第3章

编程逻辑

为了完成复杂的任务，程序需要借助清晰且严谨的逻辑结构来组织操作。本章将介绍程序设计中三种常见的逻辑结构：顺序执行结构、条件分支结构和循环控制结构，以及它们的实现方式。

3.1 程序流程图

为了设计和分析逻辑复杂程序的执行流程，常用流程图来描述程序中关键语句块之间的逻辑关系。程序流程图使用一系列图形、流程线和文字说明来描述程序的基本操作及其流程控制。常见的流程图元素有 6 种，如图 3-1 所示。

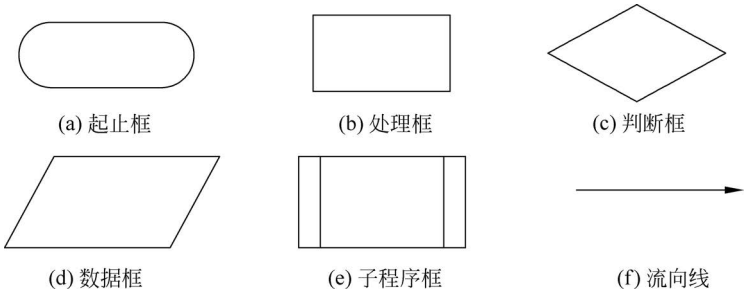


图 3-1 程序流程图基本元素

(1) 起止框：表示程序的开始或结束。

(2) 处理框：用于表示一组处理过程或操作步骤。

(3) 判断框：用于判断条件是否成立，并根据判断结果选择不同的执行路径。

(4) 数据框：表示数据的输入或输出操作。

(5) 子程序框：表示嵌套的子程序，该子程序可以单独绘制其流程图。

(6) 流向线：带有箭头的线条，用于指示程序的执行路径。

流程图的使用能够帮助程序设计者理清程序逻辑，提高代码实现的准确性和可维护性。

3.2 顺序结构

顺序结构是程序中最基本的控制结构，指程序从上到下逐行依次执行代码。顺序结构的语法如下：

```
<语句块 1>
…
<语句块 N>
```

在顺序结构中，变量的作用域自其声明之后生效，直至程序结束或超出其作用范围。

顺序结构流程图如图 3-2 所示。该流程图表示程序从开始到结束按顺序执行<语句块 1>和<语句块 2>。

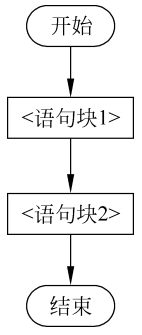


图 3-2　顺序结构流程图

举例如下：

```
age = 8
age = age + 1
print(age)
```

在此示例中，程序按顺序依次执行每行代码，从定义变量 age 到修改变量并输出结果。

3.3　分支结构

分支结构用于根据条件判断执行不同的代码块。程序根据是否满足特定条件选择性地执行某个分支，而跳过其他不满足条件的分支。

3.3.1　if 分支结构

在 Python 中，if 分支结构用于根据不同的条件执行相应的代码块，能够让程序在运行时有选择地执行特定操作。

1. if 分支结构的语法

```
if <条件 1>:      # 若<条件 1>满足，则执行<语句块 1>
    <语句块 1>
…
elif <条件 i>:    # 若不满足以上所有条件，但满足<条件 i>，则执行<语句块 i>
     <语句块 i>
…
else:             # 若以上所有条件均不满足，则执行<语句块 n>
    <语句块 n>
```

其中，<条件 1>、<条件 i>等可以是返回 True 或 False 的表达式或函数。<语句块 1>、<语句块 n >等可以是单行或多行代码。建议确保条件之间无重叠，以保证逻辑清晰。

2. 编程格式说明

(1) 每个分支的条件结束后，应在条件表达式后加上冒号(:)，例如 if <条件 1>:。

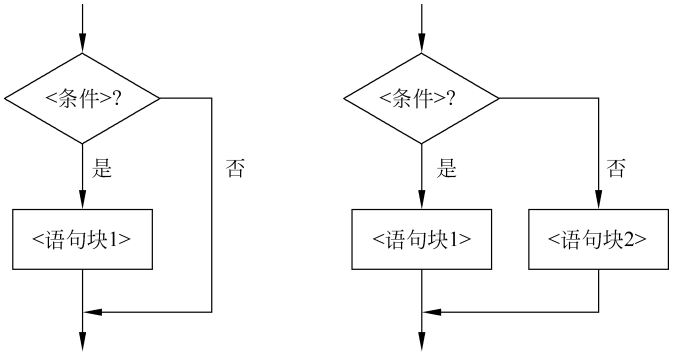
(2) 每个分支的 <语句块> 应缩进 4 个空格。Python 强制使用缩进来标识代码块层级，因此保持一致的缩进非常重要。这种格式在 if 结构和其他结构(如 for 循环和 while 循环)中也通用。

举例如下：

```
if age > 18:
    print("你已成年")
elif age > 12:
    print("你是青少年")
else:
    print("你是儿童")
```

关系运算符常用来构建判断条件，包括 <、<=、>、>=、== 和 !=。值得注意的是，一个等号 = 用于赋值，而两个等号 == 用于比较两个值是否相等。

分支结构可以是单分支、双分支或多分支结构。多个双分支组合可以形成多分支结构。分支结构的流程图如图 3-3 所示。在图 3-3 中，流程图显示了程序根据条件的满足情况选择执行路径，从而实现不同的逻辑分支。



