第3章 控制语句

学习目标

- 熟练掌握分支语句、循环语句。
- 掌握 break 语句和 continue 语句。
- 能针对具体实例编写控制程序，并合理设计程序的测试数据。能预判循环的执行次数。

分支结构又称为选择结构。本章首先介绍基于条件表达式的 if 语句分支结构中单分支语句、双分支语句、多分支语句、嵌套分支语句和分支结构的三元运算。接着介绍基于模式匹配的 match/case 分支结构。然后介绍两种循环控制语句及两种循环中断语句。最后给出一个应用实例。

3.1 基于条件表达式的 if 语句分支结构

Python 中采用 if 语句的分支结构根据条件表达式的判断结果为真(包括非零、非空)还是为假(包括零、空)，选择运行程序的其中一个分支。Python 的分支结构控制语句主要有单分支语句、双分支语句、多分支语句、嵌套分支语句和分支结构的三元运算。

视频讲解

3.1.1 单分支 if 语句

单分支 if 语句由四部分组成，分别为关键字 if、条件表达式、冒号、表达式结果为真(包括非零、非空)时要执行的语句体。其语法形式如下：

```
if 条件表达式:
    语句体
```

单分支 if 语句的执行流程如图 3.1 所示。

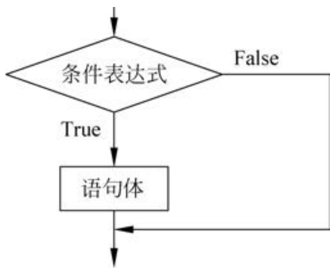

图 3.1 单分支 if 语句的执行流程

单分支 if 语句先判断条件表达式的值是真还是假。如果判断的结果为真(包括非零、非空)，则执行语句体中的操作；如果条件表达式的值为假(包括零、空)，则不执行语句体中的操作。语句体既可以包含多条语句，也可以只由一条语句组成。当语句体由多条语句组成时，要有统一的缩进形式，否则可能会出现逻辑错误或导致语法错误。

【例 3.1】 从键盘输入圆的半径，如果半径大于或等

于0,则计算并输出圆的面积和周长。

程序源代码如下：

```
#example3_1.py
#coding = gbk
import math
r = eval(input("请输入圆的半径:"))

if r >= 0:
    d = 2 * math.pi * r
    s = math.pi * r ** 2
    print('圆的周长 = ',d,'圆的面积 = ',s)
```

程序测试：运行程序 example3_1.py,首先输入一个大于或等于0的半径,如5,观察程序的运行结果。再次运行程序,输入一个小于0的半径,如−1,观察程序的运行结果。

只有在输入的半径为大于或等于0的数时,会产生正确的输入和输出。如果输入的半径小于0,则不产生任何输出。

程序 example3_1.py 的运行结果如下：

```
请输入圆的半径:5
圆的周长 =  31.4159265359 圆的面积 =  78.5398163397
```

思考：如果程序编写如下,会产生怎样的结果？

```
#question3_1.py
#coding = gbk
import math
r = eval(input("请输入圆的半径:"))

if r >= 0:
    d = 2 * math.pi * r
    s = math.pi * r ** 2
print('圆的周长 = ',d,'圆的面积 = ',s)
```

程序测试：运行程序 question3_1.py,首先输入一个大于或等于0的半径,如5,观察程序的运行结果。再次运行程序,输入一个小于0的半径,如−1,观察程序的运行结果。观察程序 example3_1.py 和 question3_1.py 运行结果的异同。并请思考：对于单分支结构的程序,如何设计测试数据以验证程序流程上没有错误？

