第3章 程序控制结构

本章将介绍程序控制结构这个重要的主题。程序控制结构是编程中用于控制程序执行流程的基本机制，它能帮助我们按照特定的逻辑进行代码的执行和处理。在本章中，我们将学习三种主要的程序控制结构：顺序结构、选择结构和循环结构。


视频讲解

3.1 顺序结构

顺序结构是编程中最简单也是最基础的程序控制结构之一。顺序结构指的是按照代码的书写顺序，一行一行地执行代码，没有任何条件判断或循环控制。在顺序结构中，代码将按照从上到下的顺序被逐行执行。每一行代码的执行结果都会影响到下一行代码的执行。

让我们来看一个简单的例子。

【例 3-1】 顺序结构示例。

```
#顺序结构示例
print("欢迎来到顺序结构的世界!")
print("这是第一行代码。")
print("这是第二行代码。")
print("这是第三行代码。")
print("顺序结构的代码将按照顺序执行。")
print("这是最后一行代码。")
```

程序运行结果如下：

```
欢迎来到顺序结构的世界!
这是第一行代码。
这是第二行代码。
这是第三行代码。
顺序结构的代码将按照顺序执行.
这是最后一行代码。
```

在例 3-1 中，代码按照从上到下的顺序一行一行地执行。首先，会打印出“欢迎来到顺序结构的世界!”，然后依次打印出每一行代码的内容，直到打印出“这是最后一行代码。”后结束。

顺序结构非常直观和易于理解，因为代码的执行顺序简单明了。在顺序结构中，每一行代码都是按照其在代码中出现的先后顺序被执行的，不会跳过也不会重复执行任何代码。顺序结构也是构建更复杂程序的基础。通过顺序结构，我们可以组合多个简单的操作，以便实现更复杂的功能和任务。在编写代码时，确保代码的顺序与预期的执行顺序一致非常重要。一旦代码的顺序出错，可能会导致程序逻辑错误和运行异常。顺序结构是编程中最基本的结构之一。

3.2　选择结构

选择结构是程序控制结构中的一种重要形式，它允许根据不同的条件选择性地执行不同的代码块。通过选择结构，我们可以根据程序中的特定条件来决定程序的执行路径，从而实现不同情况下的不同逻辑处理。

选择结构主要有三种形式：单分支结构、双分支结构和多分支结构。下面让我们来一一了解它们吧。

3.2.1　单分支结构

单分支结构是选择结构中最简单的形式，它只包含一个条件判断语句，即 if 语句。if 语句根据条件的真假来决定是否执行特定的代码块。

在单分支结构中，if 语句后面的条件表达式会被求值，如果条件表达式的值为真(True)则执行 if 代码块中的语句；如果条件表达式的值为假(False)则跳过 if 代码块，继续执行后续的代码。

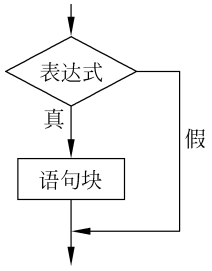

图 3-1　单分支结构的流程

单分支结构的流程如图 3-1 所示。

让我们来看一个简单的例子。

【例 3-2】 单分支结构示例。

```
#单分支结构示例
temperature = 25
if temperature > 30:
    print("天气炎热,请注意防晒!")
print("今天的温度:" + str(temperature) + "℃")
```

程序运行结果如下：

```
今天的温度: 25℃
```

在上面的例子中，首先定义了一个变量 temperature，其值为 25。然后使用 if 语句判断 temperature 是否大于 30。由于 temperature 的值为 25，不满足条件，因此不执行 if 代码块中的语句。接着，程序继续执行后续的代码，打印出今天的温度。

在实际编程中，单分支结构经常用于根据条件执行不同的操作。我们可以根据具体情况在 if 代码块中编写适当的处理逻辑，根据用户输入的条件执行不同的操作，根据条件判断来输出不同的结果等。

例如我们可以把代码进行修改，将第一句改为 temperature = 35，则程序运行结果如下：

```
天气炎热,请注意防晒!
```

3.2.2　双分支结构

双分支结构是选择结构中扩展了一种情况处理的形式，它除了包含一个条件判断语

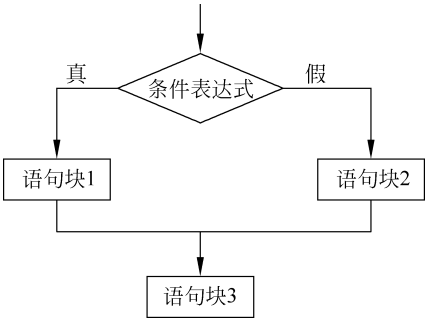

图 3-2 双分支结构的流程

句(if 语句),还包含一个 else 语句。通过双分支结构,程序可以根据条件的真假执行不同的代码块。

在双分支结构中,if 语句后面的条件表达式会被求值,如果条件表达式的值为真(True)则执行 if 代码块中的语句;如果条件表达式的值为假(False),则执行 else 代码块中的语句。

