 所示的二叉树：

其前序遍历顺序为：ABCDFEG。

其中序遍历顺序为：BAFDCEG。

其后序遍历顺序为：BFDGECA。

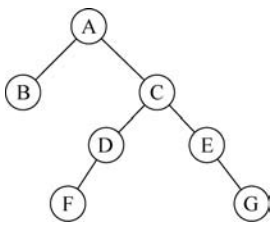


图 5-8　一棵二叉树

5.2.5　图

图(graph)是另一种非线性数据结构。图是由顶点集合 V(vertex)和边集合 E(edge)组成的，定义为 G=(V,E)。

在图结构中，数据结点一般称为顶点，而边是顶点的有序偶对。如果两个顶点之间存在一条边，那么就表示这两个顶点具有相邻关系。

5.2.6　堆

堆(heap)是一种特殊的树形数据结构，其中子结点与父结点是一种有序关系。

5.2.7　散列表

散列表(Hash table，也叫哈希表)是把键值映射(映射函数)到表(散列表)的数据结构，通过键值可以实现快速查找。

5.3　Python 序列数据概述

5.3.1　数组

数组是一种数据结构，用于存储和处理大量的数据。通过将所有的数据存储在一个或者多个数组中，然后通过索引下标访问并处理数组的元素，可以实现复杂数据处理任务。

Python 语言没有提供直接创建数组的功能，但可以使用其内置的序列数据类型(例如列表 list)实现数组的功能。

一般情况，一维数组可以使用列表来实现，二维数据可以使用列表的列表来实现。

5.3.2　序列数据类型

序列(sequence)数据类型是 Python 基础的数据结构，是一组有顺序的元素的集合。序列数据可以包含一个或者多个元素(即对象，元素也可以是其他序列数据)，也可以是一个没有任何元素的空序列。

Python 内置的序列数据类型包括元组(tuple)、列表(list)、字符串(str)和字节数据(bytes 和 bytearray)。

5.4　序列数据的基本操作

5.4.1　序列的长度、最大值、最小值、求和

通过内置函数 len()、max()、min()可以获取序列的长度、序列中元素最大值、序列中元素最小值。内置函数 sum()可以获取列表或者元组中各元素之和；如果有非数值元素，则导致 TypeError；对于字符串(str)和字节数据(bytes)，也将导致 TypeError。

【例 5.1】　序列数据的求和示例。

```
>>> t1 = (1,2,3,4)
>>> sum(t1)      #输出：10
>>> t2 = (1,'a',2)
>>> sum(t2)      # TypeError: unsupported operand type(s) for + : 'int' and 'str'
```

【例 5.2】 序列的长度、最大值、最小值操作示例。

```
>>> s = 'abcdefg'      >>> t = (10,2,3)      >>> lst = [1,2,9,5,4]      >>> b = b'ABCD'
>>> len(s)             >>> len(t)           >>> len(lst)              >>> len(b)
7                      3                     5                        4
>>> max(s)             >>> max(t)           >>> max(lst)              >>> max(b)
'g'                    10                    9                        68
>>> min(s)             >>> min(t)           >>> min(lst)              >>> min(b)
'a'                    2                     1                        65
>>> s2 = ''            >>> t2 = ()          >>> lst2 = []            >>> b2 = b''
>>> len(s2)            >>> len(t2)          >>> len(lst2)             >>> len(b2)
0                      0                     0                        0
```

5.4.2 序列的索引访问操作

序列表示可以通过索引下标访问的可迭代对象。例如，用户可以通过整数下标访问序列 s 的元素。

```
s[i]          #访问序列 s 在索引 i 处的元素
```

索引下标从 0 开始，第 1 个元素为 s[0]，第 2 个元素为 s[1]，以此类推，最后一个元素为 s[len(s) − 1]。索引下标也可以从最后一个元素开始，从−1 开始，即最后一个元素为 s[−1]，第 1 个元素为 s[−len(s)]。

如果索引下标越界，则导致 IndexError；如果索引下标不是整数，则导致 TypeError。例如：

```
>>> s = 'abc'
>>> s[0]        #输出：'a'
>>> s[3]        # IndexError: string index out of range
>>> s['a']      # TypeError: string indices must be integers
```

序列 s 的索引下标示意图如图 5-9 所示。

图 5-9 序列 s 的索引下标示意图

【例 5.3】　序列的索引访问示例。

```
>>> s = 'abcdef'          >>> t = ('a','e','i','o','u')   >>> lst = [1,2,3,4,5]    >>> b = b'ABCDEF'
>>> s[0]                  >>> t[0]                       >>> lst[0]               >>> b[0]
'a'                       'a'                           1                        65
>>> s[2]                  >>> t[1]                       >>> lst                  >>> b[1]
'c'                       'e'                           [1, 2, 3, 4, 5]          66
>>> s[ - 1]               >>> t[ - 1]                    >>> lst[2] = 'a'         >>> b[ - 1]
'f'                       'u'                           >>> lst[ - 2] = 'b'       70
>>> s[ - 3]               >>> t[ - 5]                    >>> lst                  >>> b[ - 2]
'd'                       'a'                           [1, 2, 'a', 'b', 5]      69
```

5.4.3　序列的切片操作

通过切片(slice)操作可以截取序列 s 的一部分。切片操作的基本形式为：

```
s[i:j]  或者  s[i:j:k]
```

其中，i 为序列开始下标(包含 s[i])；j 为序列结束下标(不包含 s[j])；k 为步长。如果省略 i，则从下标 0 开始；如果省略 j，则直到序列结束为止；如果省略 k，则步长为 1。

下标也可以为负数。如果截取范围内没有数据，则返回空元组；如果超过下标范围，不报错。

【例 5.4】　序列的切片操作示例。

```
>>> s='abcdef'          >>> t=('a','e','i','o','u')    >>> lst=[1,2,3,4,5]     >>> b=b'ABCDEF'
>>> s[1:3]              >>> t[-2:-1]                  >>> lst[:2]             >>> b[2:2]
'bc'                    ('o',)                       [1, 2]                   b''
>>> s[3:10]             >>> t[-2:]                    >>> lst[:1]=[]          >>> b[0:1]
'def'                   ('o', 'u')                   >>> lst                  b'A'
>>> s[8:2]              >>> t[-99:-5]                 [2, 3, 4, 5]             >>> b[1:2]
''                      ()                           >>> lst[::2]            b'B'
>>> s[:]                >>> t[-99:-3]                 [2, 3]                   >>> b[2:2]
'abcdef'                ('a', 'e')                   >>> lst[:2]='a'          b''
>>> s[:2]               >>> t[::]                     >>> lst[1:]='b'         >>> b[-1:]
'ab'                    ('a', 'e', 'i', 'o', 'u')    >>> lst                  b'F'
>>> s[::2]              >>> t[1:-1]                   ['a', 'b']               >>> b[-2:-1]
'ace'                   ('e', 'i', 'o')              >>> del lst[:1]          b'E'
>>> s[::-1]             >>> t[1::2]                   >>> lst                  >>> b[0:len(b)]
'fedcba'                ('e', 'o')                   ['b']                    b'ABCDEF'
```

5.4.4　序列的连接和重复操作

通过连接操作符＋可以连接两个序列(s1 和 s2)，形成一个新的序列对象；通过重复操

作符 * 可以重复一个序列 n 次(n 为正整数)。序列连接和重复操作的基本形式为：

```
s1 + s2  或者  s * n  或者  n * s
```