3.1.2 双分支 if/else 语句

视频讲解

双分支 if/else 语句的语法形式如下：

```
if 条件表达式:
    语句体 1
else:
    语句体 2
```

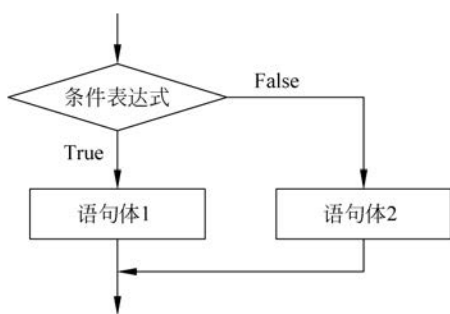

图 3.2 双分支 if/else 语句流程图

双分支 if/else 语句的执行流程如图 3.2 所示。

if/else 语句是一种双分支结构。先判断条件表达式值的真假,如果条件表达式的结果为真(包括非零、非空),则执行语句体 1 中的操作;如果条件表达式为假(包括零、空),则执行语句体 2 中的操作。语句体 1 和语句体 2 既可以包含多条语句,也可以只由一条语句组成。

【例 3.2】 从键盘输入表示年份的数字赋值给变量 t,如果年份 t 能被 400 整除,或者能被 4 整除但不能被 100 整除,则输出"t 年是闰年",否则输出"t 年不是闰年"(t 用输入的年份代替)。

程序源代码如下:

```
#example3_2.py
#coding = gbk
import math
t = int(input("请输入年份:"))

if t % 400 == 0 or (t % 4 == 0 and t % 100!= 0):
    print(t,'年是闰年',sep = "")
else:
    print(t,'年不是闰年',sep = "")
```

程序测试:运行程序,首先输入年份 1996,观察程序的运行结果。再次运行程序,输入年份 2000,观察程序的运行结果。再次运行程序,输入年份 2003,观察程序的运行结果。

程序 example3_2.py 第一次运行结果如下:

```
请输入年份:1996
1996 年是闰年
```

程序 example3_2.py 第二次运行结果如下:

```
请输入年份:2000
2000 年是闰年
```

程序 example3_2.py 第三次运行结果如下:

```
请输入年份:2003
2003 年不是闰年
```

思考一下,如果只输入一个年份值进行测试的话,能否说明程序流程无误?请总结,在用复杂的条件表达式进行判断时,应如何设计测试数据才可以验证程序流程是正确的。

【例 3.3】 某金融企业正在招聘新员工,凡是满足以下两个条件之一的求职者将会收到面试通知。

(1) 25 岁及以下且是重点大学"金融工程"专业的应届学生。

(2) 具备至少 3 年工作经验的"投资银行"专业人士。

编写程序判断一个24岁非重点大学“投资银行”专业毕业，已有3年工作经验的求职者能否收到面试通知。

分析：该企业的面试条件涉及年龄、工作年限、毕业院校类别、所学专业四方面。为此，设定以下变量：年龄age（整型，取值应该大于0），工作时间（年）jobtime（整型，取值应该大于或等于0），毕业院校类别college（字符串类型，取值为“重点”“非重点”），所学专业major（字符串类型，取值为“金融工程”“投资银行”“其他”）。条件（1）和条件（2）各自内部的逻辑关系都是“并且”。条件（1）和条件（2）之间的逻辑关系是“或”。

条件（1）的表达式如下：

```
age <= 25 and college == "重点" and major == "金融工程" and jobtime == 0
```

条件（2）的表达式如下：

```
major == "投资银行" and jobtime >= 3
```

程序源代码如下：

```
#example3_3.py
#coding = gbk
age = 24
jobtime = 3
college = "非重点"
major = "投资银行"

if (age <= 25 and college == "重点" and major == "金融工程" and jobtime == 0) \
   or (major == "投资银行" and jobtime >= 3):
    print("欢迎您来参加面试!")
else:
    print("抱歉,不符合我们的面试条件.")
```

程序example3_3.py的运行结果如下：

```
欢迎您来参加面试!
```

请思考，以上程序代码给定的年龄、工作年限、毕业院校类别、所学专业四方面应满足哪些条件可得到以上运行结果？

3.1.3 多分支if/elif/else语句

视频讲解

多分支if/elif/else语句的语法形式如下：

```
if 条件表达式 1 :
    语句体 1
elif 条件表达式 2 :
    语句体 2
…
elif 条件表达式 n-1 :
    语句体 n-1
else:
    语句体 n
```