双分支结构的流程如图 3-2 所示。

让我们来看一个简单的例子。

【例 3-3】 双分支结构示例。

```
#双分支结构示例
score = 80
if score >= 60:
    print("成绩合格,恭喜你通过了考试!")
else:
    print("成绩不合格,请多加努力!")
print("你的分数是:" + str(score))
```

程序运行结果如下:

```
成绩合格,恭喜你通过了考试!
你的分数是: 80
```

在上面的例子中,首先定义了一个变量 score,其值为 80。然后使用 if 语句判断 score 是否大于或等于 60。由于 score 的值为 80,符合 if 条件,因此执行 if 代码块中的语句,打印出成绩合格的消息。接着程序跳过 else 代码块,继续执行后续的代码,打印出你的分数是 80。

双分支结构提供了一种根据条件真假执行不同代码块的方式。通过合理使用 if 和 else 语句,我们可以根据不同的条件结果进行相应的处理,使程序具备更灵活的逻辑。

如果将语句 score = 80 修改为 score = 50,由于 score 的值为 50,不符合 if 条件,因此不执行 if 代码块中的语句,接着程序执行 else 代码块,打印出成绩不合格的消息,然后继续执行后续的代码。

3.2.3 多分支结构

多分支结构是选择结构中最灵活的形式,它是通过使用多个条件判断语句(if-elif-else)来执行不同的代码块的。在多分支结构中,程序会逐个检查条件,并执行与第一个满足条件相关联的代码块。

在多分支结构中,if 语句后面的条件表达式会被求值,如果条件表达式的值为真(True),则执行 if 代码块中的语句;如果条件表达式的值为假(False),则继续检查下一个 elif 语句的条件,如果条件为真,则执行对应的代码块;如果所有的条件都为假,则执行 else 代码块中的语句。

多分支结构的流程如图 3-3 所示。

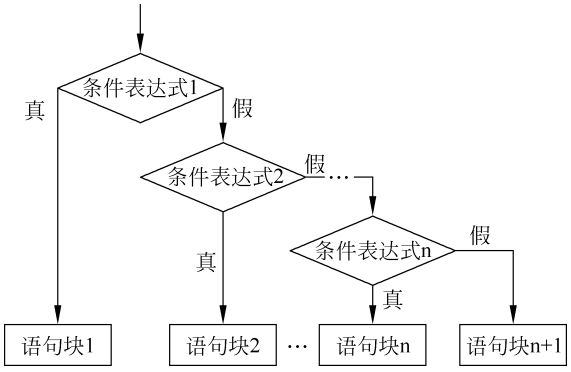

图 3-3　多分支结构的流程

【例 3-4】 多分支结构示例。

```
#多分支结构示例
grade = 85
if grade >= 90:
    print("优秀")
elif grade >= 80:
    print("良好")
elif grade >= 70:
    print("一般")
elif grade >= 60:
    print("及格")
else:
    print("不及格")
print("你的成绩是:" + str(grade))
```

程序运行结果如下：

```
良好
你的成绩是 85
```

在上面的例子中，首先定义了一个变量 grade，其值为 85。使用多个 elif 语句进行条件判断，根据不同的分数范围打印出对应的成绩等级。由于 grade 的值为 85，满足 grade >= 80 的条件，因此执行对应的代码块，打印出良好。接着，程序跳过其他的 elif 代码块，继续执行后续的代码，打印出你的成绩是 85。