(a) 单分支结构　　(b) 双分支结构

图 3-3　分支结构的流程图

例 3.1　判别中国历史朝代。

在中国历史中，隋、唐、五代十国的年代表如表 3-1 所示，如果给定某一年份，请判断其所处朝代。

表 3-1　中国历史年代表（部分）

朝　　代	起止年代（公元）
隋	581 — 618
唐	618 — 907
五代十国	907 — 960

实现代码主要采用分支结构，程序流程如图 3-4 所示。

判别历史朝代的代码如下：

```
year_str = input("请输入历史年份: ")
year = int(year_str)
dynasty = ''
if year < 581:
    dynasty = '隋之前朝代'
elif year < 618:
    dynasty = '隋朝'
elif year < 907:
    dynasty = '唐朝'
elif year < 960:
    dynasty = '五代十国'
```

```
else:
    dynasty = '五代十国之后朝代'
print('公元{0}年是{1}'.format(year, dynasty))
```

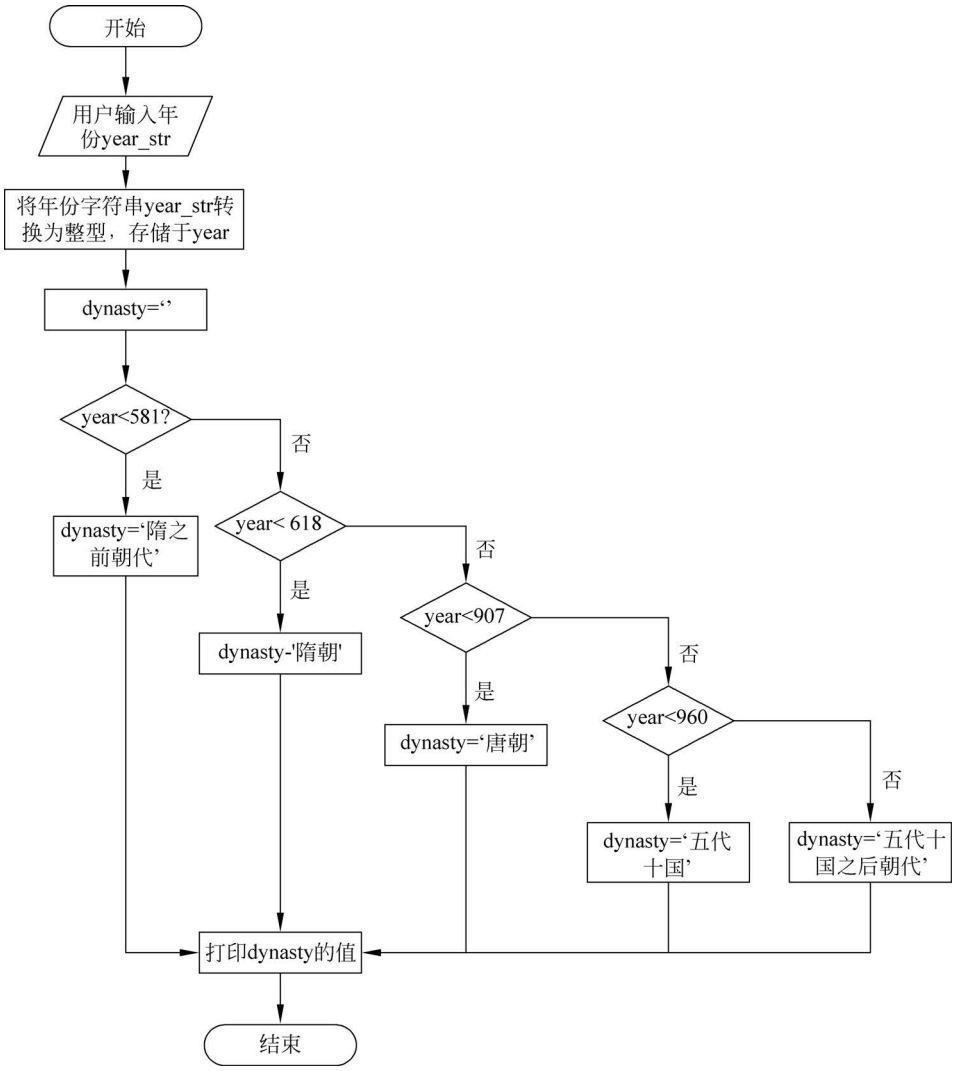


图 3-4　判别历史朝代的流程图

运行结果：

```
请输入历史年份：755
公元 755 年是唐朝
```

3.3.2　模式匹配

模式匹配通过 match 和 case 关键字实现，提供了一种功能更强大的条件分支方式。模式匹配的语法如下：

```
match variable:
    case pattern_1:
```

```
        # 当变量匹配 pattern_1 时执行的代码
    case pattern_2:
        # 当变量匹配 pattern_2 时执行的代码
    case _:
        # 默认情况(类似于 switch-case 的 "default" 分支)
```

其中,match 关键字用于指定一个匹配条件的变量(类似其他编程语言的 switch 语句的作用)。Python 通过检查变量的内容与不同的模式 case 是否匹配来决定执行哪个代码块;每个 case 后面接一个模式(pattern),表示一种条件匹配,可以是简单的值(如数字或字符串),也可以是复杂的结构(如类或嵌套的元组);case_表示默认匹配,在所有其他模式都不匹配时执行。