多分支语句的执行流程如图3.3所示。

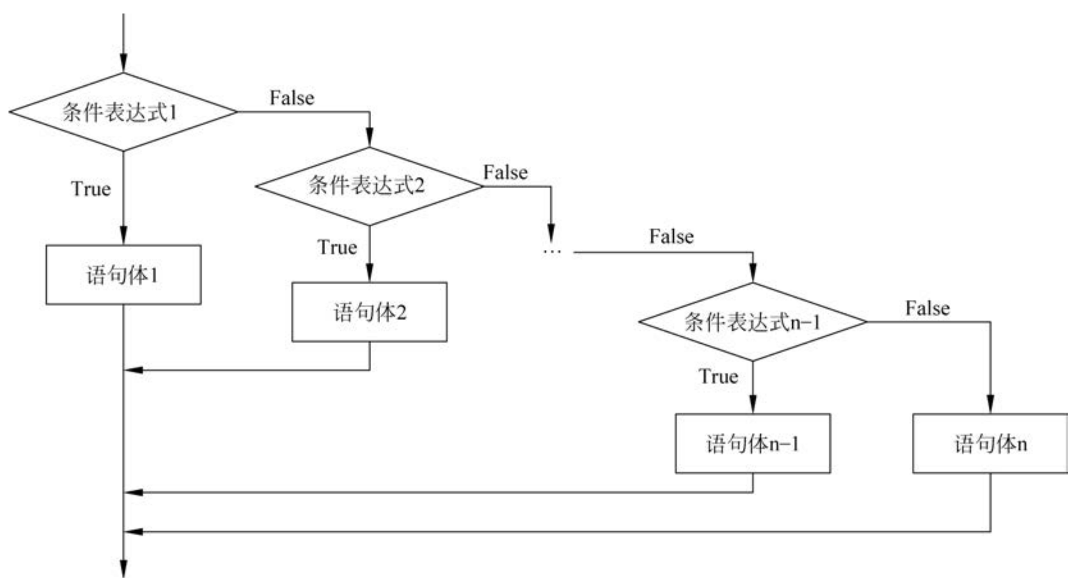

图 3.3　多分支 if/elif/else 语句流程图

if/elif/else 这种多分支结构先判断条件表达式 1 的真假。如果条件表达式 1 的结果为真(包括非零、非空),则执行语句体 1 中的操作,然后退出整个分支语句;如果条件表达式 1 的结果为假(包括零、空),则继续判断条件表达式 2 的真假;如果条件表达式 2 的结果为真(包括非零、非空),则执行语句体 2 中的操作,然后退出整个分支语句;如果条件表达式 2 的结果也为假(包括零、空),则继续判断表达式 3 的真假……从上到下依次判断条件表达式,找到第一个为真的条件表达式就执行该条件表达式下的语句体,不再判断剩余的条件表达式。如果所有的条件表达式均为假,并且最后有 else 语句部分,则执行 else 后面的语句体;如果此时没有 else 语句体,则不执行任何操作。任何一个分支的语句体执行完后,直接结束该分支结构。

语句体 1,语句体 2,…,语句体 n,既可以包含多条语句,也可以只由一条语句组成。

【例 3.4】 从键盘输入标准价格和订货量。根据订货量的大小给客户以不同的折扣率价格,计算应付货款(应付货款=订货量×价格×(1-折扣率))。订货量在 300 以下的,没有折扣;订货量在 300 及以上、500 以下的,折扣率为 3%;订货量在 500 及以上、1000 以下的,折扣率为 5%;订货量在 1000 及以上、2000 以下的,折扣率为 8%;订货量在 2000 及以上的,折扣率为 10%。

