第3章 编程基础

本章重点内容：Python 的基础语法、变量和常量、Python 中的运算符以及常用运算、Python 中的表达式。

本章学习要求：通过本章的学习，初步理解 Python 的基础语法，为以后学习 Python 编程打下基础。

3.1 通过蒙特卡洛方法计算圆周率π的值

蒙特卡洛方法(Monte Carlo method)也可以称为统计模拟方法，是 20 世纪 40 年代中期由蒙特卡洛提出的一种非常重要的数值计算方法。它以概率统计理论为指导，使用随机数来解决计算问题。

【例 3-1】 利用蒙特卡洛方法计算圆周率 π 的值。计算原理：在边长为 $2r$ 的正方形内部画一个半径为 r 的圆，随机向正方形内撒点，有些点落在圆外，有些点落在圆内，落在圆内外的点的个数分别是 x 和 y，当点的数目足够多时，可以利用点的数量代表正方形和圆的面积，公式如下：


观看视频

$$\frac{x}{y}=\frac{\pi r^2}{(2r)^2}=\frac{\pi r^2}{4r^2}=\frac{\pi}{4}$$

可得：

$$\pi=\frac{4x}{y}$$

从上述公式中可以看出，π 的值可以通过落在图中点的个数计算出来。

参考代码：

```
import numpy as np                    #引入第三方库 Numpy
import matplotlib.pyplot as plt       #引入第三方库 Matplotlib 中的 pyplot 模块,并取别名为 plt
from matplotlib.patches import Circle
n = 10000                             #投点次数
#设置圆的属性,半径为 1
radius = 1.0                          #设置半径
a, b = (0., 0.)                       #圆心位置坐标
#正方形位置属性设置,边长为 2
x_left, x_right = a - radius, a + radius
y_down, y_up = b - radius, b + radius
#在正方形区域内随机投点
```

```
x = np.random.uniform(x_left, x_right, n)
#从一个均匀分布的[x_left,x_right)中随机采样,生成 n 个样本点的横坐标
y = np.random.uniform(y_down, y_up, n)
#从一个均匀分布的[y_down,y_up)中随机采样,生成 n 个样本点的纵坐标
#计算点到圆心的距离
dis = np.sqrt((x-a)**2 + (y-b)**2)   #求平方根运算,计算点(x,y)到圆心的距离,返回距
                                     #离数组 dis
#统计落在圆内的点的数目
points_in = sum(np.where(dis <= radius, 1, 0))
#数组 dis 中满足与圆心距离小于或等于 1,返回 1,否则返回 0,累加至 res 中
#计算 pi 的近似值(蒙特卡洛方法的精髓: 用统计值去近似真实值,统计值越大,越逼近真实值)
pi = 4 * points_in / n               #统计落在图中的点的数量,计算圆周率的值
print('pi 的值为: ', pi)
#可视化
fig = plt.figure()
ax = fig.add_subplot(111)            #构建 1×1 的网格图,其中有一个子图
ax.plot(x, y,'ro',color = 'red',markersize = 1)
plt.axis('equal')                    #保持作图时正方形的边长相等,否则会变形
circle = Circle(xy = (a,b), radius = radius, alpha = 0.8)
ax.add_patch(circle)
plt.show()
```

运行结果如图 3-1 所示。

```
pi 的值为: 3.114
```

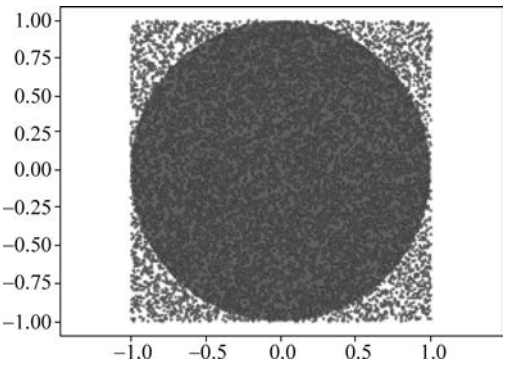

图 3-1　利用蒙特卡洛方法计算圆周率 π 的值

散落点的数量 n 的值不同,统计计算所得 π 的值也不同[①],如表 3-1 所示。

表 3-1　不同散落点的数量,统计得出不同 π 的近似值

n	π 的近似值	n	π 的近似值
1000	3.092	1 000 000	3.143 612
10 000	3.1572	10 000 000	3.141 323 2
100 000	3.137 96		