连接操作符＋和重复操作符 * 也支持复合赋值运算，即＋＝和 * ＝。

【例 5.5】 序列的连接和重复操作示例。

```
>>> s1 = 'abc'          >>> t1 = (1,2)        >>> lst1 = [1,2]          >>> b1 = b'ABC'
>>> s2 = 'xyz'          >>> t2 = ('a','b')    >>> lst2 = ['a','b']      >>> b2 = b'XYZ'
>>> s1 + s2             >>> t1 + t2           >>> lst1 + lst2          >>> b1 + b2
'abcxyz'                (1, 2, 'a', 'b')      [1, 2, 'a', 'b']         b'ABCXYZ'
>>> s1 * 3              >>> t1 * 2            >>> 2 * lst2             >>> b1 * 3
'abcabcabc'             (1, 2, 1, 2)          ['a', 'b', 'a', 'b']     b'ABCABCABC'
>>> s1 += s2            >>> t1 += t2          >>> lst1 += lst2         >>> b1 += b2
>>> s1                  >>> t1                >>> lst1                 >>> b1
'abcxyz'                (1, 2, 'a', 'b')      [1, 2, 'a', 'b']         b'ABCXYZ'
>>> s2 * = 2            >>> t2 * = 2          >>> lst2 * = 2           >>> b2 * = 2
>>> s2                  >>> t2                >>> lst2                 >>> b2
'xyzxyz'                ('a', 'b', 'a', 'b')  ['a', 'b', 'a', 'b']     b'XYZXYZ'
```

5.4.5 序列的成员关系操作

用户可以通过下列方式之一判断一个元素 x 是否存在于序列 s 中。

- x in s：如果为 True，则表示存在。
- x not in s：如果为 True，则表示不存在。
- s.count(x[, start[, end]])：返回 x 在 s(指定范围[start,end))中出现的次数。
- s.index(x[, start[, end]])：返回 x 在 s(指定范围[start,end))中第一次出现的下标。

其中，指定范围[start, end)从下标 start(包括，默认为 0)开始，到下标 end 结束(不包括，默认为 len(s))。

对于 s.index(value, [start, [stop]])方法，如果找不到时，则导致 ValueError。例如：

```
>>> 'To be or not to be, this is a question'.index('123') # ValueError: substring not found
```

【例 5.6】 序列中元素的存在性判断示例。

```
>>> s = 'Good, better, best!'   >>> t = ('r', 'g', 'b')   >>> lst = [1,2,3,2,1]   >>> b = b'Oh, Jesus!'
>>> 'o' in s                    >>> 'r' in t              >>> 1 in lst             >>> b'O' in b
True                            True                     True                    True
>>> 'g' not in s                >>> 'y' not in t         >>> 2 not in lst        >>> b'o' not in b
True                            True                     False                   True
>>> s.count('e')                >>> t.count('r')         >>> lst.count(1)        >>> b.count(b's')
3                               1                        2                       2
>>> s.index('e', 10)            >>> t.index('g')         >>> lst.index(3)        >>> b.index(b's')
10                              1                        2                       6
```

5.4.6　序列的比较运算操作

两个序列支持比较运算符(<、<=、==、!=、>=、>),字符串比较运算按顺序逐个元素进行比较。

【例 5.7】　序列的比较运算示例。

```
>>> s1 = 'abc'
>>> s2 = 'abc'
>>> s3 = 'abcd'
>>> s4 = 'cba'
>>> s1 > s4
False
>>> s2 <= s3
True
>>> s1 == s2
True
>>> s1 != s3
True
>>> 'a' > 'A'
True
>>> 'a' >= ''
True
>>> t1 = (1,2)
>>> t2 = (1,2)
>>> t3 = (1,2,3)
>>> t4 = (2,1)
>>> t1 < t4
True
>>> t1 <= t2
True
>>> t1 == t3
False
>>> t1 != t2
False
>>> t1 >= t3
False
>>> t4 > t3
True
>>> s1 = ['a','b']
>>> s2 = ['a','b']
>>> s3 = ['a','b','c']
>>> s4 = ['c','b','a']
>>> s1 < s2
False
>>> s1 <= s2
True
>>> s1 == s2
True
>>> s1!= s3
True
>>> s1 >= s3
False
>>> s4 > s3
True
>>> b1 = b'abc'
>>> b2 = b'abc'
>>> b3 = b'abcd'
>>> b4 = b'ABCD'
>>> b1 < b2
False
>>> b1 <= b2
True
>>> b1 == b2
True
>>> b1 >= b3
False
>>> b3!= b4
True
>>> b4 > b3
False
```

5.4.7　序列的排序操作

通过内置函数 sorted()可以返回序列的排序列表。通过类 reversed()构造函数可以返回序列的反序的迭代器。内置函数 sorted()形式如下：

```
sorted(iterable, key = None, reverse = False)    #返回序列的排序列表
```

其中,key 是用于计算比较键值的函数(带一个参数),例如 key=str.lower。如果 reverse=True,则反向排序。

【例 5.8】　序列的排序操作示例。

```
>>> s1 = 'axd'
>>> sorted(s1)
['a', 'd', 'x']
> > >  list ( reversed
(s1))
['d', 'x', 'a']
>>> s2 = (1,4,2)
>>> sorted(s2)
[1, 2, 4]
>>> sorted(s2,reverse = True)
[4, 2, 1]
>>> list(reversed(s2))
[2, 4, 1]
>>> s3 = 'abAC'
>>> sorted(s3, key = str.lower)
['a', 'A', 'b', 'C']
>>> list(reversed(s3))
['C', 'A', 'b', 'a']
```

5.4.8　内置函数 all()和 any()

通过内置函数 all()和 any()可以判断序列的元素是否全部和部分为 True。函数形式如下：

- all(iterable)：如果序列的所有值都为 True，返回 True；否则，返回 False。
- any(iterable)：如果序列的任意值为 True，返回 True；否则，返回 False。

例如：

```
>>> any((1, 2, 0))                    >>> all([1, 2, 0])
True                                  False
```

5.5 列表

2.7.9 节简单介绍了列表的基本概念，本节继续深入阐述列表的创建和使用。

5.5.1 创建列表实例对象

使用列表字面量可以创建列表实例对象。列表字面量采用方括号中用逗号分隔的项目定义。

【例 5.9】 使用列表字面量创建列表实例对象示例。

```
>>> list1 = []; list2 = [1]; list3 = ["a","b","c"]
>>> print(list1,list2,list3)          #输出：[] [1] ['a', 'b', 'c']
```

用户也可以通过创建 list 对象来创建列表。其基本形式如下。

- list()：创建一个空列表。
- list(iterable)：创建一个列表，包含的项目为可枚举对象 iterable 中的元素。

【例 5.10】 使用 list 对象创建列表实例对象示例。

```
>>> list1 = list(); list2 = list("abc"); list3 = list(range(3))
>>> print(list1,list2,list3)          #输出：[] ['a', 'b', 'c'] [0, 1, 2]
```

5.5.2 列表的序列操作

列表支持序列的基本操作，包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作，以及求列表长度、最大值、最小值等。

列表是可变对象，故用户可以改变列表对象中元素的值，也可以通过 del 删除某元素。

```
s[下标] = x        #设置列表元素，x 为任意对象
del s[下标]        #删除列表元素
```

列表是可变对象，故用户可以改变其切片的值，也可以通过 del 删除切片。

```
s[i:j] = x         #设置列表内容，x 为任意对象，也可以是元组、列表
del s[i:j]         #移去列表一序列元素，等同于 s[i:j] = []
s[i:j] = []        #移去列表一序列元素
```

【例 5.11】　列表的序列操作示例。

```
>>> s = [1,2,3,4,5,6]
>>> s[1] = 'a'
>>> s
[1, 'a', 3, 4, 5, 6]
>>> s[2] = []
>>> s
[1, 'a', [], 4, 5, 6]
>>> del s[3]
>>> s
[1, 'a', [], 5, 6]
>>> s[:2]
[1, 'a']
>>> s[2:3] = []
>>> s
[1, 'a', 5, 6]
>>> s[:1] = []
>>> s
['a', 5, 6]
>>> s[:2] = 'b'
>>> s
['b', 6]
>>> del s[:1]
>>> s
[6]
```