相对于单分支和双分支结构，多分支结构允许我们根据多个条件的不同结果执行相应的代码块，从而实现更加细致的逻辑处理。

3.2.4　选择结构的嵌套

选择结构的嵌套是指在一个选择结构中包含另一个选择结构，通过嵌套的方式实现更为复杂的条件判断和代码执行。

在选择结构的嵌套中，内层的选择结构会根据外层选择结构的判断结果来执行相应的代码块。嵌套结构可以嵌套多层，每一层都可以根据需求进行条件判断和代码执行。

选择结构的嵌套的流程如图 3-4 所示。

【例 3-5】 判断一个数字是否为偶数，并输出相应的结果。

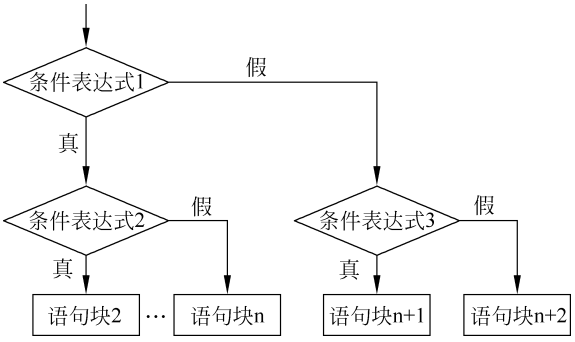

图 3-4　选择结构的嵌套的流程

```
#选择结构的嵌套示例 1
num = 6
if num % 2 == 0:
  if num == 0:
    print("数字为 0")
  else:
    print("数字为偶数")
else:
  print("数字为奇数")
```

程序运行结果如下：

```
数字为偶数
```

在上面的例子中，首先定义了一个变量 num，其值为 6。外层的 if 语句判断 num 是否为偶数，如果是偶数，则进入内层的 if 语句判断 num 是否为 0，如果是 0，则打印出数字为 0；如果不是 0，则打印出数字为偶数。如果外层的 if 语句判断 num 不是偶数，则执行 else 代码块，打印出数字为奇数。

【例 3-6】 根据用户输入的年龄判断是否能参加一个活动。

```
#选择结构的嵌套示例 2
age = int(input("请输入您的年龄:"))
if age >= 18:
  if age <= 60:
    print("您可以参加活动")
  else:
    print("对不起,活动仅限于 18 - 60 岁年龄段")
else:
  print("对不起,您未达到参加活动的年龄要求")
```

在上面的例子中，使用内置函数 input()获取用户输入的年龄，并将其转换为整型。外层的 if 语句判断年龄是否大于或等于 18，如果是，则进入内层的 if 语句判断年龄是否小于或等于 60，如果是，则打印出可以参加活动的消息；如果不是，则打印出活动仅限于 18～60 岁年龄段的消息。如果外层的 if 语句判断年龄小于 18，则执行 else 代码块，打印出未达到参加活动的年龄要求的消息。

选择结构的嵌套提供了更大的灵活性，可以根据需要嵌套多层选择结构来处理复杂的条件判断和代码执行。

3.3　循环结构

视频讲解

循环结构是编程中一种重要的控制结构，它允许我们多次执行相同或相似的代码块，以便重复执行相同的任务或处理逐个元素的集合。循环结构可以根据指定的条件来确定是否继续执行循环体内的代码，从而实现重复执行的功能。

在大多数编程语言中，常见的循环结构主要有两种：while 循环和 for 循环，接下来我们将分别介绍这两种循环结构。

3.3.1　while 循环

while 循环结构是根据条件来判断是否需要继续重复执行语句块，故可以称为条件循环。无论循环的次数是否预知，都可以使用 while 语句来循环执行语句块。while 语句包括以下两种形式。

1. 基本 while 语句

while 循环由一个循环条件表达式和一个循环体组成，格式如下：

```
while 条件表达式:
    语句块
```

while 语句的执行过程如下：在每一次循环开始之前先测试循环条件，如果条件为真，则执行循环体内的代码，然后再次执行循环条件的测试，直到条件为假时停止循环。

循环条件表达式通常使用布尔表达式来判断是否满足循环继续执行的条件。当循环条件为真时，循环体内的代码会被执行，然后再次执行循环条件的测试，直到循环条件为假时退出循环。

while 循环的流程如图 3-5 所示。

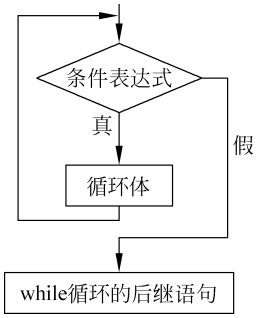

图 3-5　while 循环的流程

【例 3-7】　使用 while 循环计算 1 到 10 的累加和。

```
#while 循环示例 1
sum = 0
i = 1
while i <= 10:
    sum += i
    i += 1
print("1 到 10 的累加和为:", sum)
```

程序运行结果如下：

```
1 到 10 的累加和为:55
```

在上面的例子中，首先定义了两个变量 sum 和 i，分别用来保存累加和及循环变量的值。while 循环的条件是 i 小于或等于 10，当循环条件为真时，循环体内的代码会被执行。