分析:键盘输入标准价格 price、订货量 Quantity,依照上述标准进行判断,得出折扣率。注意,还需要考虑输入的订货量和标准价格小于 0 时的错误情况。

程序源代码如下:

```
#example3_4.py
#coding = gbk
price = eval(input('请输入标准价格:'))
Quantity = eval(input("请输入订货量: "))

if Quantity < 0:
```

```
    Coff = -1
elif Quantity < 300:
    Coff = 0.0
elif Quantity < 500:
    Coff = 0.03
elif Quantity < 1000:
    Coff = 0.05
elif Quantity < 2000:
    Coff = 0.08
else:
    Coff = 0.1

if Quantity >= 0 and price >= 0:
    print("折扣率为:,"Coff)
    Pays = Quantity * price * (1 - Coff)
    print("支付金额:",Pays)
else:
    print("输入的订货量与标准价格均不能小于 0!")
```

程序 example3_4.py 第一次的运行结果如下：

```
请输入标准价格:10
请输入订货量: 500
折扣率为:0.05
支付金额: 4750.0
```

程序 example3_4.py 第二次的运行结果如下：

```
请输入标准价格:10
请输入订货量: -100
输入的订货量与标准价格均不能小于 0!
```

程序 example3_4.py 第三次的运行结果如下：

```
请输入标准价格:-10
请输入订货量: 200
输入的订货量与标准价格均不能小于 0!
```

请思考，需要输入多少个标准价格和订货量组成的测试数据，才能验证程序的每个分支都是正确的？

3.1.4　分支结构的嵌套

视频讲解

在分支结构的某一个分支的语句体中又嵌套新的分支结构，这种情况称为分支结构的嵌套（又称为选择结构的嵌套）。分支结构的嵌套形式因问题不同而千差万别，因此透彻分析每个分支的逻辑情况是编写程序的基础。

【例 3.5】 输入客户类型、标准价格和订货量。根据客户类型（<5 为新客户，≥5 为老客户）和订货量给予不同的折扣率，计算应付货款（应付货款＝订货量×价格×（1－折扣率））。

如果是新客户：订货量在 800 以下的，没有折扣，否则折扣率为 2%。如果是老客户：订货量在 500 以下的，折扣率为 3%；订货量在 500 及以上、1000 以下的，折扣率为 5%；订

货量在1000及以上、2000以下的，折扣率为8%；订货量在2000及以上的，折扣率为10%。请绘制流程图，并编写程序。

分析：输入数据后，应首先对客户类型、价格和订货量的输入值进行简单判断，判断其是否大于0。当这三个值均大于0时才开始做应付货款的计算，否则提示“输入错误”。数据输入正确后的处理流程如图3.4所示。