① 由于点是随机均匀落下,因此当 n 值相同时,π 值的统计计算结果会略有不同。

由表 3-1 可以看出，当 n 的数量越大时，计算的结果越接近 π 的值，即统计的数量越多，得到的结果越接近实际值，这也是大数据的应用思想，当样本数量越大时，其中的规律越明显，越接近真实值。

在程序中，量 n 的值参与运算的地方有多次，为了修改程序和阅读程序的方便，仅在程序开头对 n 赋予了确切的值，在后面的代码中，需要 n 参与运算的，直接用字母 n 代替即可。一般在程序设计中，参与运算的名称称为变量，它可以被赋予不同的值；固定不变的值称为常量。

3.2　变量和常量

3.2.1　变量

变量是程序中值可以发生变化的元素，在使用之前需要给它命名，即关联一个标识符，保障它的唯一性和指代性。变量是在程序中创建的名字，用来表示程序中的“事物”或“参数”。通常来说，变量可以代表一个值，可以是整数、字符串、浮点数等，有意义的变量名会使得代码更具有可读性。

与其他编程语言稍显不同的是，当书写代码 n=1000 时，Python 解释器做了两件事情：第一，在内存中创建了一个值 1000；第二，在内存中创建了一个名为 n 的变量，并把它指向了 1000。变量名和对应的值都包含在该段程序的命名空间(name space)中，如图 3-2 所示。变量名称类似标签，唯一指向了一个值，而不需要事先为其分配内存空间，因此在 Python 的变量使用中，即用即命名，不需要事先指定变量的类型，也可以赋值不同类型的数据，不会发生溢出。

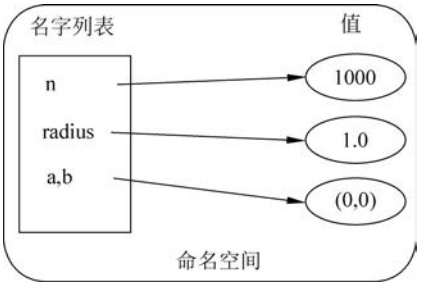

图 3-2　变量和命名空间

Python 中变量的命名规则如下：

(1) 变量标识符对大小写敏感。

(2) 变量名必须以下画线或字母开头，而后面接任意数目的字母、数字或下画线。不能使用空格。以下是合法命名的标识符：its_name_，_its_name _is_xx_。

(3) 变量名称中的字母区分大小写，如 its_name 和 Its_name 是不同的。

(4) 变量的名称要有意义，要能够代表变量的含义，如表示半径的变量名称可以这样起：radiusNum 或 radius_num，这样可以有效提高程序的可读性。

(5) 禁止使用 Python 保留字。Python 的保留字如下。

and	elif	import	raise
as	else	in	return
assert	except	is	try
break	finally	lambda	while
class	for	not	with
continue	from	or	
def	global	pass	
del	if	print	

3.2.2 数据

在计算机中,数据是指所有能输入计算机并被计算机程序处理的符号的介质的总称,是用于输入电子计算机进行处理,具有一定意义的数字、字母、符号和模拟量等的通称。数据(data)是事实或观察的结果,是对客观事物的逻辑归纳,是用于表示客观事物的未经加工的原始素材。

数据是信息的表现形式和载体,可以是符号、文字、数字、语音、图像、视频等。数据和信息是不可分离的,数据是信息的表达,信息是数据的内涵,信息中蕴含知识。数据本身没有意义,数据只有对实体行为产生影响时才成为信息。数据可以是连续的值,如声音、图像,称为模拟数据;也可以是离散的,即不连续,如符号、文字等。

现实世界中的信息要想能够被计算机处理,都需要转化成计算机能够处理的类型,有一个量化到数字化的过程,如声音、图片、文字在被计算机处理之前,都要在保留原有含义的同时,转化成数字的形式,一般称为“嵌入”(embedding)。以文字为例,可以使用 Python 中的 Gesim 库中的 Word2Vec()方法,在最大限度保留语义的前提下,将文字数据嵌入成计算机可以处理的数值形式。