循环体内的代码是将 i 的值加到 sum 上，sum ＋＝ i 是一个复合赋值运算符，它的作用是将变量 i 的值累加到变量 sum 上，并将结果重新赋值给 sum，它等价于 sum＝sum ＋ i。当 i 的值等于 11 时，循环条件为假，退出循环。最后，打印出 1 到 10 的累加和。

【例 3-8】 从 1 开始打印到用户输入的数。

```
#while 循环示例 2:从 1 开始打印到用户输入的数
n = int(input("请输入一个整数:"))
i = 1
while i <= n:
    print(i)
    i += 1
```

上述示例程序中，我们使用了 while 循环来从 1 开始打印到用户输入的数。首先使用 input()函数获取用户输入的数，并使用 int()函数将其转换为整数类型并赋值给变量 n。然后定义了一个循环变量 i 的初始值为 1。循环条件是 i <= n，当循环条件为真时，执行循环体内的代码。循环体内的代码是打印 i 的值，并将 i 的值加 1。循环条件为假时，退出循环。

从上面两个例子我们可以看出，通过 while 循环，我们可以重复执行特定的代码块，直到达到指定的终止条件为止，这样可以在各种情况下灵活地处理重复执行的逻辑。

2. 扩展 while 语句

在 Python 中，while 循环后面也可以跟随一个 else 子句。这个 else 子句将在 while 循环中由于循环条件不再满足而正常结束时执行。但是如果 while 循环是通过 break 语句提前退出的，那么 else 子句将不会被执行。关于 break 语句我们将在 3.3.3 节进行介绍。

格式：

```
while 条件表达式:
      语句块 1
else:
      语句块 2
```

这个特性在需要知道循环是正常结束还是被中断的情况下特别有用。例如，你可能想要执行一些清理操作或记录循环正常完成的情况。下面是一个简单的例子，它使用 while 循环和 else 子句来打印一系列数字，并在循环正常结束时打印一条消息。

【例 3-9】 打印数字。

```
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("循环正常结束,没有使用 break 语句。")
```

程序运行结果如下：

```
0
1
2
3
4
循环正常结束,没有使用 break 语句。
```

在这个例子中，count 从 0 开始，并在每次迭代中增加 1。当 count 达到 5 时，while 循环的条件 count < 5 将不再为真，循环将正常结束。因此，else 子句中的消息会被打印出来。

3.3.2　for 循环

for 循环是一种计数循环，通过指定循环的次数来执行循环体内的代码。通常在以下情况使用 for 循环。

(1) 循环次数已知；

(2) 需要遍历处理 Python 的序列结构或可迭代对象中的每个元素。

for 语句的格式如下：

```
for 循环变量 in 遍历结构:
        语句块 1
[else:
        语句块 2]
```

for 循环的执行过程如下：从遍历结构中逐一提取元素，放入循环变量，循环次数就是元素的个数，每次循环中的循环变量值就是遍历结构中提取的当前元素值。

可选的 else 部分执行方式和 while 语句类似。如果全部元素被遍历后，结束执行循环体，则执行 else 后的语句块 2；若因在语句块 1 中执行了 break 语句而结束循环时，不会执行 else 后的语句块 2。

注意：

(1) for 循环的循环次数等于序列结构或可迭代对象的元素个数。

(2) 若需要按指定次数循环，可以使用 range()函数产生的 range 对象来配合控制循环。

【例 3-10】 计算 1 到 10 的累加和。

```
#for 循环示例 1:计算 1 到 10 的累加和
sum = 0
for i in range(1, 11):
    sum += i
print("1 到 10 的累加和为:", sum)
```

程序运行结果如下：

```
1 到 10 的累加和为:55
```

在前面的例子中，我们采用了 while 循环来计算 1 到 10 的累加和。在本例中，我们使用了 for 循环来计算 1 到 10 的累加和。range(1,11)表示一个整数序列，从 1 到 10(注意，终止值虽然是 11，但不包含)。每次循环迭代时，控制变量 i 会依次取到序列中的值，然后将 i 的值加到 sum 上。循环结束后，打印出 1 到 10 的累加和。

【例 3-11】 遍历列表元素并打印。

```
#for 循环示例 2:遍历列表元素并打印
fruits = ["apple", "banana", "orange", "grape"]
for fruit in fruits:
    print(fruit)
```

程序运行结果如下：

```
apple
banana
orange
grape
```

上述示例程序中，我们使用了 for 循环来遍历列表中的元素并逐个打印出来。定义了一个名为 fruits 的列表，其中包含了多个水果名称。使用 fruit 作为循环变量，for fruit in fruits 表示对列表 fruits 进行迭代。每次循环迭代时，循环变量 fruit 会依次取到列表中的值，然后将其打印出来。循环结束后，完成对列表的遍历。