5.5.3　列表对象的方法

列表是可变对象，其包含的主要方法如表 5-1 所示。假设表中的示例基于 s=[1,3,2]。

表 5-1　列表对象的主要方法

方　法	说　明	示　例
s. append(x)	把对象 x 追加到列表 s 尾部	s.append('a')　# s = [1, 3, 2, 'a'] s.append([1,2])　# s = [1, 3, 2, 'a', [1, 2]]
s. clear()	删除所有元素。相当于 del s[:]	s. clear()　# s = []
s. copy()	拷贝列表	s1 = s.copy()　# s1 = s = [1,3,2] id(s),id(s1)　# (3143376592008, 3143376591496)
s. extend(t)	把序列 t 附加到 s 尾部	s.extend([4])　# s = [1, 3, 2, 4] s.extend('ab')　# s = [1, 3, 2, 4, 'a', 'b']
s. insert(i, x)	在下标 i 位置插入对象 x	s. insert(1,4)　# s = [1, 4, 3, 2] s. insert(8,5)　# s = [1, 4, 3, 2, 5]
s. pop([i])	返回并移除下标 i 位置对象，省略 i 时为最后对象。若超出下标，将导致 IndexError	s. pop()　# 输出 2. s = [1, 3] s. pop(0)　# 输出 1. s = [3]
s. remove(x)	移除列表中第一个出现的 x。若对象不存在，将导致 ValueError	s. remove(1)　# s = [3, 2] s. remove('a')　# ValueError: list.remove(x): x not in list
s. reverse()	列表反转	s. reverse()　# s = [2, 3, 1]
s. sort()	列表排序	s. sort()　# s = [1, 2, 3]

5.5.4　列表解析表达式

使用列表解析可以简单高效地处理一个可迭代对象，并生成结果列表。列表解析表达式的形式如下：

- [expr for i_1 in 序列 1…for i_N in 序列 N]：迭代序列里所有内容，并计算生成列表。
- [expr for i_1 in 序列 1…for i_N in 序列 N if cond_expr]：按条件迭代，并计算生成列表。

表达式 expr 使用每次迭代内容 $i_1 \cdots i_N$，计算生成一个列表。如果指定了条件表达式 cond_expr，则只有满足条件的元素参与迭代。

【例 5.12】　列表解析表达式示例。

```
>>> [i**2 for i in range(10)]                                    #平方值
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [(i,i**2) for i in range(10)]                                #序号，平方值
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81)]
>>> [i for i in range(10) if i%2==0]                             #取偶数
[0, 2, 4, 6, 8]
>>> [(x, y, x*y) for x in range(1, 4) for y in range(1, 4) if x>=y]  #二重循环
[(1, 1, 1), (2, 1, 2), (2, 2, 4), (3, 1, 3), (3, 2, 6), (3, 3, 9)]
```

5.5.5 列表的排序

Python 语言提供了下列排序算法。

(1) 内置数据类型 list 中的方法 sort()，把列表中的数据项按升序重新排列。

(2) 内置函数 sorted()则保持原列表不变，返回一个新的包含按升序排列的项的列表。

```
sort( * , key=None, reverse=False)
sorted(iterable, * , key=None, reverse=False)
```

其中，iterable 是待排序的可迭代对象；key 是比较函数(默认为 None，按自然序排序)；reverse 用于指定是否逆序排序。

Python 系统提供的排序方法使用了一种归并排序算法的版本(使用了 Python 无法编写的底层实现)，从而避免了 Python 本身附加的大量开销，故其速度比使用 Python 代码实现的归并排序法快很多(10～20 倍)。系统排序方法同样可以用于任何可比较的数据类型，例如，Python 内置的 str、int 和 float 数据类型。

【例 5.13】 Python 语言提供的排序算法示例。

```
>>> a = [59,12,77,64,72,69,46,89,31,9]
>>> sorted(a)      #输出: [9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
>>> a              #输出: [59, 12, 77, 64, 72, 69, 46, 89, 31, 9]
>>> a.sort(reverse=True)
>>> a              #输出: [89, 77, 72, 69, 64, 59, 46, 31, 12, 9]
```

5.6 元组

2.7.10 节简单介绍了元组的基本概念，本节继续深入阐述元组的创建和使用。

5.6.1 创建元组实例对象

用户可以使用元组字面量创建元组实例对象，也可以通过创建 tuple 对象来创建元组。其基本形式如下：

- tuple()：创建一个空列表。

• tuple(iterable)：创建一个列表，包含的项目为可枚举对象 iterable 中的元素。

【例 5.14】　使用 tuple 对象创建元组实例对象示例。

```
>>> t1 = tuple()
>>> t2 = tuple("abc")
>>> t3 = tuple([1,2,3])
>>> t4 = tuple(range(3))
>>> print(t1,t2,t3,t4)    #输出：() ('a', 'b', 'c') (1, 2, 3) (0, 1, 2)
```

5.6.2　元组的序列操作

元组支持序列的基本操作，包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作，以及求元组长度、最大值、最小值等。

【例 5.15】　元组的序列操作示例。

```
>>> t1 = (1,2,3,4,5,6,7,8,9,10)
>>> len(t1)          #输出：10
>>> max(t1)          #输出：10
>>> sum(t1)          #输出：55
```

5.7　集合

集合数据类型是没有顺序的简单对象的聚集，且集合中元素不重复。Python 集合数据类型包括可变集合对象(set)和不可变集合对象(frozenset)。

5.7.1　集合的定义

可变集合(set)通过花括号中用逗号分隔的项目定义。其基本形式如下：

```
{x1[, x2, …, xn]}
```

其中，x_1、x_2、…、x_n 为任意可 hash 对象。集合中的元素不可重复，且无序，其存储依据对象的 hash 码。hash 码是根据对象的值计算出来的一个唯一值。一个对象如果定义了特殊方法__hash__()，则该对象为可 hash 对象。所有内置不可变对象(bool、int、float、complex、str、tuple、frozenset 等)，都是可 hash 对象；所以内置可变对象(list、dict、set)，都是非 hash 对象(因为可变对象的值可以变化，故无法计算一个唯一的 hash 值)。集合中可以包含内置不可变对象，不能包含内置可变对象。

{}表示空的 dict，因为 dict 也使用花括号定义。空集为 set()。

可变集合也可以通过创建 set 对象来创建，不可变集合通过创建 frozenset 对象来创建。

其基本形式如下：

- set()：创建一个空的可变集合。
- set(iterable)：创建一个可变集合，包含的项目为可枚举对象 iterable 中的元素。
- frozenset()：创建一个空的不可变集合。
- frozenset(iterable)：创建一个不可变集合，包含的项目为可枚举对象 iterable 中的元素。

【例 5.16】 创建集合对象示例。

```
>>> {1,2,1}
{1, 2}
>>> {1,'a',True}
{1, 'a'}
>>> {1.2, True}
{True, 1.2}
>>> set()
set()
>>> frozenset()
frozenset()
>>> set('Hello')
{'H', 'e', 'o', 'l'}
>>> {'a',[1,2]}
Traceback (most recent call last):
  File "< pyshell # 13 >", line 1, in < module >
    {'a',[1,2]}
TypeError: unhashable type: 'list'
```

5.7.2 集合的运算：并集、交集、差集和对称差集

集合支持表 5-2 所示的集合运算。

表 5-2 集合运算

运 算 符	说 明
s1 | s2 | …	返回 s1、s2、…的并集：s1∪s2∪…
s1 & s2 & …	返回 s1、s2、…的交集：s1∩s2∩…
s1 - s2 - …	返回 s1、s2、…的差集，也记作 s1\s2\…
s1 ^ s2	返回 s1、s2 的对称差集：s1△s2