3.2.3 常量

在例 3-1 中,与 n、radius、a 和 b 等变量时常变化所不同的是,在圆周率的推导公式 $\pi=\frac{4x}{y}$ 中,4 是一个不会变化的量,通常把这种程序中不发生变化的元素称为常量。C++中使用 const 保留字指定常量,而 Python 并没有定义常量的保留字。但是 Python 是一门功能强大的语言,所以 Python 中定义常量可以用自定义类的方法来创建。

3.3 运算和表达式

3.3.1 常用的运算

1. 算术运算

在程序设计中,常用运算有算术运算、关系运算、逻辑运算和赋值运算等。算术运算包括数学中常见的运算,如加、减、乘、除、乘方和开方等,它是数学中最古老也最基础的部分。

用算术运算符连接变量而形成的式子称为算术表达式。例 3-1 中，pi = 4 * points_in / n 就是用乘除法符号将常量 4 和变量 points_in、n 连接起来的算术表达式，算术表达式可返回算术计算结果。下面，自定义 pi 和半径的值，通过算术运算来计算圆的周长：

```
>>> pi = 3.14
>>> r = 3
>>> 2 * pi * r
```

执行上述代码后，返回的结果是 18.84。

注意，在使用变量前必须对其赋值，否则编译器会报错。在 Python 中使用变量，预先可以不用定义变量类型，但必须对变量赋值。对变量赋值的过程就是定义类型的过程，且同一个变量，先后可赋予不同类型的值而不会报错。在上例中，如果没有预先对变量 pi 和 r 赋值，则会报错：

```
>>> 2 * pi * r
Traceback (most recent call last):
  File "< pyshell # 0 >", line 1, in < module >
    2 * pi * r
NameError: name 'pi' is not defined
```

或者交换顺序出现同类报错：

```
>>> 2 * r * pi
Traceback (most recent call last):
  File "< pyshell # 0 >", line 1, in < module >
    2 * r * pi
NameError: name 'r' is not defined
```

Python 中的算术运算符如表 3-2 所示。在该表中，假设有变量 x 和 y，预先赋值 x＝10，y＝3。

表 3-2　Python 中的算术运算符

算术运算符	运 算 描 述	示　　例
＋	加法	x＋y 结果为 13
－	减法	x－y 结果为 7
*	乘法	x * y 结果为 30
/	除法	x/y 结果为 3.3333…
**	幂运算	x ** y 的结果是 1000
//	返回商的整数部分	x//y 的结果是 3
%	返回余数部分	x%y 的结果为 1(余数)
＝	赋值运算	x＝10，将 10 赋值给变量 x
＋＝	加法赋值运算	x＋＝10 的运算结果是 x＝20，等效于 x＝x＋10
－＝	减法赋值运算	x－＝10 的运算结果是 x＝0，等效于 x＝x－10
* ＝	乘法赋值运算	x * ＝10 的运算结果是 100，等效于 x＝x * 10
/＝	除法赋值运算	x/＝10 的运算结果是 1.0，等效于 x＝x/10

代码如下：

```
>>> x = 10
>>> y = 3
>>> x/y
3.3333333333333335
>>> x ** y
1000
>>> x//y
3
>>> x % y
1
```

注意，Python 支持不同的数字类型相加，它使用数字类型强制转换的方式来解决数字类型不一致的问题。也就是说，它会将一个操作数转换为和另一个操作数相同的数据类型。但是这种强制转换的操作不是随意进行的，要遵循一定的规则：整数可以转换为浮点数，非复数可以转换为复数。可以理解为是简单类型可以转换为复杂类型，因为浮点数可以表示和整数一样的值(小数点后面的数为 0 就可以了)，复数的虚部为 0 时就可以用来表示非复数。如：

```
>>> x = 3
>>> y = 3.4
>>> x + y
6.4
```

x 的值是整数，y 的值是浮点数，最后在进行求和运算时，将 x 值的 3 转换为浮点数 3.0 参与运算，最后的结果也是浮点数 6.4。

2. 关系运算

传统的关系运算包括集合运算(并、差、交等)，专门的关系运算包括选择、投影、连接、除法、外连接等。有些关系运算需要几个基本运算的组合，要经过若干步骤才能完成。程序设计中关系运算的本质是通过比较两个值来确定它们之间的关系。用关系运算符连接的表达式叫作关系表达式。关系表达式的值只有两种：True(真)和 False(假)。Python 中的关系运算符如表 3-3 所示。表中有两个变量 x 和 y，其中假定 x=10，y=20。