通过 for 循环，我们可以指定循环的次数或遍历集合中的元素，实现对特定代码块的重复执行或对元素的逐个处理。它是一种常用的循环结构，使得处理重复性任务变得更加简洁和高效。

在 Python 中，for 循环后面同样可以跟随一个 else 子句。这个 else 子句在 for 循环正常遍历完其迭代对象的所有元素后执行。如果循环因为任何原因（例如 break 语句）被提前终止，else 子句将不会被执行。

这个特性在某些情况下非常有用，比如需要在循环正常结束后执行一些操作，但又不想在循环体内部重复这些代码时。下面是一个简单的例子，展示如何在 for 循环后面使用 else 子句。

【例 3-12】 遍历列表元素并打印，如果遇到特定的元素（数字 5），则提前退出循环，否则在循环结束后打印一条消息“数字 5 不在列表中”。

```
#假设我们有一个列表,想要遍历它并打印出每个元素
#如果遍历过程中没有遇到特定的元素(比如数字 5),则在循环结束后打印一条消息
numbers = [1, 2, 3, 4, 6, 7, 8, 9]
for num in numbers:
    print(num)
    if num == 5:
      break   #如果找到数字 5,则提前退出循环
else:
    #如果循环正常结束(即没有遇到 break 语句),则执行 else 子句
    print("数字 5 不在列表中")

print("循环结束,继续执行后续代码…")
```

在这个例子中，我们有一个包含数字的列表 numbers。我们遍历这个列表，并检查是否包含数字 5。如果找到数字 5，我们将使用 break 语句退出循环。由于 break 语句导致循环提前结束，因此 else 子句不会被执行。如果列表中没有数字 5，循环将正常结束，此时 else 子句会被执行，并打印出“数字 5 不在列表中”的消息。最后，无论是否执行了 else 子句，都会打印“循环结束，继续执行后续代码…”，表示程序继续执行 for 循环后面的代码。

3.3.3 循环控制语句

当我们编写循环时，有时候需要在特定条件下改变循环的行为，例如提前终止循环或跳过当前迭代。这就是循环控制语句的作用所在。

循环控制语句可以让我们在循环过程中根据条件的满足与否来动态地改变循环的流

程。它们是 break 语句和 continue 语句，可以在循环体内部使用。

break 语句是用于提前终止循环的控制语句。当某个条件满足时，我们可以使用 break 语句立即结束整个循环，不再执行循环体内剩余的代码，直接跳出循环。这在我们需要在特定条件下停止循环执行的情况下非常有用。

【例 3-13】 判断一个数是否为质数。

```
#判断一个数是否为质数
n = int(input('请输入一个整数:'))
for i in range(2, n):
  if n % i == 0:
    print('{}不是质数'.format(n))
    break
else:
  print('{}是质数'.format(n))
  print("循环结束")
```

程序运行结果如下：

```
请输入一个整数:11
11 是质数
```

在上述示例程序中，我们使用了 break 语句来提前结束循环。循环通过 range(2,n)控制。当 n 能够被循环变量 i 整除，表明 n 不是一个质数，执行 break 语句，此时循环立即终止，不再执行剩余的循环体内代码。当循环遍历结束，即 n 不能被所有变量 i 整除，表明 n 是质数，执行 else 语句，输出 n 是质数。

continue 语句是用于跳过当前迭代并继续下一次迭代的控制语句。当某个条件满足时，我们可以使用 continue 语句跳过当前迭代的剩余代码，直接进入下一次迭代(与 break 语句不同的是 continue 语句不会直接跳出循环)。这在我们需要在特定条件下跳过一些迭代的情况下非常有用。

【例 3-14】 使用 continue 跳过当前迭代。

```
#使用 continue 跳过当前迭代
for i in range(1, 6):
  if i % 2 == 0:
    continue
  print(i)
print("循环结束")
```

程序运行结果如下：

```
1
3
5
循环结束
```

在上述示例程序中，我们使用了 continue 语句来跳过当前迭代。循环通过 range(1,6)控制，从 1 到 5 遍历。当循环变量 i 为偶数时，执行 continue 语句，此时将跳过该次迭代剩下的代码，并进行下一次迭代。因此，在循环中，只有奇数会被打印出来。最后打印出“循环结束”。