例 3.2 匹配消息模式。

编写一个根据不同类型的消息结构进行处理的函数。消息的类型可以是文本、图片或视频,每种类型的消息结构不同,包含的内容也不同。希望使用模式匹配来简化代码逻辑,使其能够根据消息的类型,匹配相应的结构并进行处理,实现代码如下:

```
def handle_message(message):
    match message:
        case {"type": "text", "content": text}:
            print(f"Text message: {text}")
        case {"type": "image", "content": image_path}:
            print(f"Image message with path: {image_path}")
        case {"type": "video", "duration": duration}:
            print(f"Video message of duration {duration} seconds")
        case _:
            print("Unknown message type")

# 使用示例
message1 = {"type": "text", "content": "Hello World"}
message2 = {"type": "image", "content": "/path/to/image.jpg"}
message3 = {"type": "video", "duration": 120}
handle_message(message1)
handle_message(message2)
handle_message(message3)
```

上述代码实现了 handle_message()函数,它利用 Python 的模式匹配功能来处理不同类型的消息。通过 match 语句来匹配消息的结构,并根据消息类型进行处理。

第一种情况为文本消息,case {"type": "text", "content": text} 匹配到消息类型为 "text" 的结构,并提取消息的内容 text,然后输出该文本内容。

第二种情况为图像消息,case {"type": "image", "content": image_path} 匹配到消息类型为 "image",提取图像的路径 image_path,然后输出该路径。

第三种情况为视频消息,case {"type": "video", "duration": duration} 匹配到消息类型为 "video",提取视频的时长 duration,然后输出视频时长。

最后一种情况为未知消息类型,“case _”是一个通配符,它处理任何未匹配到的消息类型,输出提示信息 "Unknown message type"。

运行结果:

```
Text message: Hello World
Image message with path: /path/to/image.jpg
Video message of duration 120 seconds
```

3.4　循环结构

循环结构用于在程序中重复执行某段代码，直到满足特定条件后退出。在 Python 中，主要有两种循环实现方式：for 循环和 while 循环。for 循环用于遍历集合，while 循环用于条件控制。循环结构的流程图如图 3-5 所示。

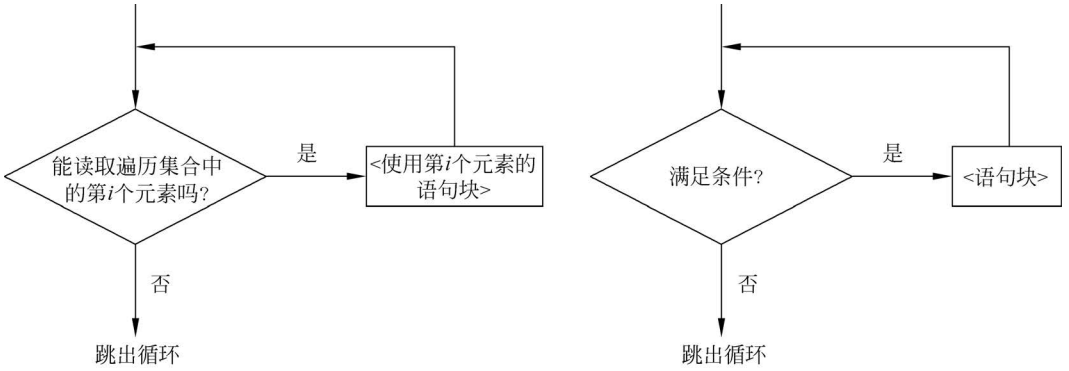


图 3-5　循环结构流程图

3.4.1　for 循环

for 循环是一种遍历循环，用于迭代一个序列中的每一项，并执行相应代码。该序列可以是列表、元组、字符串或生成器等可迭代对象。

for 循环的语法

```
for <循环变量> in <遍历集合>:
    <语句块 1>
else:     # 通常可省略
    <语句块 2>
```

在此结构中，<循环变量> 会依次获取 <遍历集合> 中的每个元素。如果 for 循环正常执行结束，则会执行 else 中的 <语句块 2>，若循环被 break 语句中断，则不会执行 else 块。

举例如下：

```
for i in range(1, 5):
    print(i)
```

运行结果：

```
1
2
3
4
```

在这段代码中，range(1, 5) 是 Python 的内置函数，用于生成从 1 开始、不包括 5 的整数序列 [1, 2, 3, 4]。for 循环会依次将 i 设置为序列中的每个值，并在每次迭代中执行 print(i)。

具体过程如下：

第一次循环，i = 1，输出 1。

第二次循环，i = 2，输出 2。

第三次循环，i = 3，输出 3。

第四次循环，i = 4，输出 4。

这里使用的 range()函数语法如下：

```
range(start, stop[, step])
```

其中，参数 start 表示计数从 start 开始(默认值为 0)，stop 表示计数到该值结束，但不包括该值。step 表示步长，可省略，默认为 1。

3.4.2 while 循环

for 循环在预知循环次数时使用，而 while 循环则用于在不知道具体循环次数时的条件循环。只要满足 while 语句的条件，循环将持续执行，直到条件不成立为止。

while 循环的语法

```
while <条件>:   # <条件>为布尔表达式,结果为 True 或 False
    <语句块 1>
else:           # 本分支和 for 循环的 else 类似,通常可省略
    <语句块 2>
```