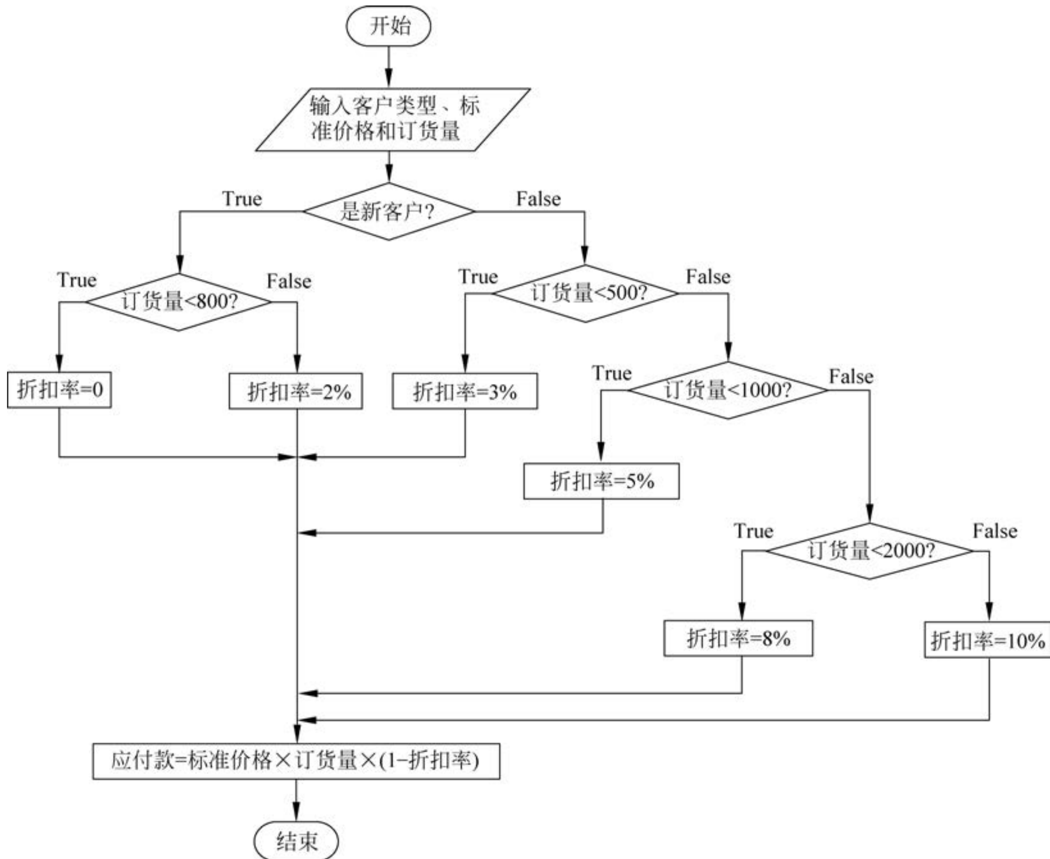

图3.4　例3.5的处理流程图

程序源代码如下：

```
#example3_5.py
#coding = gbk

Ctype = int(input("请输入客户类型(小于5为新客户):"))
Price = eval(input('请输入标准价格:'))
Quantity = eval(input("请输入订货量:"))

if Ctype > 0 and Price > 0 and Quantity > 0:
    if Ctype < 5:
        if Quantity < 800:
            Coff = 0
        else:
            Coff = 0.02
```

```
    else:
        if Quantity < 500:
            Coff = 0.03
        elif Quantity < 1000:
            Coff = 0.05
        elif Quantity < 2000:
            Coff = 0.08
        else:
            Coff = 0.1
    Pays = Quantity * Price * (1 - Coff)
    print("应付款为:",Pays)

else:
    print("输入错误。")
```

程序测试：运行程序，首先输入客户类型为4，标准价格为10，订货量为700，观察程序的运行结果。再次运行程序，输入客户类型为6，标准价格为10，订货量为700，观察程序的运行结果。

程序 example3_5.py 第一次的运行结果如下：

```
请输入客户类型(小于5为新客户):4
请输入标准价格:10
请输入订货量:700
应付款为: 7000.0
```

程序 example3_5.py 第二次的运行结果如下：

```
请输入客户类型(小于5为新客户):6
请输入标准价格:10
请输入订货量:700
应付款为: 6650.0
```

3.1.5　分支结构的三元运算

视频讲解

对于简单的 if/else 双分支结构，可以使用三元运算表达式来实现。例如：

```
x = 5
if x > 0:
    y = 1
else:
    y = 0
```

可以用三元运算改写为

```
x = 5
y = 1 if x > 0 else 0
```

结果完全一样。

if/else 双分支结构的三元运算表达式如下：

```
变量 = 值1  if  条件表达式  else  值2
```

如果条件表达式为真，变量取“值1”，否则变量取“值2”。

3.2 pass 语句

pass 是一个空语句，不做任何事情，一般只用作占位语句。使用 pass 语句是为了保持程序结构的完整性。在程序设计的过程中，可以用 pass 语句替代某些代码，在后续过程中再做补充。

```
>>> a = 5
>>> if a == 5:
        #没想好
else:

SyntaxError: expected an indented block
```

上述代码中，程序结构不完整，导致出错。

```
>>> if a == 5:
        pass                    #没想好,用 pass 来占位
else:
        print('ok')
```