表 3-3 Python 中的关系运算符

关系运算符	描　　述	示　　例
==	如果两个操作数的值相等，则结果为真	(x==y)求值结果为 False
!=	如果两个操作数的值不相等，则结果为真	(x!=y)求值结果为 True
>	如果左操作数的值大于右操作数的值，则结果为真	(x>y)求值结果为 False
<	如果左操作数的值小于右操作数的值，则结果为真	(x<y)求值结果为 True
>=	如果左操作数的值大于或等于右操作数的值，则结果为真	(x>=y)求值结果为 False
<=	如果左操作数的值小于或等于右操作数的值，则结果为真	(x<=y)求值结果为 True

如下面的代码：

```
>>> 3 > 2
True
>>> 3 == 3
True
>>> 3 < 2
False
```

3. 逻辑运算

逻辑运算又称布尔运算。乔治·布尔(George Boole,1815—1864)1815 年 11 月 2 日生于英格兰的林肯郡。他是 19 世纪最重要的数学家之一,出版了《逻辑的数学分析》,这是他对符号逻辑诸多贡献中的第一次。1854 年,他出版了《思维规律的研究》一书,这是他最著名的著作,其中介绍了布尔代数,布尔代数中所进行的二维运算就是布尔运算。用等式表示判断,把推理视为等式的变换。这种变换的有效性不依赖于人们对符号的解释,只依赖于符号的组合规律,这一逻辑理论被人们称为布尔代数[①]。

逻辑运算包括与运算、或运算和非运算三种。逻辑运算的结果只有两种：真和假,常见的逻辑运算符如表 3-4 所示,表中有两个变量 x 和 y,其中 x 的值为 True,y 的值为 False。

表 3-4 Python 中的逻辑运算符

逻辑运算符	描 述	示 例
and	如果两个操作数都为真,则结果为真	(x and y)的结果为 False
or	如果两个操作数中的任何一个为真,则结果为真	(x or y)的结果为 True
not	用于反转操作数的逻辑状态	not(x and y)的结果为 True

程序中的逻辑运算一般用在条件语句或者循环语句中,用于判断执行的条件或循环的次数,这一点在例 3-2 中有所体现。

【例 3-2】 输入学生的姓名,再输入成绩,根据给出的条件判断成绩的等级,最后打印输出：某某的成绩是什么等级。判断标准：小于 60 分,Fail；60～69 分,Pass；70～79 分,Good；80～89 分,Average；90～100 分,Outstanding。

参考代码：

```
name = input("What is your name? ")
score = int(input("What is your score? "))
if score < 60:
    print ("Hello, % s,your grades is 'Fail'" % name)
elif score > = 60 and score < 70:
    print ("Hello, % s,your grades is 'Pass'" % name)
elif score > = 70 and score < 80:
    print ("Hello, % s,your grades is 'Average'" % name)
```

① 引自百度百科中“逻辑运算”词条。

```
elif score >= 80 and score < 90:
    print ("Hello, %s,your grades is 'Good'" % name)
elif score >= 90 and score <= 100:
    print ("Hello, %s,your grades is 'Outstanding'" % name)
else:
    print("Your input is incorrect. ")
```

上面的代码中,if 和 elif 是条件语句,后续章节会有详细介绍,用于解释判断标准的语句,如 score >= 60 and score < 70 是逻辑表达式,该类表达式的结果只有两种:真和假,当表达式的值为真时,程序执行条件语句;当表达式的值为假时,程序不执行该条件语句。

4. 成员运算

成员运算一般用来判断某一元素是否在一个指定的序列对象中,其返回值为布尔值。Python 中的成员运算符如表 3-5 所示。

表 3-5 Python 中的成员运算符

成员运算符	描 述	示 例

in	若某一元素在一个指定的序列中,则返回 True,否则返回 False	>>> x=10
		>>> L=[1,2,3.4]
		>>> x in L
		False

not in	若某一元素不在一个指定的序列中,则返回 True,否则返回 False	>>> x='10'
		>>> s='abcdefg'
		>>> x not in s
		True