通过使用 break 和 continue 语句，我们可以根据循环内部的条件来灵活控制循环的行

为。这种灵活性使得我们能够根据具体的需求改变循环的流程，增加条件判断和逻辑处理，从而实现更加精确和高效的循环控制。无论是提前终止循环还是跳过当前迭代，循环控制语句为我们提供了强大的工具来控制循环的执行。

3.3.4 循环的嵌套

循环的嵌套也是一种循环结构，它允许在一个循环内部包含另一个循环。通过嵌套循环，我们可以在外层循环的每次迭代中执行内层循环，从而实现更加复杂的循环逻辑和嵌套迭代操作。

嵌套循环的工作原理是：外层循环的每次迭代都会触发内层循环的完整执行。在内层循环执行完毕后，外层循环再继续进行下一次迭代，直到外层循环结束。这种层层嵌套的循环结构可以实现多维度的数据遍历、复杂的条件判断和逻辑处理。

【例 3-15】 使用嵌套循环输出九九乘法表。

```
#使用嵌套循环输出九九乘法表
for i in range(1, 10):
  for j in range(1, i+1):
    product = i * j
    print(f"{i} * {j} = {product}", end="\t")
  print()
```

程序运行结果如下：

```
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7  7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8  8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9  9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
```

在上述示例程序中，我们使用了嵌套 for 循环来输出九九乘法表。外层循环控制被乘数 i，从 1 到 9 循环。内层循环控制乘数 j，从 1 到 i 循环。每次内层循环迭代时，计算出乘积 product，然后用 print 语句输出乘法表的一项。外层循环继续迭代，直到 i 达到最大值 9，九九乘法表输出完成。

【例 3-16】 使用 while 循环寻找 1～100 的所有质数(3_16_findPrime.py)。

```
#使用 while 循环寻找 1～100 的所有质数
print("1～100 的所有质数为:")
#循环遍历 1～100 的所有数
i = 2                    #从 2 开始,因为 1 不是质数
while i <= 100:
  j = 2
  is_prime = True        #用来判断 i 是否为质数,初始值为 True
  while j <= i // 2:     #从 2 开始一直除到 i/2,判断是否有因子
    if i % j == 0:
```

```
        is_prime = False          # 如果 i 除以 j 没有余数,说明有因子,不是质数
        break
    j += 1

    if is_prime:                  # 如果 i 是质数,将其进行输出
      print(i, end = " ")
```

程序运行结果如下：

```
1～100 的所有质数为:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

在上述例子中,我们使用了两个嵌套的 while 循环,外层循环遍历 1～100 的所有数,内层循环判断当前数是否为质数。在具体实现过程中,内层循环从 2 开始一直除到当前数的一半,如果在这个范围内发现了当前数的因子,即可以整除当前数,那么这个数就不是质数,将 is_prime 设为 False,并且直接跳出循环。最后,如果 is_prime 为 True,则将其进行打印输出,最终就得到了 1～100 的所有质数。

下面是一个使用外层 for 循环和内层 while 循环打印数字三角形的示例。

【例 3-17】 使用外层 for 循环和内层 while 循环打印数字三角形(3_17_printTriangle.py)。