上述代码使用了 pass 语句，程序结构完整，没有出现错误。

保留 pass 语句不影响其他语句的执行，例如：

```
>>> if a == 5:
        pass                    #没想好,用 pass 来占位
        print('再想想')
else:
        print('ok')
```

上述代码的运行结果如下：

```
再想想
```

3.3 基于模式匹配的 match/case 分支结构

Python 3.10 开始引入了基于模式匹配的 match/case 分支结构。该结构与 if 语句的分支结构功能类似，但更加灵活。match/case 分支的结构如下：

```
match 匹配对象:
    case 匹配表达式 1:
        分支 1
    case 匹配表达式 2:
        分支 2
    …
    case _:
        分支 n
```

match 后面的匹配对象依次对 case 后面的表达式进行匹配。当找到第一个匹配的 case 子句后，执行该子句所在分支的语句块，就不再去判断匹配对象与剩余的 case 后面表

达式是否匹配，然后结束整个 match/case 结构。如果没有一个 case 后面的表达式与 match 后面的对象相匹配，则执行“case _”分支（默认分支）后面的表达式；此时如果没有默认分支，则不执行任何分支。

3.3.1 匹配简单对象

case 后面的表达式可以是数值等简单对象或由这些对象构成的条件表达式。例如：

```
x = 5
match x:
    case 3:
        print("x = 3")
    case 5|6:               #运算符"|"表示或
        print("x = 5 或 x = 6")
    case _:                 #默认分支,当上述分支均不匹配时执行该分支
        print("x 不是 3、5 或 6 中的任何一个")
```

如上代码的运行结果如下：

```
x = 5 或 x = 6
```

match 结构也可以没有默认分支，就像 if 语句没有 else 分支一样。还可以在 case 后面使用变量，用该变量接收 match 后面的表达式值，并可以用 if 语句添加判断条件。例如：

```
x = 10
match x:
    case 3:
        print("x = 3")
    case 5|6:
        print("x = 5 或 x = 6")
    case y if y > 8:        # 用 if 语句添加判断条件
        print("x 大于 8")
        print("y = ",y,sep = "")
```

如上代码的运行结果如下：

```
x 大于 8
y = 10
```

上述代码中，前两个 case 后面的表达式都不能与 match 后面的表达式匹配。轮到第 3 个 case 时，先将 x 的值赋给 y。这样 y 得到 x 的值 10，并且条件表达式 y＞8 的计算结果为 True，因此执行该分支下的语句块。

3.3.2 匹配序列对象

在匹配列表或元组等序列时，需要长度和元素都匹配才表示匹配成功。例如：

```
x = [1,2]
match x:
    case [1]:
        print("匹配单个元素为 1 的序列")
    case [1,y]:
```

```
        print("匹配长度为 2 的序列,且第一个元素为 1,并将第 2 个元素赋值给 y")
```

如上代码的运行结果如下：

```
匹配长度为 2 的序列,且第一个元素为 1,并将第 2 个元素赋值给 y
```

在程序执行时，第一个 case 后面的列表长度与 match 后面的列表对象 x 的长度不匹配。第二个 case 后面的列表对象[1,y]与 x 在长度和元素值上都匹配。

序列内部元素的匹配中，也可以使用或运算符(|)，例如：

```
x = [1,2,"test"]
match x:
    case [1, (2|5), y]:
        print("匹配 3 个元素,第 2 个为 2 或 5,第 3 个元素不限")
    case _:
        print("都不匹配")
```

如上代码的运行结果如下：

```
匹配 3 个元素,第 2 个为 2 或 5,第 3 个元素不限
```

可以为序列中元素值的匹配添加条件。例如，以下程序中对序列的第三个元素匹配添加了字符串长度大于 0 的条件。例如：

```
x = [1,2,"test"]
match x:
    case [1, (2|5) as s, y] if len(y)> 0:
        print("共 3 个元素,为第 2 个元素赋予别名 s," +
             "并将匹配时的第 2 个元素值赋给 s,同时要求第 3 个元素 y 的长度大于 0")
    case _:
        print("都不匹配")
```