当<条件>为 True 时，while 循环会重复执行<语句块 1>；当<条件>变为 False 时，循环终止，程序继续执行 else 块中的<语句块 2>(若存在)。<条件>的更新通常在<语句块 1>中进行，以确保循环能在合适的时机退出。

例 3.3 猜数字游戏。

实现一个简单的猜数字游戏，程序预设一个正确答案，用户通过输入数字进行猜测。每次用户输入数字后，程序会提示猜测的数字是否过大或过小，直到用户猜中正确答案，程序结束，代码如下：

```
 1. number = 62       # 程序中预设一个答案,供用户猜
 2. correct = False
 3. # 当 correct 为 False 时,while 循环继续; 否则,循环退出
 4. while not correct:
 5.     guess = int(input('请输入一个从 0～100 的整数: '))
 6.     if guess == number:
 7.         print('恭喜你,你猜对了!')
 8.         correct = True
 9.     elif guess < number:
10.         print('猜错了,你猜的数太小了')
11.     elif guess > number:
12.         print('猜错了,你猜的数太大了')
13. print('程序结束')
```

在上述代码中，行 1 设置变量 number = 62，这是程序中预设的正确答案。行 2 设置变量 correct 为 False，用于判断用户是否已经猜中答案。行 4 中 while 循环在 correct 为 False 时继续执行。当 not correct 为 True 时，循环继续；当 correct 被设置为 True 后，循环终止。行 5 中 input()函数用于接收用户输入，将输入的字符串通过 int()函数转换为整数，并赋给变量 guess，表示用户的猜测。行 6～8 判断用户输入的 guess 是否等于预设的 number。如果相等，程序输出“恭喜你，你猜对了!”，并将 correct 设置为 True 以终止 while 循环。行 9～10 判断 guess 是否小于 number，如果是，输出提示“猜错了，你猜的数太小了”。行 11～12 判断 guess 是否大于 number，如果是，输出提示“猜错了，你猜的数太大了”。最后，行 13 在用户猜

中答案后，程序跳出 while 循环，输出“程序结束”。

运行结果：

```
请输入一个从 0 到 100 范围内整数: 50
猜错了,你猜的数太小了
请输入一个从 0 到 100 范围内整数: 75
猜错了,你猜的数太大
请输入一个从 0 到 100 范围内整数: 62
恭喜你,你猜对了!
程序结束
```

【二分法思想】

在猜数字游戏中，有一种高效的策略是每次选择当前范围内的中位数作为猜测点（例如，对于范围 [m, n]，猜测 (m+n) / 2），根据提示缩小范围，重复这一过程直到找到正确答案。这正是二分法的应用，通过逐步将区间缩小一半，能够快速锁定目标值。二分法思想广泛应用于计算机算法中，如二分查找、快速排序和求解非线性方程等。该方法的计算复杂度通常为 O(log n)，在大数据量时也能高效地处理问题，大大提高了程序的执行效率。

3.4.3　break 语句

在程序执行时，通常情况下，只有当 for 语句中的遍历条件或 while 语句中的循环条件不满足时，程序才会跳出循环。然而，在某些情况下，可能需要在满足特定条件时立即退出循环。为此，大多数编程语言中都引入 break 语句来实现该功能。

break 语句通常出现在 for 或 while 循环的语句块中，通常与 if 条件语句一起使用。它的作用是强制跳出循环，无论循环条件是否满足，或者迭代次数是否达到预设值，一旦执行到 break 语句，程序将立即跳出当前循环。

break 语句的语法如下所示。

(1) 在 while 循环中的使用。

```
while <条件 1>:
    …
    if <条件 2>:
        break
    …
```

在 while 循环中，当满足<条件 1>时，循环会继续执行。如果在循环体中的 if 语句判断<条件 2>成立，break 语句会强制跳出循环，不再继续执行循环。

(2) 在 for 循环中的使用。

```
for <循环变量> in <遍历集合>:
    …
    if <条件 2>:
        break
    …
```

在 for 循环中，当循环变量遍历<遍历集合>时，循环会按设定的规则进行迭代。如果 if 语句中<条件 2>成立，break 语句将立即跳出循环，无论遍历是否完成。

例 3.4 英文字符转大写。

用户输入任意英文字符串，并将其中的小写字母转换为大写字母。程序应持续运行，直到用户输入 exit 关键字，这时程序终止运行。输入 exit 时，程序不进行大写转换，而是直接退出循环，代码如下：

```
while True:
    str = input("请输入一个英文字符串: ")
    if str == 'exit':
        break
    print("大写: ", str.upper())
print("程序结束")
```

该段程序通过 while True 实现一个无限循环，直到用户输入 exit 时，通过 break 语句强制跳出循环。对于其他输入，程序会将用户输入的小写字母通过字符串方法 upper()转换为大写并输出。

运行结果：

```
请输入一个英文字符串: I have a dream
大写: I HAVE A DREAM
请输入一个英文字符串: exit
程序结束
```

3.4.4 continue 语句

break 语句用于强制跳出整个循环。然而，有时并不需要完全退出循环，而是希望中断当前迭代，跳过本次循环的剩余部分，直接进入下一次迭代。为此，大多数编程语言引入了 continue 语句。

continue 语句的作用是跳过当前迭代中尚未执行的代码，立即进入下一次迭代。它的使用位置与 break 语句类似，通常用于 if 条件语句中。

continue 的语法如下所示。

(1) 在 while 循环中的使用。

```
while <条件 1>:
    …
    if <条件 2>:
        continue
    …
```