集合的对象方法如表 5-3 所示。

表 5-3 集合的对象方法

方 法	说 明
s1.isdisjoint(s2)	如果集合 s1 和 s2 没有共同元素，返回 True；否则返回 False
s1.issubset(s2)	如果集合 s1 是 s2 的子集，返回 True；否则返回 False
s1.issuperset(s2)	如果集合 s1 是 s2 的超集，返回 True；否则返回 False
s1.union(s2, …)	返回 s1、s2、…的并集：s1∪s2∪…
s1.intersection(s2, …)	返回 s1、s2、…的交集：s1∩s2∩…
s1.difference(s2, …)	返回 s1、s2、…的差集：s1－s2－…
s1.symmetric_difference(s2)	返回 s1 和 s2 的对称差集：s1△s2

【例 5.17】 集合的运算示例。

```
>>> s1 = {1,2,3}
>>> s2 = {2,3,4}
>>> s1 | s2
{1, 2, 3, 4}
>>> s1 & s2
{2, 3}
>>> s1 - s2
{1}
>>> s1 ^ s2
{1, 4}
>>> s1.union(s2)
{1, 2, 3, 4}
>>> s1.intersection(s2)
{2, 3}
>>> s1.difference(s2)
{1}
>>> s1.symmetric_difference(s2)
{1, 4}
```

5.7.3　可变集合的方法

set 集合是可变对象，包含的主要方法如表 5-4 所示。假设表中的示例基于“s1＝{1，2，3}；s2＝{2，3，4}”。

表 5-4　可变集合对象的主要方法

方　法	说　明	示　例
s1.update(s2，…) s1 |＝s2 | …	并集 s1＝ s1∪s2∪…	>>> s1.update(s2) # s1 = {1, 2, 3, 4}
s1.intersection_update(s2，…) s1 &＝ s2 & …	交集 s1＝ s1∩s2∩…	>>> s1.intersection_update(s2) # s1 = {2, 3}
s1.difference_update(s2，…) s1 －＝ s2 －…	差集 s1＝ s1－s2－…	>>> s1.difference_update(s2) # s1 = {1}
s1.symmetric_difference_update(s2) s1 ^＝ s2	对称差集 s1＝ s1△s2	s1.symmetric_difference_update(s2) # s1 = {1, 4}
s.add(x)	把对象 x 添加到集合 s	>>> s1.add('a') # s1 = {1, 2, 3, 'a'}
s.remove(x)	从集合 s 中移除对象 x。若不存在，则导致 KeyError	>>> s1.remove(1) # s1 = {2, 3}
s.discard(x)	从集合 s 中移除对象 x（如果存在的话）	>>> s1.discard(3) # s1 = {1, 2}
s.pop()	从集合 s 随机弹出一个元素，如果 s 为空，则导致 KeyError	>>> s1.pop()#输出：1.s1 = {2, 3}
s.clear()	清空集合 s	>>> s1.clear()　# s1 = set()

5.8　字典（映射）

2.7.11 节简单介绍了字典的基本概念，本节继续深入阐述字典的创建和使用。

5.8.1　对象的 hash 值

字典是键和值的映射关系。字典的键必须是可 hash 的对象，即实现了__hash__()的对象。一个对象的 hash 值也可以使用内置函数 hash()获得。

【例 5.18】　对象的 hash 值示例。

```
>>> hash(100)   #结果：100
>>> hash(1.23)  #结果：530343892119149569
>>> hash('abc') #结果：901130859749610928
```

不可变对象（bool、int、float、complex、str、tuple、frozenset 等）是可 hash 对象；而可变

对象通常是不可 hash 对象，因为不可变对象的内容可以改变，因而无法通过 hash()函数获取其 hash 值。

字典的键只能使用不可变的对象，但字典的值可以使用不可变或可变的对象。一般而言，应该使用简单的对象作为键。

5.8.2 字典的创建

用户可以使用字典字面量创建字典实例对象，也可以通过创建 dict 对象来创建字典。其基本形式如下：

- dict()：创建一个空字典。
- dict(** kwargs)：使用关键字参数，创建一个新的字典。此方法最紧凑。
- dict(mapping)：从一个字典对象创建一个新的字典。
- dict(iterable)：使用序列，创建一个新的字典。

【例 5.19】 创建字典对象示例。

```
>>> {}                              >>> dict({1:'food', 2:'drink'})
{}                                  {1: 'food', 2: 'drink'}
>>> {'a':'apple','b':'boy'}         >>> dict([('id','1001'),('name','Jenny')])
{'a': 'apple', 'b': 'boy'}          {'id': '1001', 'name': 'Jenny'}
>>> dict()                          >>> dict(baidu = 'baidu.com', google = 'google.com')
{}                                  {'baidu': 'baidu.com', 'google': 'google.com'}
```

5.8.3 字典的访问操作

字典 d 可以通过键 key 来访问，其基本形式如下。

- d[key]：返回键为 key 的 value；如果 key 不存在，则导致 KeyError。
- d[key] = value：设置 d[key]的值为 value；如果 key 不存在，则添加键/值对。
- del d[key]：删除字典元素；如果 key 不存在，则导致 KeyError。

字典 d 支持下列视图对象，通过它们可以动态访问字典的数据，其基本形式如下：

- d.keys()：返回字典 d 的键 key 的列表。
- d.values()：返回字典 d 的值 value 的列表。
- d.items()：返回字典 d 的(key, value)对的列表。

字典 d 及其视图 d.items()、d.values()、d.keys()都是可迭代对象，用户可以使用 for 循环进行迭代。

【例 5.20】 字典对象的访问操作示例。

```
>>> d = {1:'food', 2:'drink', 3:'fruit'}
>>> d[1]       #输出'food'
>>> for k in d:
        print("键 = {},值 = {};".format(k, d
[k]),end = " ")
键 = 1,值 = food; 键 = 2,值 = drink; 键 = 3,值
 = fruit;
>>> for k, v in d.items():
        print("键 = {},值 = {};".format(k, v),
end = " ")
键 = 1,值 = food; 键 = 2,值 = drink; 键 = 3,值
 = fruit;
```

5.8.4　字典对象的方法

字典是可变对象，其包含的主要方法如表 5-5 所示。假设表中的示例基于 d={1：'food'，2：'drink'，3：'fruit'}。

表 5-5　字典对象的主要方法

方　法	说　明	示　例
d. clear()	删除所有元素	>>> d.clear();d　#结果：{}
d. copy()	浅拷贝字典	>>> d1 = d.copy(); id(d), id(d1) (2487537820800, 2487537277976)
d. get(k)	返回键 k 对应的值，如果 key 不存在，返回 None	>>> d.get(1),d.get(5) ('food', None)
d. get(k, v)	返回键 k 对应的值，如果 key 不存在，返回 v	>>> d.get(1,'无'),d.get(5,'无') ('food', '无')
d. pop(k)	如果键 k 存在，返回其值，并删除该项目；否则导致 KeyError	>>> d.pop(1), d ('food', {2: 'drink', 3: 'fruit'})
d. pop(k, v)	如果键 k 存在，返回其值，并删除该项目；否则返回 v	>>> d.pop(5,'无'), d ('无', {1: 'food', 2: 'drink', 3: 'fruit'})
d. popitem()	以 LIFO(后进先出)方式删除字典中的一个键值对，并且以(key,value)形式返回该键值对。如果字典为空，将导致 KeyError	>>> d.popitem(),d ((3, 'fruit'), {1: 'food', 2: 'drink'})

d. setdefault(k, v)	如果键 k 存在，返回其值；否则添加项目 k= v,v 默认为 None	>>> d.setdefault(1)#结果：'food'
		>>> d.setdefault(4);d
		{1: 'food', 2: 'drink', 3: 'fruit', 4: None}

d. update([other])	使用字典或键值对，更新或添加项目到字典 d	>>> d1 = {1: '食物', 4: '书籍'}
		>>> d.update(d1);d
		{1: '食物', 2: 'drink', 3: 'fruit', 4: '书籍'}