如上代码的运行结果如下：

```
共 3 个元素,为第 2 个元素赋予别名 s,并将匹配时的第 2 个元素值赋给 s,同时要求第 3 个元素 y 的
长度大于 0
```

程序执行后，可以查看 s 的值为 2，y 的值为'test'。

在匹配表达式中，也可以采用变量名前添加星号(*)的方式来表示序列中剩余的元素。例如，以下例子中，用 * rest 来匹配元组 x 中除第一个元素(这里值为 1)后的所有剩余元素(这里是 2 和"test")。

```
x = (1,2,"test")
match x:
    case [1|2 as y, *rest]:
        print("匹配第一项为 1 或 2 的序列")
    case _:
        print("没有找到匹配项")
```

如上代码的运行结果如下：

```
匹配第一项为 1 或 2 的序列
```

执行该程序后，y 被赋予 1；其余项构成列表[2,'test']，赋予变量 rest。变量名前面加

* 的含义将在第 6 章介绍。

3.3.3 匹配字典对象

case 后面也可以是字典。只要 case 表达式中的字典键(key)在 match 对象中存在,即表示匹配成功。例如:

```
d = {"number":1024,"name":"liu"}
match d:
    case {"name" : _}:
        print("匹配存在 key 为 name 的字典")
    case _:
        print("没有匹配项")
```

如上代码的运行结果如下:

```
匹配存在 key 为 name 的字典
```

在对字典的键进行匹配时,同时也可以要求匹配字典中值的类型。以下例子中,除了需要匹配字典中的两个键,还要求键 number 对应的值必须是整数类型。

```
d = {"number":1024,"name":"liu"}
match d:
    case {"name":n,"number":int(x)}:
        print("匹配 number 对应的值为整数,且存在 key 为 name 和 number 的字典")
        print("x = ",x,sep = "")
        print("n = ",n,sep = "")
    case _:
        print("没有匹配项")
```

如上代码的运行结果如下:

```
匹配 number 对应的值为整数,且存在 key 为 name 和 number 的字典
x = 1024
n = liu
```

匹配表达式中,可以采用变量名前面加两个星号(**)的方式来表示字典中剩余的元素。例如,以下代码中用 ** rest 来匹配字典 d 中除键 age 所对应元素外的所有剩余元素。

```
d = {"number":1024,"name":"liu","age":18}
match d:
    case {"age":int(a), ** rest}:
        print("匹配有 key 为 age 且其对应值为整数的字典")
        print("a = ",a,sep = "")
        print("rest = ",rest,sep = "")
    case _:
        print("没有匹配项")
```

如上代码的运行结果如下:

```
匹配有 key 为 age 且其对应值为整数的字典
a = 18
rest = {'number': 1024, 'name': 'liu'}
```

字典 d 中，键 age 的值为 18，与 case 子句的字典中"age"：int(a)相匹配。字典其余的键及相应的值与 ** rest 相匹配，并构成一个新的字典，赋值给变量 rest。因此，变量 a 被赋予了整数 18，变量 rest 被赋予了字典{'number'：1024，'name'：'liu'}。变量名前面加 ** 的含义将在第 6 章介绍。

当 match 的对象是类对象时，匹配的规则与字典类似。只要对象类型和对象的属性满足 case 后面的条件，就能匹配。对象中的属性个数可以超过 case 后面提到的属性个数。由于类与对象要在第 9 章介绍，这里不对对象的匹配展开阐述，读者可以在学习完第 9 章后再阅读 Python 官方文档或其他相关资料来了解此用法。

3.4 循环结构控制语句

Python 语言中包含 while 和 for 两种循环结构。while 循环结构是在给定的判断条件为真(包括非零、非空)时，重复执行某些操作；判断条件为假(包括零、空)时，结束循环。for 循环结构是当被遍历的可迭代对象中还有新的值可取时，重复执行某些操作；当被遍历的可迭代对象中没有新的值可取时，结束循环。