在 while 循环中，如果在某次迭代时满足<条件 2>，continue 语句会跳过剩余的代码块，直接进入下一次循环，并重新判断<条件 1>是否成立以决定是否继续执行。

(2) 在 for 循环中的使用。

```
for <循环变量> in <遍历集合>:
    …
    if <条件 2>:
        continue
    …
```

在 for 循环中，continue 语句会跳过当前迭代的剩余代码，并立即开始下一次循环，从而使循环变量获取下一个值进行下一次迭代。

例 3.5　英文字符转大写并过滤非英文输入。

编写一个程序允许用户持续输入英文字符串，并将其中小写字母转换为大写输出。如果用户输入非英文字符，程序提示错误信息并跳过该输入。当用户输入 exit 时，程序终止运行。

```
1.  while True:
2.      str = input("请输入一个英文字符串: ")
3.      if str == 'exit':
4.          break
5.      if not str.isascii():
6.          print("输入非英文字符串")
7.          continue
8.      print("大写: ", str.upper())
9.  print("程序结束")
```

在代码段行 5 中，使用字符串方法 isascii()判断输入的字符串是否为纯英文字符。如果字符串中的所有字符属于 ASCII 字符集，该方法返回 True，否则返回 False。如果输入非英文字符，程序执行行 6 和 7，continue 语句跳过本次循环的剩余部分(即不执行行 8)，直接开始下一次循环，返回行 1，等待新的输入。

运行结果：

```
请输入一个英文字符串: 我有一个梦想
输入非英文字符串
```

【循环编程的经验】

避免死循环：在循环结构中，应确保程序能够在有限次循环后跳出循环，防止进入死循环。如果循环条件一直为真，程序将无法正常结束。

循环嵌套规则：在嵌套循环中，内循环必须完整地嵌套在外循环内部。内循环的终止条件不能影响外循环的正常执行。

3.4.5　海象操作符

海象操作符(Walrus Operator)是一种新的赋值表达式，表示为：=。它允许在表达式中进行赋值并返回赋值结果，从而使代码更加简洁和高效。海象操作符的主要用途是在条件表达式或循环中同时进行变量赋值和条件检测，减少代码量并提高可读性。

(1) 在 while 循环中使用海象操作符：海象操作符可以在 while 循环中简化代码逻辑，避免重复的赋值操作，代码如下：

```
while (n := get_value()) > 0:
    process(n)
```

上述代码中，海象操作符：=同时完成了 n 的赋值和条件判断，使代码简洁高效。如果不使用海象操作符，代码如下：

```
n = get_value()
while n > 0:
    process(n)
    n = get_value()
```

在不使用海象操作符的情况下，需要先进行赋值操作，再单独判断循环条件，这会导致代

码重复。

(2) 在 if 语句中使用海象操作符：海象操作符还可以在 if 语句中同时进行赋值和条件判断，简化代码结构。例如：

```
if (result := calculate_value()) > 10:
    print(f"Result is large: {result}")
```

在上述代码中，result 被赋值的同时，其值也被用于 if 语句的条件判断。如果不使用海象操作符，代码如下：

```
result = calculate_value()
if result > 10:
    print(f"Result is large: {result}")
```

使用海象操作符可以减少代码行数，使代码更加简洁，提高代码的可读性和维护性。

3.5 应用案例

3.5.1 斐波那契数列

数学家莱昂纳多·斐波那契在研究兔子繁殖问题时，提出了著名的斐波那契数列(Fibonacci sequence)，也被称为“兔子数列”。斐波那契数列的生成规则如下：

```
F(0) = 0
F(1) = 1
```

对于 $n \geqslant 2$，斐波那契数列满足：$F(n)=F(n-1)+F(n-2)$，其中 n 为自然数($n \in N$)。

随着数列项数的增加，前一项与后一项的比值逐渐趋近于黄金分割数值 0.618，因此该数列也被称为“黄金分割数列”。

例 3.6 生成斐波那契数列。

编写程序生成 100 以内的斐波那契数列，并验证其与黄金分割现象的关系。

```
1.  x1, x2 = 0, 1
2.  max_number = 100
3.  while x2 < max_number:
4.      ratio = round(x1/x2, 5) if x2 != 0 else 0     # 避免除以 0 的情况
5.      print(x2, ratio)
6.      x1, x2 = x2, x1 + x2
```

这里，行 1 初始化斐波那契数列的前两项，x1 代表第一个数，x2 代表第二个数。行 2 设置最大数 max_number，即程序将生成不超过 100 的斐波那契数列。在行 3 中，当当前数列项 x2 小于 max_number 时，继续循环生成斐波那契数列；否则，跳出循环。行 4 计算前一项与当前项之比 ratio，取小数点后 5 位；为了避免除以 0 的情况，当 x2 为 0 时，ratio 设为 0。行 5 打印当前斐波那契数列项 x2 和前一项与当前项之比 ratio。行 6 更新数列中的项，将当前项 x2 赋值为下一次循环的第一项 x1，而下一项 x2 则为上一项两项之和。