5.9　算法基础

5.9.1　算法概述

算法是指解决问题的一种方法或一个过程。算法通常使用计算机程序来实现。算法接手待处理的输入数据；然后执行相应的处理过程；最后输出处理的结果。

算法的实现为若干指令的有穷序列，具有如下性质。

(1) 输入数据。算法可以接收用于处理的外部数据。

(2) 输出结果。算法可以产生输出结果。

(3) 确定性。算法的组成指令必须是准确无歧义。

(4) 有限性。算法指令的执行次数必须是有限的，执行的时间也必须是有限的。

在计算机上执行一个算法，会产生内存开销和时间开销。算法的性能分析包括时间性

能分析和空间性能分析两个方面。

5.9.2 算法的时间复杂度分析

衡量算法有效性的一个指标是运行时间。算法的运行时间长度，与算法本身的设计和所求解的问题的规模有关。算法的时间性能分析，又称之为算法的时间复杂度（time complexity）分析。

问题的规模（size）即算法求解问题的输入量，通常用一个整数表示。例如，矩阵乘积问题的规模是矩阵的阶数，图论问题的规模则是图中的顶点数或边数。

对于问题规模较大的数据，如果算法的时间复杂度呈指数分布，完成算法的时间可能趋向于无穷大，即无法完成。

一个算法运行的总时间取决于以下两个主要因素。

（1）每条语句的执行时间成本。

（2）每条语句的执行次数（频度）。

即一个算法所耗费的时间等于算法中每条语句的执行时间之和。每条语句的执行时间为该语句的执行次数（频度）×该语句执行一次所需时间。

每条语句执行一次所需的时间取决于实际运行程序的机器性能。独立于机器系统分析算法的时间性能时，可以假设每条语句执行一次所需的时间均是单位时间，故一个算法的运行时间等于算法中所有语句的频度之和。

【例 5.21】 算法中语句的频度之和示例（frequency.py）。

```
total = 0
for i in range(n):
    for j in range(n):
        total += a[i][j]
print(total)
```

在例 5.21 中，循环语句运行了 n×n 次，总算法执行语句频度为 n^2+2。

5.9.3 增长量级

对于问题规模 n，假如算法 A 中所有语句的频度之和为 100n+1；算法 B 中所有语句的频度之和为 n^2+n+1，则算法 A 和 B 对于不同问题规模的运行时间对照表如表 5-6 所示。

表 5-6 算法 A 和 B 对于不同问题规模的运行时间对照表

问题规模 n	算法 A 运行时间	算法 B 运行时间
10	1001	111
100	10001	10101
1000	100001	1001001
10000	1000001	100010001

由表 5-6 可以看出，随着问题规模 n 的增长，算法的运行时间主要取决于最高指数项。

在算法分析中，通常使用增长量级来描述。

增长量级用于描述函数的渐进增长行为，一般使用大 O 符号表示。例如，2n、100n 与 n+1 属于相同的增长量级，记为 O(n)，表示函数随 n 线性增长。

算法分析中常用的增长量级如表 5-7 所示。

表 5-7　常用的增长量级

函数类型	增长量级	举　例	说　明
常量型	1	`count - = 1`	语句(整数递减)

对数型	$\log_2 N$	`while n > 0:`
		`  n = n // 2`
		`  count += 1`
			折半(二分查找法等)

线性型	n	`for i in range(n):`
		`  if i % 2 != 0: # 奇数`
		`  sum_odd += i`
			单循环
			(统计奇数的个数、顺序查找法等)

线性对数型	$N\log_2 N$	请参见 5.10.6 节归并排序法	分而治之算法(归并排序法等)

二次型	n^2	`for i in range(1, n):`
		`  s = ""`
		`  for j in range(1, n):`
		`  s += str.format("{0:1} * {1:1} = {2:2}", i, j, i * j)`
		`  print(s)`
			两重嵌套循环(打印九九乘法表、冒泡排序算法、选择排序算法、插入排序算法等)

三次型	n^3	`for i in range(n):`
		`  for j in range(i + 1, n):`
		`  for k in range(j + 1, n):`
		`  if (a[i] + a[j] + a[k]) == 0:`
		`  count += 1`
			三重嵌套循环

5.9.4　算法的空间复杂度分析

衡量算法有效性的另一个指标是内存消耗。对于复杂的算法，如果其消耗的内存超过运行该算法的计算机的可用物理内存，则算法无法正常执行。算法的内存消耗分析，又称之为算法的空间复杂度(space complexity)分析。

Python 语言面向对象特性的主要代价之一是内存消耗。Python 的内存消耗与其在不同计算机上的实现有关。不同版本的 Python 有可能使用不同方法实现同一种数据类型。

确定一个 Python 程序内存使用的典型方法是，先统计程序所使用对象的数量，然后根据对象的类型乘以各对象占用的字节数。使用函数 sys.getsizeof(x)，可以返回一个内置数据类型 x 在系统中所占用的字节数。

【例 5.22】　Python 语言中对象占用内存大小示例。

```
>>> import sys
>>> sys.getsizeof(100)      # 整数对象占用内存大小.输出：28
>>> sys.getsizeof("1.23")   # 字符串对象占用内存大小.输出：53
>>> sys.getsizeof(True)     # 布尔逻辑型对象占用内存大小.输出：28
```

5.10 常用的查找和排序算法

5.10.1 顺序查找法

查找算法是在程序设计中最常用到的算法。假定要从 n 个元素中查找 x 的值是否存在，最原始的办法是从头到尾逐个查找，这种查找方法称为顺序查找法。

顺序查找算法有三种情形可能发生：最好的情况下，第一个项就是我们要找的数据对象，只有一次比较；最差的情况下，需要 n 次比较，全部比较完之后查不到数据；平均情况下，比较次数为 n/2 次。即算法的时间复杂度为 O(n)。

【例 5.23】 在列表中顺序查找特定数值 x(sequentialSearch.py)。

```
def sequentialSearch(alist, item):                  # 顺序查找法
    pos = 0                                         # 初始查找位置
    found = False                                   # 未找到数据对象
    while pos < len(alist) and not found:           # 列表未结束并且还未找到则一直循环
        if alist[pos] == item:                      # 找到匹配对象,返回 True
            found = True
        else:                                       # 否则查找位置 + 1
            pos = pos + 1
    return found
def main():
    testlist = [1, 3, 33, 8, 37, 29, 32, 15, 5]     # 测试数据列表
    print(sequentialSearch(testlist, 3))            # 查找数据 3
    print(sequentialSearch(testlist, 13))           # 查找数据 13
if __name__ == '__main__': main()
```

程序运行结果如下：

```
True
False
```

5.10.2 二分查找法

二分查找法又称折半查找法，用于预排序列表的查找问题。

要在排序列表 alist 中查找元素 t，首先，将列表 alist 中间位置的项与查找关键字 t 比较，如果两者相等，则查找成功；否则利用中间项将列表分成前、后两个子表，如果中间位置项目大于 t，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到