```
# 使用外层 for 循环和内层 while 循环打印数字三角形
height = 5
for i in range(1, height + 1):        # 外层 for 循环控制行数
  num = 1
  j = 1
  while j <= i:                       # 内层 while 循环控制每行数字的个数
    print(num, end = " ")
    num += 1
    j += 1
  print()
```

程序运行结果如下：

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

在上述例子中,我们使用了外层 for 循环和内层 while 循环来打印数字三角形。外层 for 循环控制打印的行数,从 1 到指定的 height 值。内层 while 循环则控制每行打印的数字个数,初始值为 1,通过不断递增 num 来打印不同的数字。每次内层 while 循环迭代时,使用 print 语句输出一个数字,并在末尾添加一个空格,形成数字三角形的一行。外层 for 循环迭代一次后,会进行换行继续打印下一行。经过嵌套循环的迭代,我们可以得到一个数字三角形。

以上我们仅是列举了三个非常简单的循环嵌套的例子,在实际中,嵌套循环的应用非常广泛。例如,在图形绘制中,可以使用嵌套循环逐行逐列地绘制矩形、三角形等复杂形状;在数据分析和处理中,可以使用嵌套循环遍历二维列表或多维数组,进行元素查找、计算等操作;在算法设计中,可以使用嵌套循环实现复杂的搜索、排序等算法。

需要注意的是,在使用嵌套循环时,需要合理设计循环变量和循环条件,确保循环能够

正确执行并避免死循环。此外，嵌套循环的性能也需要考虑，尽量避免多层级的深度嵌套循环，以提高代码的效率和可读性。

总之，循环的嵌套是一种强大的编程技巧，它允许我们利用层层嵌套的循环结构来处理更加复杂和多样化的问题。通过合理设计和运用嵌套循环，我们可以实现更灵活、高效的循环控制和迭代操作，从而提升程序的功能和性能。

➢ 生活中的循环

大道至简，人生除了努力没有捷径。凡事要做好、做成、取得成功，其实没有什么轻松的捷径，都是需要坚持不懈地重复(循环)实践、长时间深耕。

《弟子规》中也告诉我们，“不力行，但学文，长浮华，成何人”，也是说凡事要力行，反复实践、练习。

“一万小时定律”(要成为某个领域的专家，需要大约 10000 小时的练习)也是强调了持续不断练习和努力的重要性，也就是专注于一件事上的努力循环。


视频讲解

3.4 循环实践

3.4.1 随机验证码的生成

1. 任务描述

在登录某些系统的时候，经常会需要我们输入一个随机生成的 4 位验证码，验证码一般较多是数字验证码，也有数字和字母混合的验证码。

下面我们就用循环结构来实现一个 4 位混合验证码的生成。

2. 任务分析

验证码最大的特点是随机性，所以 4 位验证码的生成最基本的问题是产生一位随机数或字母，然后把这个随机的过程重复 4 次，即可产生 4 位随机验证码。对于每次产生的是随机数字还是随机字母，我们也可以随机决定。

我们先来解决产生一位随机码的问题。

(1) 首先我们先随机产生 1 个数 0 或者 1。

(2) 如果是 0，随机生成 1 位 0～9 的数字。

(3) 如果是 1，随机生成 1 位 A～Z 的字母。

接下来，我们将产生 1 位随机码的过程，重复循环 4 次，即可得到 4 位随机验证码。

3. 任务实施

具体步骤如下。

(1) 导入所需库，使用 random 库来生成随机数。

(2) 初始化一个空字符串来存储生成的 4 位验证码。

(3) 生成 0-1 随机数。

(4) 如果随机数为 0，生成随机数字，使用 random.randint(0,9)生成一个 0 到 9 的随机

数字，并添加到字符串末尾。

(5) 如果随机数为1，生成随机字母，并添加到字符串末尾。

(6) 通过使用for循环结构，将以上过程循环4次。

(7) 使用print()函数打印这个字符串作为验证码。

生成4位随机验证码的代码(3_18_randomCode.py)如下：

```
#使用循环结构生成一个4位随机验证码
#引入random库来生成随机数
import random

#初始化一个空字符串来存储数字
code = ''

#循环4次，每次生成一个0～9的数字 或 A～Z的字母
for i in range(0,4):
    x = random.randint(0,1)
    if x == 0:              #产生随机数字
        code += str(random.randint(0,9))
    else:                   #产生随机字母
        code += chr(random.randint(65,90))
#打印生成的验证码
print("生成的验证码是:" + code)
```

某次运行结果如下：

```
生成的验证码是: A8P6
```

利用Python编程中的顺序结构和循环结构，结合random库中的random.randint()函数，我们设计并实现了一个生成4位随机验证码的方法。该方法确保了验证码的随机性，同时满足了4位的要求。通过巧妙地运用库函数，我们简化了随机数生成的过程，使得整个验证码生成过程既高效又可靠。这种方法不仅展示了Python编程的灵活性，也体现了编程中结构化思维的重要性。

3.4.2　百鸡百钱

1. 任务描述

百鸡百钱是我国古代数学家张丘建在《算经》一书中提出的数学问题："鸡翁一值钱五，鸡母一值钱三，鸡雏三值钱一。百钱买百鸡，问鸡翁、鸡母、鸡雏各几何？"

即公鸡5文钱一只，母鸡3文钱一只，小鸡3只一文钱，用100文钱买100只鸡，问公鸡、母鸡、小鸡要买多少只刚好凑足100文钱。

2. 任务分析

这个问题是一个经典的数学问题，我们的任务是找出公鸡、母鸡和小鸡各需要买多少只才能刚好凑足100文钱。要解决这个问题，我们需要设置三个变量：x表示公鸡的数量，y表示母鸡的数量，z表示小鸡的数量。然后，我们可以根据题目条件建立以下方程组：

(1) x+y+z=100(总数量)；

(2) 5x+3y+z/3=100(总价格)。

由于小鸡是3只一文钱,所以在第(2)个方程中,小鸡的价格是z/3。

3. 任务实施

接下来,我们将通过编程来找出满足这两个方程的x、y和z的值。

(1) 初始化变量:我们需要遍历所有可能的公鸡和母鸡的数量组合,然后计算小鸡的数量。

(2) 遍历所有可能的组合:由于公鸡和母鸡的数量都不能为负数,我们可以使用两个嵌套的循环来遍历所有可能的公鸡和母鸡的数量组合。

(3) 检查是否满足条件:对于每一对公鸡和母鸡的数量,我们可以计算出小鸡的数量,并检查是否满足两个条件:总数量为100只和总价格为100文钱。

(4) 输出结果:如果找到了满足条件的组合,就输出这些值。

具体代码(3_19_chickenMoney.py)如下:

```
#遍历所有可能的公鸡和母鸡的数量组合
for x in range(0, 21):                                  #公鸡最多可以买20只(5*20=100)
    for y in range(0, 34):                              #母鸡最多可以买33只(3*33=99)
        z = 100 - x - y                                 #根据总数量计算小鸡的数量
        if z % 3 == 0 and 5*x + 3*y + z/3 == 100:   #检查是否满足条件
            #直接打印满足条件的鸡的数量组合
            print(f"满足条件的鸡的数量为:公鸡{x}只,母鸡{y}只,小鸡{z}只。")
            break                                       #如果找到了满足条件的组合,就跳出循环
```

运行结果如下:

```
满足条件的鸡的数量为:公鸡0只,母鸡25只,小鸡75只。
满足条件的鸡的数量为:公鸡4只,母鸡18只,小鸡78只。
满足条件的鸡的数量为:公鸡8只,母鸡11只,小鸡81只。
满足条件的鸡的数量为:公鸡12只,母鸡4只,小鸡84只。
```

我们通过两个嵌套的for循环遍历了所有可能的公鸡和母鸡的数量组合,并计算了对应的小鸡数量。一旦找到满足总数量和总价格条件的组合,就立即打印输出,从而成功解决了百鸡百钱的问题。这种方法体现了编程在解决数学问题时的灵活性和高效性。通过这个问题,我们可以看到编程如何帮助我们快速找到满足多个条件的解决方案,从而拓宽了解决此类问题的思路和方法。

巩固训练

1. 输入一个百分制的成绩,要求根据不同分数输出成绩等级A、B、C、D、E。90分以上为A,80～89分为B,70～79分为C,60～69分为D,60分以下为E。

2. 接收用户输入的整数,判断该数是否为素数(素数就是质数,即除了1和它本身以外不能被其他的数整除的数)。

3. 输出所有三位水仙花数。

4. 有一分数数列:2/1,3/2,5/3,8/5,13/8,21/13,…求出这个数列的前20项之和。提

示：抓住分子与分母的变化规律。

5. 产生 5 个[0-1]的随机数，计算 5 个实数的平均值，并且按要求完成如下操作：

(1) 以实数形式将平均值输出到屏幕，要求保留 5 位小数。

(2) 以百分比形式将平均值输出到屏幕，要求保留 3 位小数。

提示：需用到 random 模块的 uniform()函数。

6. 产生 100 个[0,500]的随机整数，输出这 100 个数中所有的素数，要求每行输出 8 个整数，每个整数占 5 列，右对齐(for 循环)。

提示：需用到 random 模块的 randint()函数，产生的数放在 list 列表中。

7. 某大型超市为了促销，采用购物打折优惠方法，每位顾客一次购物：

① 在 500 元以上者，按九五折优惠；

② 在 1000 元以上者，按九折优惠；

③ 在 1500 元以上者，按八五折优惠；

④ 在 2000 元以上者，按八折优惠。

编写程序，计算所购商品优惠后的价格。

8. 猜数字游戏，随机产生一个范围内的数字，请用户猜测答案。要求：

(1) 数字的上下限由用户输入。

(2) 每次运行程序，答案可以是随机的。我们需要引入外援：random 模块，randint()函数，返回一个随机的整数。

(3) 每运行一次程序应该提供给用户多次猜测机会，程序需要重复运行某些代码(循环结构)。

(4) 猜错的时候程序应该给点提示，告诉用户输入的值是大了还是小了(比较操作符，分支结构)。

(5) 统计用户猜测的次数。

运行效果参考如下：

```
请输入猜测的数字最小值和最大值,用逗号隔开:
10,50
请输入你的猜测:
30
偏大了,请继续猜:20
偏小了,请继续猜:26
偏大了,请继续猜:25
偏大了,请继续猜:23
偏大了,请继续猜:21
恭喜你猜对了!你一共猜了 6 次
```