输出结果：

```
1 0.0
1 1.0
2 0.5
```

```
3 0.66667
5 0.6
8 0.625
13 0.61538
21 0.61905
34 0.61765
55 0.61818
89 0.61798
```

输出结果的第一列组成了斐波那契数列，第二列为前一项与当前项的比值。可观察到，随着斐波那契数列的增加，相邻两项的比值逐渐接近 0.618，即黄金分割数。

【现实中的斐波那契数列】

科学家们发现，在许多植物的花瓣、萼片、果实的数量以及排列方式中，存在着一种神奇的规律，这种规律与斐波那契数列高度吻合。例如，向日葵的花盘中有两组螺旋线：一组以顺时针方向盘绕，另一组以逆时针方向盘绕，且两组螺旋线相互嵌套，如图 3-6 所示。

图 3-6　向日葵的两类曲线

在图中，逆时针螺线有 13 条(绿色线条表示)，蓝色的顺时针螺线有 21 条，13 和 21 正是斐波那契数列中相邻的两项。虽然不同品种的向日葵螺旋线的数量有所不同，但大多数情况下，这些螺旋线的数量都是斐波那契数列中的两个连续数字，如 13 和 21、34 和 55、55 和 89 或 89 和 144。类似的现象也存在于其他植物中，如罗马花椰菜、松果、菠萝、树枝等。通过这种特殊的排列方式，植物能够更有效地利用阳光和空气，从而有助于它们的生长和繁衍。

3.5.2　计算圆周率

圆周率的计算方法很多，其中有一种基于蒙特卡洛(Monte Carlo)思想的计算方法。该方法一般认为美国在第二次世界大战中研制原子弹的“曼哈顿计划”中首先提出。但实际上，1777 年，法国数学家布丰就提出用投针实验的方法求圆周率 π。

例 3.7　利用蒙特卡洛方法计算圆周率 π 的值。

使用蒙特卡洛方法求解 π 的基本步骤如下：在图 3-7 所示，边长为 1 的单位正方形中占据半径为 1 的单位圆的右上方，向单位正方形范围内，随机投掷 n 个飞镖后，统计落在了右上

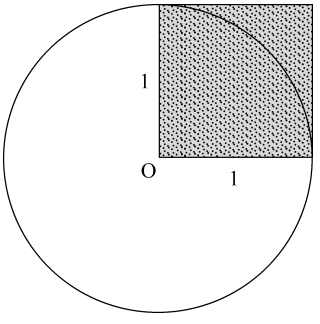


图 3-7　蒙特卡洛方法

方的1/4圆内的飞镖数量为m。根据统计模拟方法，1/4圆的面积(即π/4)与单位正方形面积(即1)之比，约等于落在1/4圆内飞镖数量m与落在单位正方形内的飞镖数n之比，即$\pi/4=m/n$，也即$\pi=4m/n$。

代码如下：

```
1. from random import random
2. from math import sqrt
3. n = 1000000
4. count_in_circle = 0
5. for i in range(0, n):
6.     x, y = random(), random()
7.     dist = sqrt(x**2 + y**2)
8.     if dist <= 1:
9.         count_in_circle += 1
10. pi = 4 * count_in_circle / n
11. print("Pi's value is", pi)
```

在上述代码中，行1导入生成随机数的random()函数；行2导入math库的sqrt()函数，用于计算平方根；行3设置$n=1000000$，表示模拟投掷飞镖的次数；行4初始化count_in_circle=0，用于计数落在圆内的飞镖数量。

行5开始循环，运行n次，模拟n次投掷飞镖；第6行为每次投掷生成随机坐标(x,y)，其中x和y是在[0,1)范围内的随机数。行7计算该坐标点到圆心的距离，并将结果存入dist变量。行8～行9中，如果dist小于或等于1，说明该点落在半径为1的圆内，将count_in_circle加1。

循环结束后，在行10使用公式pi=4 * count_in_circle/n近似计算圆周率π的值，最后在行11输出计算结果。

程序运行三次，得到如下三行输出结果。由于每次生成的随机数不同，导致结果略有差异。增加n的值可以提高结果的精确度，使计算的π值更接近真实值。

```
Pi's value is 3.14238
Pi's value is 3.142416
Pi's value is 3.141184
```

3.5.3 广播模型、扩散模型和传染模型

1. 广播模型

假设一个相关人群，人数为N，这里相关人群是指感知某一信息或感染某一疾病的潜在人群。在某一时刻t，人群可被划分为两类：一类是知情者或感染者(Infective)，感染者数量用I_t表示，另一类是还未感染的潜在人群，称为易感者(Susceptible)，易感者数量用S_t表示。这里$N=I_t+S_t$。

广播模型公式为：$I_{t+1}=I_t+P_{\text{broad}}S_t$

其中I_t和I_{t+1}分别表示第t和$t+1$时刻的感染者人数，P_{broad}为感染概率，S_t为第t时刻易感者人数。

广播模型刻画了新闻、信息和技术，并通过电视、广播、互联网等进行传播，是一种“一对多”的传播方式。

例 3.8　模拟广播传播：全村人听到广播所需的天数。

一个有 100 位居民的小村庄，村主任每天通过广播播放一次修路倡议。假设每位居民在任意一天听到广播的概率为 10%（即 P_broad=0.1）。问：需要多少天，才能确保全村人都听到这个倡议。

```
Number = 100
day = 0
infective = 0.0
infective_number = 0
probability = 0.1
infective_sequence_str = ''
while (infective_number < Number):
    day += 1
    susceptible = Number - infective
    infective = infective + probability * susceptible
    infective_number = int(infective + 0.5)
    infective_sequence_str = infective_sequence_str + "{} ".format(infective_number)
print(infective_sequence_str)
print("In the {}th day, everyone receives the message from the broadcast.".format(day))
```