满足条件的记录，即查找成功；或者直到子表不存在为止，即查找不成功。

对于包含 N 个元素的列表，其时间复杂度为 $O(\log_2 N)$。

【例 5.24】　二分查找法的递归实现(binarySearch.py)。

```
def _binarySearch(key, a, lo, hi):
    if hi <= lo: return -1                          #查找失败,返回-1
    mid = (lo + hi) // 2                            #计算中间位置
    if a[mid] > key:                                #中间位置项目大于查找关键字
        return _binarySearch(key, a, lo, mid)       #递归查找前一子表
    elif a[mid] < key:                              #中间位置项目小于查找关键字
        return _binarySearch(key, a, mid+1, hi)     #递归查找后一子表
    else:                                           #中间位置项目等于查找关键字
        return mid                                  #查找成功,返回下标位置
def binarySearch(key, a):                           #二分查找
    return _binarySearch(key, a, 0, len(a))         #递归二分查找法
def main():
    a = [1,13,26,33,45,55,68,72,83,99]
    print("关键字位于列表索引",binarySearch(33, a)) #二分查找关键字 33
    print("关键字位于列表索引",binarySearch(58, a)) #二分查找关键字 58
if __name__ == '__main__': main()
```

程序运行结果如下：

```
关键字位于列表索引 3
关键字位于列表索引 -1
```

5.10.3　冒泡排序法

冒泡排序法是最简单的排序算法。对于包含 N 个元素的列表 A，按递增顺序排序的冒泡法的算法如下：

(1) 第 1 轮比较：从第一个元素开始，对列表中所有 N 个元素进行两两大小比较，如果不满足升序关系，则交换。即 A[0]与 A[1]比较，若 A[0]>A[1]，则 A[0]与 A[1]交换；然后 A[1]与 A[2]比较，若 A[1]>A[2]，则 A[1]与 A[2]交换；直至最后 A[N−2]与 A[N−1]比较，若 A[N−2]>A[N−1]，则 A[N−2]与 A[N−1]交换。第一轮比较完成后，列表元素中最大的数“沉”到列表最后，而那些较小的数如同气泡一样上浮一个位置，顾名思义“冒泡法”排序。

(2) 第 2 轮比较：从第一个元素开始，对列表中前 N−1 个元素(第 N 个元素，即 A[N−1]已经最大，无须参加排序)继续两两大小比较，如果不满足升序关系，则交换。第二轮比较完成后，列表元素中次大的数“沉”到最后，即 A[N−2]为列表元素中次大的数。

(3) 以此类推，进行第 N−1 轮比较后，列表中所有元素均按递增顺序排好序。

若要按递减顺序对列表排序，则每次两两大小比较时，如果不满足降序关系，则交换即可。

冒泡排序法的过程如表 5-8 所示。

表 5-8　冒泡排序法示例

原始列表	2	97	86	64	50	80	3	71	8	76
第 1 轮比较	2	86	64	50	80	3	71	8	76	97
第 2 轮比较	2	64	50	80	3	71	8	76	86	97
第 3 轮比较	2	50	64	3	71	8	76	80	86	97
第 4 轮比较	2	50	3	64	8	71	76	80	86	97
第 5 轮比较	2	3	50	8	64	71	76	80	86	97
第 6 轮比较	2	3	8	50	64	71	76	80	86	97
第 7 轮比较	2	3	8	50	64	71	76	80	86	97
第 8 轮比较	2	3	8	50	64	71	76	80	86	97
第 9 轮比较	2	3	8	50	64	71	76	80	86	97

冒泡排序算法的主要时间消耗是比较次数。当 i=1 时，比较次数为 N−1；当 i=2 时，比较次数为 N−2；以此类推。总共比较次数为(N−1)+(N−2)+…+2+1=N(N−1)/2，故冒泡排序算法的时间复杂度为 $O(N^2)$。

【例 5.25】　冒泡排序算法的实现(bubbleSort.py)。

```
def bubbleSort(a):
    for i in range(len(a) - 1, - 1, - 1):                    # 外循环
        for j in range(i):                                   # 内循环
            if a[j] > a[j + 1]:                              # 大数往下沉
                a[j], a[j + 1] = a[j + 1], a[j]
def main():
    a = [2,97,86,64,50,80,3,71,8,76]
    bubbleSort(a)
    print(a)
if __name__ == '__main__': main()
```

程序运行结果如下：

```
[2, 3, 8, 50, 64, 71, 76, 80, 86, 97]
```

5.10.4　选择排序法

对于包含 N 个元素的列表 A，按递增顺序排序的选择法的基本思想是：每次在若干无序数据中查找最小数，并放在无序数据中的首位。其算法如下：

(1) 从 N 个元素的列表中找最小值及其下标，最小值与列表的第 1 个元素交换。

(2) 从列表的第 2 个元素开始的 N−1 个元素中再找最小值及其下标，该最小值(即整个列表元素的次小值)与列表第 2 个元素交换。

(3) 以此类推，进行第 N−1 轮选择和交换后，列表中所有元素均按递增顺序排好序。

若要按递减顺序对列表排序，只要每次查找并交换最大值即可。

选择排序法的过程如表 5-9 所示。

表 5-9　选择排序法示例

原始数组	59	12	77	64	72	69	46	89	31	9
第 1 轮比较	9	12	77	64	72	69	46	89	31	59
第 2 轮比较	9	12	77	64	72	69	46	89	31	59
第 3 轮比较	9	12	31	64	72	69	46	89	77	59
第 4 轮比较	9	12	31	46	72	69	64	89	77	59
第 5 轮比较	9	12	31	46	59	69	64	89	77	72
第 6 轮比较	9	12	31	46	59	64	69	89	77	72
第 7 轮比较	9	12	31	46	59	64	69	89	77	72
第 8 轮比较	9	12	31	46	59	64	69	72	77	89
第 9 轮比较	9	12	31	46	59	64	69	72	77	89

选择排序算法的主要时间消耗是比较次数。当 i=1 时，比较次数为 N−1；当 i=2 时，比较次数为 N−2；以此类推。总共比较次数为(N−1)+(N−2)+…+2+1=N (N−1)/2，故选择排序算法的时间复杂度为 $O(N^2)$。

【例 5.26】　选择排序算法的实现(selectionSort. py)。

```
def selectionSort(a):
    for i in range(0, len(a)):          #外循环(0～N－1)
        m = i                           #当前位置下标
        for j in range(i + 1, len(a)):#内循环
            if a[j] < a[m]:             #查找最小值的位置
                m = j
        a[i], a[m] = a[m], a[i]         #元素交换
def main():
    a = [59,12,77,64,72,69,46,89,31,9]
    selectionSort(a)
    print(a)
if __name__ == '__main__': main()
```

程序运行结果如下：

```
[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

5.10.5　插入排序法

对于包含 N 个元素的列表 A，按递增顺序排序的插入排序法的基本思想是：依次检查列表中的每个元素，将其插入到其左侧已经排好序的列表中的适当位置。其算法如下。

(1) 第 2 个元素与列表中其左侧的第 1 个元素比较，如果 A[0]>A[1]，则交换位置，结果左侧 2 个元素排序完毕。

(2) 第 3 个元素依次与其左侧的列表的元素比较，直至插入对应的排序位置，结果左侧

的 3 个元素排序完毕。

(3) 以此类推，进行第 N－1 轮比较和交换后，列表中所有元素均按递增顺序排好序。

若要按递减顺序对列表排序，只要每次查找并交换最大值即可。

插入排序法的过程如表 5-10 所示。

表 5-10　插入排序法示例

原始数组	59	12	77	64	72	69	46	89	31	9
第 1 轮比较	12	59	77	64	72	69	46	89	31	9
第 2 轮比较	12	59	77	64	72	69	46	89	31	9
第 3 轮比较	12	59	64	77	72	69	46	89	31	9
第 4 轮比较	12	59	64	72	77	69	46	89	31	9
第 5 轮比较	12	59	64	69	72	77	46	89	31	9
第 6 轮比较	12	46	59	64	69	72	77	89	31	9
第 7 轮比较	12	46	59	64	69	72	77	89	31	9
第 8 轮比较	12	31	46	59	64	69	72	77	89	9
第 9 轮比较	9	12	31	46	59	64	69	72	77	89

在最理想的情况下(列表处于排序状态)，while 循环仅仅需要一次比较，故总的运行时间为线性；在最差情况下(列表为逆序状态)，此时内循环指令执行次数为 1＋2＋…＋N－1＝N(N－1)/2，故插入排序算法的时间复杂度为 $O(N^2)$。

【例 5.27】　插入排序算法的实现(insertSort.py)。