【例 3-3】 有一个列表 L=[1,2,3,4,5,6,7,1,2,3,4,5,6,7],要求去掉其中重复的元素。

参考代码:

```
>>> L = [1,2,3,4,5,6,7,1,2,3,4,5,6,7]
>>> L1 = list()
>>> for x in L:
    if x not in L1:
        L1.append(x)
>>> L1
```

输出结果:

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

5. 身份运算

身份运算用来判断两个变量是否指向同一对象,或者某一对象是否在一个序列中。Python 中的身份运算符如表 3-6 所示。

表 3-6 Python 中的身份运算符

身份运算符	描 述	示 例

is	判断两个变量是否指向同一个对象，返回值为 True 和 False	>>> a=1
		>>> b=1
		>>> a is b
		True
		>>> b=2
		>>> a is b
		False
		>>> b=a
		>>> a is b
		True
		>>> id(a)
		502515440
		>>> id(b)
		502515440

is not	判断两个变量是否没有指向同一个变量，返回值为 True 和 False	>>> a=1
		>>> b=2
		>>> a is not b
		True

3.3.2 表达式

表达式是运算符和操作数所构成的序列，是 Python 程序常见的代码。用算术运算符连接起来的表达式称为算术表达式，如 a+b；用关系运算符连接起来的表达式称为关系表达式，如 a<b；用逻辑运算符连接起来的表达式称为逻辑表达式，如 a and b。和数学中的算术运算一样，Python 程序设计中的表达式也有运算顺序。通常，算术运算表达式>关系运算表达式>逻辑运算表达式。

3.4 Python 相关基础语法

3.4.1 空格

Python 语言书写的程序，对格式的要求非常严格，尤其是对齐和缩进。如例 3-2 中，if 语句和 elif 语句后面一行的 print 语句都有缩进，一般是 4 个空格或者按一次 Tab 键的距离。Python 利用空格实现对程序的分段。Python 中明确要求不要混合使用 4 个空格和 Tab 键，只使用二者之一。一段程序中，如果开始使用的是 4 个空格，那么后续代码中都用 4 个空格来控制程序的分段，而不是 Tab 键。不同的平台对 Tab 键展开空格的个数的显示效果并不相同。使用空格的好处是，可以使得各个平台下显示效果完全一致。实际编程过程中，一般使用编辑器进行这种自动转换。例如：按一次 Tab 键自动转换为 4 个空格，以及删除空格时，自动删除一组即 4 个空格(到 Tab 键的合适位置)。

再看下面的代码：

```
>>> print('score', 90 * 0.3 + 81 * 0.7)
score 83.69999999999999
>>> print('score', 90 * 0.3 + 81 * 0.7)  #这样看起来更好一些
score 83.69999999999999
```

上述两段代码的执行结果是一样的，主要区别就在于下面一段代码在算术符号的两边加上了空格，这里的空格是没有语法含义的，一般都是从可读性的角度考虑。一般关键词后以及逗号的后面会加一个空格。这个空格是可选的，加上它可以使得程序更加方便阅读。

3.4.2 注释

在程序设计中，为了提高程序的可阅读性和可理解性，相关代码和程序需要一定的注释。一般来说，源程序的有效注释在20%以上，不宜太多，也不宜过少。通常，函数头部、源文件头部、标志性变量等均需要标注明确的注释，以便于代码的阅读。在例3-1中，便用#标注了多个注释，极大地提高了程序的可读性。在Python中，注释常用的形式有单行注释和多行注释。

(1) 单行注释。单行注释的标识符是#，注释标识符后面的语句或者文字是不会被执行的。如3.4.1节例子中代码print('score', 90 * 0.3 + 81 * 0.7)后面的#，该符号后面的“这样看起来更好一些”这些文字都不会被执行。单行注释有时候也用在程序调试中，若不想让某行代码被执行，就在其之前加入#；若想让它运行，就将该符号去掉。对注释的灵活运用可以使得程序调试更加灵活。