在上述代码中，行 1～6 初始化各变量，其中 infective 用小数表示已听到广播的居民数量，infective_number 用整数表示用于输出的感染人数，其他变量用于控制传播过程和结果输出。行 7～12 使用 while 循环，当接收广播的人数小于全村人数时，继续循环：行 8 更新天数 day；行 9 计算尚未听到广播的居民数量 susceptible；行 10 根据广播模型，计算当天接收广播的居民数 infective，累积上一天的接收人数，行 11 使用 int()函数将浮点型 infective 四舍五入转换为整数，确保感染人数为整数（为了实现四舍五入，浮点数 infective 加上 0.5 后再取整）；行 12 将当天接收广播的人数拼接到存储结果的字符串 infective_sequence_str 中。在行 13～14 中，当接收广播的人数等于全体人数时，循环终止：行 13 输出每天的接收广播人数序列；行 14 输出第几天时，全村人都接收到了广播信息。

输出结果：

```
10 19 27 34 41 47 52 57 61 65 69 72 75 77 79 81 83 85 86 88 89 90 91 92 93 94 94 95 95 96 96 97 97 97
97 98 98 98 98 99 99 99 99 99 99 99 99 99 99 99 100
In the 51th day, everyone receives the message from the broadcast.
```

程序输出了每天接收广播的居民人数。通过模拟广播传播，最终在第 51 天，全村所有居民都接收到了广播信息。

2. 扩散模型

广播模型考虑了从一个原点向外部多个对象广播信息或传染病毒，在实际中，还存在着另一种信息传播的方式——扩散模型，即获得信息的多个感染者，可将信息或病毒传递给它们周围的对象，形成多对多的扩散传播方式。

扩散模型公式为：$I_{i+1}=I_t+P_{\text{diffuse}}\cdot\frac{I_t}{N}\cdot S_t$

其中 $P_{\text{diffuse}}=P_{\text{spread}}\cdot P_{\text{contact}}$，将扩散概率 P_{diffuse} 定义为传播概率 P_{spread} 与接触概率 P_{contact} 的乘积。

例 3.9　模拟信息扩散：全村人知晓倡议所需的天数。

一个有 100 位居民的小村庄，村主任将修路倡议扩散给他周围的居民。该倡议在人们之

间的扩散概率为10%(即P_diffuse=0.1)。要求编写一个程序,模拟这种信息扩散的过程,计算多少天后全村的居民都知道这个倡议。

```
Number = 100
day = 1
infective = 1
infective_number = 1
probability_diffuse = 0.1
infective_sequence_str = str(infective_number) + ''
while (infective_number < Number):
    day += 1
    susceptible = Number - infective
    infective = infective + probability_diffuse * (infective/Number) * susceptible
    infective_number = int(infective + 0.5)
    infective_sequence_str = infective_sequence_str + "{} ".format(infective_number)
print(infective_sequence_str)
print("In the {}th day, everyone knows the message from the message diffusion.".format(day))
```

运行结果:

```
1 1 1 1 1 2 2 2 2 2 3 3 3 3 4 4 4 5 5 6 6 7 8 8 9 10 11 12 13 14 15 16 18 19 21 22 24 26 28 30 32 34 37 39
41 44 46 49 51 54 56 59 61 63 66 68 70 72 74 76 78 80 81 83 84 86 87 88 89 90 91 92 92 93 94 94 95 95
96 96 97 97 97 97 98 98 98 98 99 99 99 99 99 99 99 99 99 99 99 100
In the 100th day, everyone knows the message from the message diffusion.
```

输出每天知晓消息的居民人数,输出结果显示在第100天时,全村的100位居民都知晓了这个倡议。

在实际情况下,大多数信息同时通过广播和扩散两种方式传播。这种组合模型被称为巴斯模型。它通过广播概率和扩散概率来决定信息传播的权重,巴斯模型公式如下:

$$I_{i+1}=P_{\text{broad}}\cdot S_t+P_{\text{diffuse}}\cdot\frac{I_t}{N}\cdot S_t$$

其中,P_{broad} 为广播概率,P_{diffuse} 为扩散概率。

3. SIR模型

广播模型和扩散模型均假设人一旦获得该消息,则永远不会放弃它。但对传染病情况不完全适用,当一个人感染了传染病后,经过若干天后可能会痊愈。因此,SIR模型是研究传染病传播的经典模型,包括三类人群。

易感者(Susceptible):尚未感染传染病,但可能被感染的人。

传染者(Infective):已经感染并能传播疾病的人。

痊愈者(Recover):曾感染但已痊愈,理论上不再传播疾病,但此例假设痊愈者与易感者具有相同的感染概率。

SIR模型的传染病传播公式为:

$$I_{t+1}=I_t+P_{\text{contact}}\cdot P_{\text{spread}}\cdot\frac{I_t}{N}\cdot S_t-P_{\text{recover}}\cdot I_t$$