```
def insertSort(a):
    for i in range(1, len(a)):                  #外循环(1～N－1)
        j = i
        while (j > 0) and (a[j] < a[j－1]):     #内循环
            a[j], a[j－1] = a[j－1], a[j]       #元素交换
            j －= 1                             #继续循环
def main():
    a = [59,12,77,64,72,69,46,89,31,9]
    insertSort(a)
    print(a)
if __name__ == '__main__': main()
```

程序运行结果如下：

```
[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

5.10.6　归并排序法

归并排序法基于分而治之(divide and conquer)的思想。算法的操作步骤如下。

(1) 将包含 N 个元素的列表分成两个含 N/2 元素的子列表。

（2）对两个子列表递归调用归并排序（最后可以将整个列表分解成 N 个子列表）。

（3）合并两个已排序好的子列表。

假设列表 a = [59,12,77,64,72,69,46,89,31,9]，归并算法的示意图如图 5-10 所示。

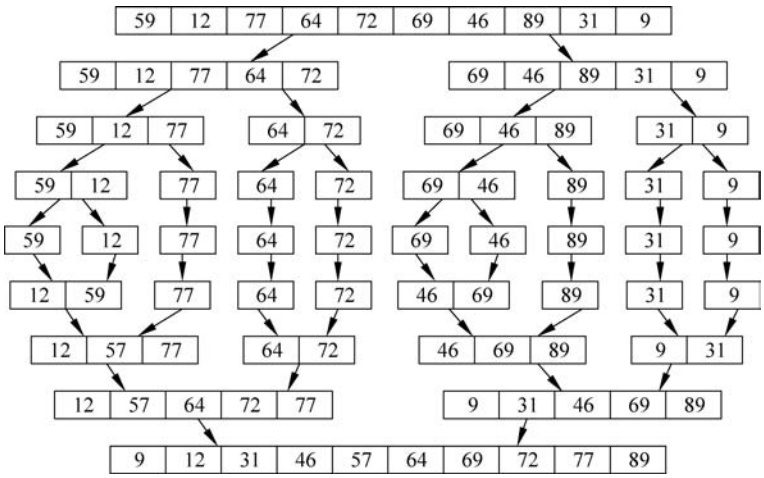


图 5-10　归并排序法示意图

对于长度为 N 的列表，归并排序算法将列表分开成子列表一共要 $\log_2 N$ 步。每步都是一个合并有序列表的过程，时间复杂度可以记为 O(N)，故归并排序算法的时间复杂度为 O($N\log_2 N$)。其效率是比较高的。

【例 5.28】　归并排序算法的实现（mergeSort.py）。

```
def merge(left, right):                          #合并两个列表
    merged = []
    i, j = 0, 0                                  #i 和 j 分别作为 left 和 right 的下标
    left_len, right_len = len(left), len(right)  #分别获取左右子列表的长度
    while i < left_len and j < right_len:        #循环归并左右子列表元素
        if left[i] <= right[j]:
            merged.append(left[i])               #归并左子列表元素
            i += 1
        else:
            merged.append(right[j])              #归并右子列表元素
            j += 1
    merged.extend(left[i:])                      #归并左子列表剩余元素
    merged.extend(right[j:])                     #归并右子列表剩余元素
    return merged                                #返回归并好的列表
def mergeSort(a):                                #归并排序
    if len(a) <= 1:                              #空或者只有 1 个元素，直接返回列表
        return a
    mid = len(a) // 2                            #列表中间位置
```

```
    left = mergeSort(a[:mid])              #归并排序左子列表
    right = mergeSort(a[mid:])             #归并排序右子列表
    return merge(left, right)              #合并排好序的左右两个子列表
def main():
    a = [59,12,77,64,72,69,46,89,31,9]
    a1 = mergeSort(a)
    print(a1)
if __name__ == '__main__': main()
```

程序运行结果如下：

```
[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

5.10.7 快速排序法

快速排序是对冒泡排序的一种改进，由 C. A. R. Hoare 在 1962 年提出。其基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小；然后递归对这两部分数据分别进行快速排序。

快速排序算法的一趟排序的操作步骤如下：

(1) 设置两个变量 i 和 j，分别为列表首末元素的下标，即 i=0，j=N−1。

(2) 设置列表的第 1 个元素作为关键数据，即 key=A[0]。

(3) 从 j 开始向前搜索，找到第一个小于 key 的值 A[j]，将 A[j]和 A[i]互换。

(4) 从 i 开始向后搜索，找到第一个大于 key 的 A[i]，将 A[i]和 A[j]互换。

(5) 重复第(3)和(4)步，直到 i=j。

假设列表 a = [59,12,77,64,72,69,46,89,31,9]，快速排序算法的一趟排序示例如表 5-11 所示。

表 5-11 快速排序算法的一趟排序示例

下标	0	1	2	3	4	5	6	7	8	9
原始数组 i=0，j=9，key=59	59	12	77	64	72	69	46	89	31	9
第 1 轮比较交换 i=0，j=9	9	12	77	64	72	69	46	89	31	59
第 2 轮比较交换 i=2，j=9	9	12	59	64	72	69	46	89	31	77
第 3 轮比较交换 i=2，j=8	9	12	31	64	72	69	46	89	59	77
第 4 轮比较交换 i=3，j=8	9	12	31	59	72	69	46	89	64	77
第 5 轮比较交换 i=3，j=6	9	12	31	46	72	69	59	89	64	77
第 6 轮比较交换 i=4，j=6	9	12	31	46	59	69	72	89	64	77
第 7 轮比较交换 i=4，j=4	9	12	31	46	59	69	72	89	64	77

快速排序最坏情况下，每次划分选取的基准都是当前无序列表中关键字最小(或最大)的记录，时间复杂度为 $O(N^2)$；平均情况下，其时间复杂度为 $O(N\log_2 N)$。

【例 5.29】　快速排序算法的实现(quickSort.py)。

```
def quickSort(a, low, high):          #对列表 a 快速排序,列表下界为 low,上界为 high
    i = low                           #i 等于列表下界
    j = high                          #j 等于列表上界
    if i >= j:                        #如果下界大于等于上界,返回结果列表 a
        return a
    key = a[i]                        #设置列表的第 1 个元素作为关键数据
    while i < j:                      #循环直到 i = j
        while i < j and a[j] >= key:  #j 开始向前搜索,找到第一个小于 key 的值 a[j]
            j = j - 1
        a[i] = a[j]
        while i < j and a[i] <= key:  #i 开始向后搜索,找到第一个大于 key 的 a[i]
            i = i + 1
        a[j] = a[i]
    a[i] = key                        #a[i]等于关键数据
    quickSort(a, low, i - 1)          #递归调用快速排序算法(列表下界为 low,上界为 i - 1)
    quickSort(a, j + 1, high)         #递归调用快速排序算法(列表下界为 j + 1,上界为 high)
def main():
    a = [59,12,77,64,72,69,46,89,31,9]
    quickSort(a, 0, len(a) - 1)
    print(a)
if __name__ == '__main__': main()
```

程序运行结果如下：