(2) 多行注释。当注释的内容比较长时，可以采用多行注释，标识符是三引号。在例3-4中，代码所完成的任务在注释中说明，由于内容较长，所以用多行注释来体现。

【例3-4】 利用泰勒级数计算e的近似值。

参考代码：

```
'''利用泰勒级数计算 e 的近似值,
当最后一项的绝对值小于 10^ - 5 时认为达到了精度要求,
要求统计总共累加了多少项并输出程序的总运行时间'''
import time
start_time = time.clock()
n = 1
count = 1
term = 1.0
e = 0
while abs(term)> 1e - 5:
    term = term/n
    e = e + term
    n = n + 1
    count += 1
end_time = time.clock()
print("e = %f,运行次数: %d 次" %(e,count))
print("运行耗时" + str(end_time - start_time) + "s")
```

该例中，程序前面的三行文字被包含在三引号中，属于注释文字，程序执行时不会执行注释。

3.5 random 库

random 库是 Python 中用来产生随机数的标准库，使用时需要通过 import random 或 from random import * 语句导入。random 库中主要包含两类函数，常用的有 8 个。random 库常用函数如表 3-7 所示。

表 3-7 random 库常用函数

类 别	函 数	功 能 说 明
基本随机函数	seed(a=None)	改变随机数生成器的种子，在调用其他随机模块之前调用该函数，可固定生成同一个随机数，以便于案例结果复现
	random()	随机生成一个[0.0，1.0]区间的浮点数
扩展随机函数	randint(a,b)	随机生成一个[a,b]区间的整数
	getrandbits(k)	随机生成一个 k 比特(b)长度的非负整数
	randrange(start,stop[,step])	随机生成一个[start,stop)区间，步长为 step 的整数
	uniform(a,b)	随机生成一个[a,b]区间的小数
	choice(seq)	从非空序列 seq 中随机返回一个元素
	shuffle(seq)	随机排列序列中的元素，并返回打乱顺序后的原序列，即不会生成新的序列

【例 3-5】 random 库中常见函数的使用。


观看视频

参考代码：

```
>>> from random import *      #引用 random 库
>>> random()                  #生成 0～1 的随机小数
0.005971912566784976
>>> seed(10)                  #设置随机生成器的种子为 10,然后再生成随机数
>>> random()
0.5714025946899135
>>> seed(10)                  #再次设置随机生成器的种子为 10,即可生成和上次一样的随机数
>>> random()
0.5714025946899135
>>> random()                  #没有设置随机生成器的种子,则生成不同的随机数
0.4288890546751146
>>> for x in range(10):       #随机生成 1～100 的 10 个整数
    print(randint(1,100),end = ' ')
74 2 27 60 63 36 84 21 5 67
>>> getrandbits(3)            #随机生成一个 3b 的整数
3
>>> randrange(1,20,3)         #随机生成一个 1～20,步长为 3 的整数
7
>>> uniform(2,3)              #随机生成一个 2～3 的小数
2.076089346781348
>>> choice([1,2,3,4,5,6,7,8,9])          #随机选中列表序列中的一个元素并返回
6
>>> shuffle([1,2,3,4,5,6,7,8,9])         #随机将列表中的元素乱序
```

```
>>> x = shuffle([1,2,3,4,5,6,7,8,9])
>>> x
>>> print(x)
None
>>> print(shuffle([1,2,3,4,5,6,7,8,9]))  #由于乱序而不会生成新的列表,因此没有任何返回值
None
>>> y = [1,2,3,4,5,6,7,8,9]              #随机打乱原有的列表序列
>>> shuffle(y)
>>> y
[1, 7, 9, 6, 2, 3, 8, 5, 4]
>>>
```


观看视频

【例 3-6】 随机游走(random walk)也称随机漫步。随机行走是指基于过去的表现,无法预测将来的发展步骤和方向。其核心概念是指任何无规则行走者所带的守恒量都各自对应着一个扩散运输定律,接近于布朗运动,是布朗运动理想的数学状态,现阶段主要应用于互联网链接分析及金融股票市场中[①]。随机游走对于股市而言指股价的短期变动不可预测,各种投资咨询服务、收益预测和复杂的图形都毫无用处。现在,随机游走理论已发展成三种形式:强式、半强式和弱式。要求:模拟一维的 1000 步的随机游走,从 0 开始,步长为 1 和－1,且以相等的概率出现。