其中,P_{contact}、P_{spread}、P_{recover} 分别为传染病的接触概率、传播概率和痊愈概率。

例3.10 SIR模型模拟:传染病传播与痊愈模拟。

在一个有100位居民的小村庄,村主任感染了某种传染病。接触概率、传播概率和痊愈概

率分别为 P_contact、P_spread 和 P_recover。假设痊愈者在痊愈后不会对该传染病产生免疫力，具有与易感者相同的感染概率。模拟 50 天内该村庄的感染人数变化情况。

```
1. Number = 100              # 村庄总人数
2. days = 50                 # 模拟天数
3. infective = 10            # 初始感染人数
4. infective_number = 10     # 整数形式的初始感染人数
5. contact_spread_recover_str = input("Please input the probabilities of contact, spread and
   recover in a comma-delimited format:")
6. prob_contact_str, prob_spread_str, prob_recover_str = contact_spread_recover_str.split(',')
7. prob_contact = float(prob_contact_str)
8. prob_spread = float(prob_spread_str)
9. prob_recover = float(prob_recover_str)
10. infective_sequence_str = str(infective_number) + ''  # 用于存储感染人数变化的字符串
11. for day in range(2, days + 1):
12.     susceptible = Number - infective  # 计算易感人数
13.     infective = infective + prob_contact * prob_spread * (infective / Number) * susceptible -
        prob_recover * infective
14.     infective_number = int(infective + 0.5)  # 四舍五入取整
15.     infective_sequence_str += "{} ".format(infective_number)
16. print(infective_sequence_str)  # 输出感染人数序列
```

在上述代码中，行 1～4 初始化变量；行 5 读入用户输入的接触概率、传播概率和痊愈概率字符串；行 6 调用字符串的 split()函数，将用户输入的字符串以逗号分隔，分别赋值给 prob_contact_str、prob_spread_str 和 prob_recover_str；行 7～9 将三个概率字符串分别转换为浮点数，赋给相应的这三个概率；行 10 将感染人数赋给感染序列字符串；行 11～15 模拟传染病传播过程，从第 2 天到第 50 天迭代循环：计算易感人数(行 12)，使用 SIR 模型公式，计算每天新增的感染人数和痊愈人数(行 13)，将浮点型转化为整数型的感染人数(行 14)，然后将感染人数追加到感染人数序列字符串中。

运行第一次，输入 0.5、0.6、0.2 之后，输出结果：

```
Please input the probabilities of contact, spread and recover in a comma-delimited format:0.5,0.6,0.2
10 11 11 12 13 14 15 15 16 17 18 19 20 20 21 22 23 23 24 25 25 26 27 27 28 28 29 29 29 30 30 30 31 31
31 31 31 32 32 32 32 32 32 32 32 33 33 33 33 33
```

运行第二次，输入 0.5、0.6、0.4 之后，输出结果：

```
Please input the probabilities of contact, spread and recover in a comma-delimited format:0.5,0.6,0.4
10 9 8 7 6 5 5 4 4 3 3 3 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

对比两次运行结果，可以发现：第一次输入 $P_{\text{contact}}=0.5$、$P_{\text{spread}}=0.6$ 和 $P_{\text{recover}}=0.2$ 时，感染人数逐渐增加并趋于稳定，最终有 33 个人处于感染状态，达到平衡；第二次输入 $P_{\text{contact}}=0.5$、$P_{\text{spread}}=0.6$ 和 $P_{\text{recover}}=0.4$ 时，感染人数递减直至 0，最终疾病消失。

两次结果的差异可以通过基本再生数 R_0 来解释，基本再生数 R_0 定义为：

$$R_0=\frac{P_{\text{spread}}\cdot P_{\text{contact}}}{P_{\text{recover}}}$$

第一次运行：$R_0=0.5*0.6/0.2=1.5$。当 R_0 大于 1 时，传染病会在整个人群中传播。因此，感染人数逐渐增加，并最终稳定在 33 人左右。

第二次运行：$R_0=0.5*0.6/0.4=0.75$。当 R_0 小于 1 时，传染病趋于消失，感染人数逐

渐减少，最终降至 0。实验结果验证了以上结论。

本章小结

编程逻辑如表 3-2 所示。

表 3-2　编程逻辑

类　别	原理/方法/属性	说　明
程序流程图	使用流程图描述程序执行流程	起止框、处理框、判断框等图形用于展示程序的逻辑结构，帮助理清程序逻辑
顺序结构	从上到下依次执行代码	程序最基本的结构，代码按顺序执行
分支结构	根据条件执行不同代码块	if-elif-else 结构控制程序执行路径，分为单分支控制程序、双分支控制程序和多分支控制程序
模式匹配	使用 match-case 结构	简化多条件分支逻辑
循环结构	重复执行代码直到条件满足	包括 for 和 while 循环，分别适用于遍历和条件控制
for 循环	遍历集合	适用于已知循环次数，依次遍历集合元素
while 循环	条件控制循环	条件满足时循环执行代码块，适用于不确定循环次数
break 语句	强制退出循环	在满足特定条件时退出循环，通常与 if 搭配使用
continue 语句	跳过当前迭代	跳过本次循环剩余代码，直接进入下一次迭代
海象操作符	使用：＝进行赋值并返回值	提高代码简洁性，可在表达式中同时赋值和条件判断
应用案例	实际问题应用编程逻辑	包括斐波那契数列、圆周率计算、信息扩散模型等，展示逻辑结构的实际应用