```
[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

5.11　应用举例

5.11.1　基于列表的简易花名册管理系统

通过列表可以很方便实现一个花名册管理系统，实现名字的显示、查询、增加、删除、修改等功能。

【例 5.30】　简易花名册管理系统(name_list.py)。

```
def menu():                  #显示菜单
    print(" = " * 35)
    print("简易花名册程序")
    print("1.显示名字")
    print("2.添加名字")
    print("3.删除名字")
```

```
    print("4.修改名字")
    print("5.查询名字")
    print("6.退出系统")
    print("="*35)
names = []                                          #创建存储花名册的列表对象
while True:                                         #重复执行
    menu()                                          #显示菜单
    num = input("请输入选择的功能序号(1到6):")      #获取用户输入
    #根据用户选择,执行相应的功能
    if num == '1':
        print(names)
    elif num == '2':
        name = input("请输入要添加的名字:")
        names.append(name)
        print(names)
    elif num == '3':
        name = input("请输入要删除的名字:")
        if name in names:
            names.remove(name)
        else:
            print("系统中不存在名字:{}".format(name))
        print(names)
    elif num == '4':
        name = input("请输入要修改的名字:")
        if name in names:
            index = names.index(name)
            new_name = input("请输入修改后的名字:")
            names[index] = new_name
        else:
            print("系统中不存在名字:{}".format(name))
        print(names)
    elif num == '5':
        name = input("请输入要查询的名字:")
        if name in names:
            print("系统中存在名字:{}".format(name))
        else:
            print("系统中不存在名字:{}".format(name))
    elif num == '6':
        break
    else:
        print("选项错误,请重新选择!")
```

程序运行结果如图 5-11 所示。

```
==========
简易花名册程序
1. 显示名字
2. 添加名字
3. 删除名字
4. 修改名字
5. 查询名字
6. 退出系统
==========
请输入选择的功能序号（1到6）：
```

图 5-11　简易花名册管理系统

5.11.2　频数表和直方图

构建频数表的方法可以使用字典(键为项,值为计数)。通过遍历数据列表,在频数字典中递增对应项的值,可以很容易地实现频数表。

绘制直方图可以使用前面章节介绍的海龟对象。在前文的海龟绘图示例中,当创建一个新的海龟对象时,将自动创建一个海龟窗口。使用 turtle 模块中的 Screen()构造函数,还可以单独创建绘图窗口或者屏幕,然后添加海龟对象,以实现调用方法来自定义绘图窗口。

turtle 模块中的 Screen()构造函数及相关绘图函数的语法和调用形式如下。

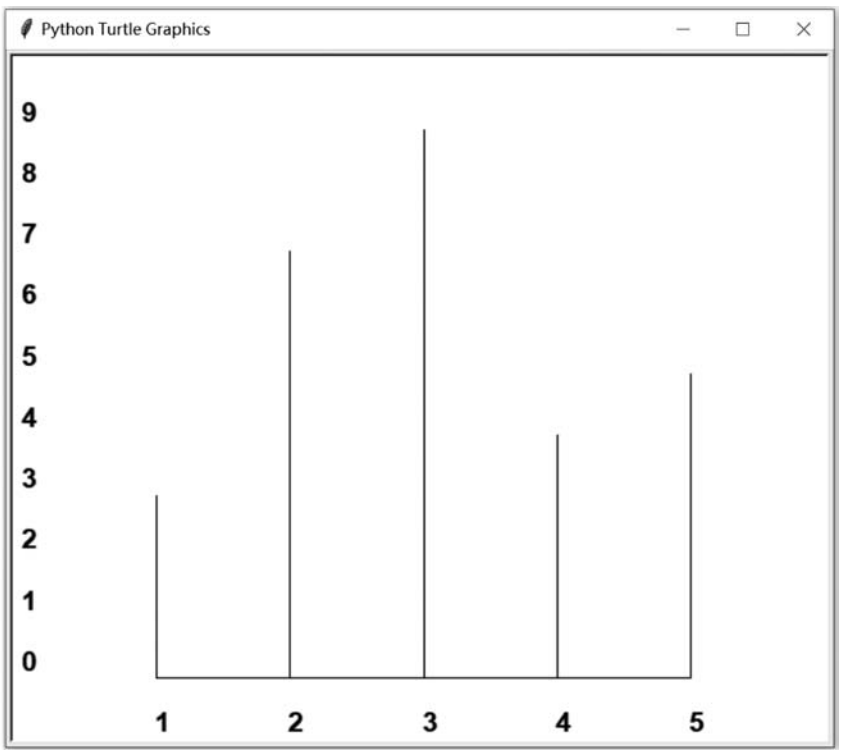
- wn = turtle.Screen()：创建新的绘图窗口。
- wn.setworldcoordinates(xLL, yLL, xUR, yUR)：调整窗口坐标,使窗口左下角的点为(xLL, yLL),窗口右上角的点为(xUR, yUR)。
- wn.exitonclick()：当用户在窗口中的某个位置单击鼠标时,关闭窗口。

【例 5.31】　构建给定列表数据的频数表,并绘制直方图(freq.py)。

```
import turtle
def freq_table(data_list):          #统计列表 data_list 中的值的个数,返回包含值计数的字典
    countDict = {}                  #创建储存频数的字典 k: 值,v: 计数
    for item in data_list:
        countDict[item] = countDict.get(item,0) + 1
    return countDict
def draw_freq(freq_dict):           #绘制值计数的字典 freq_dict 的直方图
    itemList = list(freq_dict.keys())      #获取键的列表
    maxItem = len(itemList) - 1            #获取最大项目数
    itemList.sort()
    countList = freq_dict.values()         #获取计数的列表
    maxCount = max(countList)              #获取最大的计数值
    #使用海龟对象,绘制直方图
    wn = turtle.Screen()                   #创建海龟绘图窗口
    t = turtle.Turtle()                    #创建海龟对象
    wn.setworldcoordinates(-1, -1, maxItem + 1, maxCount + 1)      #设置绘图窗口大小
    t.hideturtle()                         #隐藏海龟
    t.up();t.goto(0, 0);t.down()           #绘制基准线(X 轴)
    t.goto(maxItem, 0); t.up()
    for i in range(0,maxCount + 1):        #绘制 Y 轴标签
```

```
            t.goto( - 1, i)
            t.write(str(i), font = ("Helvetica", 16, "bold"))
        for index in range(len(itemList)):
            t.goto(index, - 1)                    #绘制标签
            t.write(str(itemList[index]),font = ("Helvetica",16,"bold"))
            t.goto(index, 0)                      #绘制计数线条(高度)
            t.down()
            t.goto(index, freq_dict[itemList[index]])
            t.up()
        wn.exitonclick()
data = [3,1,2,1,3,1,2,2,3,5,3,5,4,5,3,4,5,2,3,3,2,2,3,4,2,5,4,3]
freq_dict = freq_table(data)                      #返回频数字典
#打印结果
itemList = list(freq_dict.keys())
itemList.sort()
print("值", "计数")
for item in itemList:
    print(item, " ", freq_dict[item])
draw_freq(freq_dict)
```

程序运行结果如图 5-12 所示。



值	计数
1	3
2	7
3	9
4	4
5	5

图 5-12　给定列表数据频数表的直方图

本章小结


组合数据和数据结构
数据结构基础
数据结构的定义
数据的逻辑结构
数据的物理结构
常用算法
常用的数据结构
线性表
队列
栈
树
图
堆
散列表
Python序列数据类型
序列数据类型
序列的长度、最大值和最小值
序列的索引访问操作
序列的切片操作
序列的连接和重复操作
序列的成员关系操作
序列的比较运算操作
序列排序操作
列表
创建列表实例对象
列表的序列操作
列表的对象方法
列表解析表达式
列表的排序
元组
创建元组对象
元组的系列操作
集合
集合的定义
集合的运算
可变集合的方法
字典
对象的哈希值
字典的创建
字典的访问操作
字典对象的方法
算法基础
时间复杂度
空间复杂度
常用的查找和排序算法
顺序查找算法
二分查找算法
冒泡排序法
选择排序法
插入排序法
归并排序法
快速排序法