参考代码:

```
import random
import matplotlib.pyplot as plt
position = 0                    #设置初始位置为 0
walk = []                       #新建空列表,用来存储随机游走的步长及方向
steps = 1000                    #设置步数
for i in range(steps):
    if random.randint(0,1):     #randint()随机生成[0,1]区间的整数,不是 0 就是 1
        step = 1
    else:
        step = -1               #每步向前一步或者向后一步
    position = position + step  #在原有位置 position 的基础上,计算下一步前进或后退之后的位置
walk.append(position)           #将 position 添加至列表 walk 中
#print(walk)
fig = plt.figure()              #生成绘图窗口
ax = fig.add_subplot(111)       #返回一个 ax 对象,参数 111 表示生成的窗口排列为 1 行 1 列(仅一
                                #个窗口),在第 1 个窗口进行绘图
ax.plot(walk)                   #以列表 walk 中的数据绘制图形
plt.show()                      #显示图形
```

运行结果如图 3-3 所示。

【例 3-7】 在例 3-6 中,要求步长为 1 或者－1 的概率是相同的,在代码中,产生步长正负的条件表达式为 random.randint(0,1),该表达式会在[0,1]区间随机生成整数 0 或者 1。要求:测试该表达式随机生成 1 的概率值。

① 引自百度百科中“随机漫步”词条。

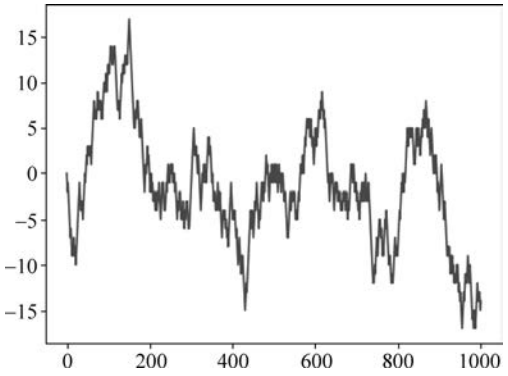

图 3-3 例 3-6 随机游走运行结果①

参考代码：

```
import random
import matplotlib.pyplot as plt
n = 1000                          #共执行 1000 次
total = 0                         #用 total 计算生成值为 1 的次数
for x in range(n):
    total = total + random.randint(0,1)
p = total/n
print(p)
```

运行结果如下：

```
0.493
>>>
```

【分析】 改变测试的数量 n 的值，当 n 为 10 000 时，p 的值为 0.4988；当 n 为 100 时，p 的值为 0.44；当 n 为 10 时，p 的值为 0.4。可以看出，当 n 越大时，表达式 random.randint(0,1)生成 1 的概率越接近 0.5，这和例 3-1 是类似的问题，就像抛硬币会有正反面，抛一次两次看不出，当抛的次数越来越多时，出现正面和反面的概率是相同的，都是 0.5。系统产生随机数也是一样的，表达式 random.randint(0,1)随机产生 0 和 1 的概率是相同的。在随机事件的大量重复出现中，结论往往呈现出某种固定的值。这也是目前大数据应用背后的逻辑：当样本数量足够大时，可以反映其背后的真实规律。

3.6 练习

1. 下列语句中，(　　)在 Python 中是非法的。

   A. x=y=z=1　　B. x=(y=z+1)　　C. x,y=y,x　　D. x+=y

2. 已知 x，y = 1，2，那么执行 x，y = y，x 之后，x 的值为(　　)。

   A. 1　　B. (1，2)　　C. 不符合语法，报错　　D. 2

① 由于随机数每次产生的会有所不同，因此程序每次运行会得到不同的结果。

3. 以下选项中不符合 Python 语言变量命名规则的是(　　)。

A. 3_1　　B. X　　C. ss　　D. InputStr

4. 以下注释语句中,不正确的是(　　)。

A. ＃Python 注释　　B. '''Python 注释'''

C. """Python 注释"""　　D. // Python 注释

5. print(r'\\')和 print('\\')的结果是(　　)。

A. \和\\　　B. \\和\\　　C. \和\　　D. \\和\