在介绍以上两种基本循环的简单结构后，本节将接着介绍与循环结构紧密相关的 break 和 continue 两种循环中断语句。在 break 语句的基础上，本节还将进一步介绍带 else 的循环结构。最后将介绍循环的嵌套结构。

视频讲解

3.4.1 简单 while 循环结构

简单的 while 循环语句结构如下：

```
while 条件表达式:
    循环体
```

简单的 while 循环由关键字 while、条件表达式、冒号、循环体构成。简单 while 循环结构的执行流程图如图 3.5 所示。其执行过程如下。

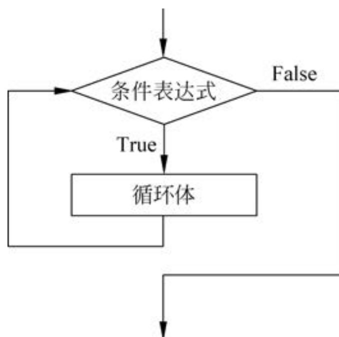

图 3.5 简单 while 循环结构的执行流程

(1) 计算 while 关键词后面条件表达式的值。如果其值为真(包括非零、非空)，则转步骤(2)；否则转步骤(3)。

(2) 执行循环体，转步骤(1)。

(3) 循环结束。

循环开始之前，如果 while 关键词后面条件表达式的值为假(包括零、空)，则不会进入循环体，直接跳过循环部分。如果一开始 while 关键词后面条件表达式的值为真(包括非零、非空)，则执行循环体；每执行完一次循环体，重新计算 while 关键词后面条件表达式的值，若为真，则继续执行循环体；循环体执行结束后重新判断 while 关键词后面的条件表达式；直到该条件表达式的值为假(包括零、空)，则结束循环。条件表达式中变量的取值决定条件表达式值的真假，该变量称为循环控制变量。

在使用 while 语句时，要注意以下几点。

(1) 组成循环体的各语句必须以相同的格式缩进。

(2) 循环体既可以由单条语句组成，也可以由多条语句组成。如果语句尚未确定，可以暂时使用 pass 语句表示空操作，但不能没有任何语句。

(3) 循环开始之前要为循环控制变量赋初值，使得 while 后面的条件表达式有初始的真、假值。

(4) 如果一开始 while 后面的条件表达式为假(包括零、空)，则不会进入循环；否则就进入循环，开始执行循环体。

(5) 循环体中要有语句改变循环控制变量的值，使得 while 后面的条件表达式因为该变量值的改变而可能出现结果为假(包括零、空)的情况，从而能够导致循环终止，否则会造成无限循环。

(6) Python 对大小写敏感，关键字 while 必须小写。

与 3.3 节中介绍的 if 语句比较，相同点和不同点如下。

(1) 相同点：两者都由表达式、冒号、缩进的语句体组成。并且都是在表达式的值为真时执行语句体。

(2) 不同点：if 语句执行完语句体后，马上退出 if 语句。while 语句执行完语句体后，立刻又返回到条件表达式重新计算，只要条件表达式的值为真，它会一直重复这一过程，直到条件表达式的值为假时才结束循环。

while 循环既可以用于解决循环次数确定的问题，也可以用于解决循环次数不确定的问题。下面分别讨论这两种使用方式。

1. 利用计数器解决循环次数确定的问题

循环次数确定的问题是指在编写程序或循环开始执行之前可以预知循环即将执行的次数。为了控制循环次数，通常在程序中设置一个计数变量(循环控制变量)，每次循环，该变量进行自增或自减操作，当变量值自增到大于设定的上限值或者自减到小于设定的下限值时，循环结束。

【例 3.6】 计算并输出小于或等于 200 的所有正偶数之和。

分析：设置变量 aInt 从 1 开始计数，每次增长 1，直到 aInt 超过 200，循环终止。可以预知循环执行 200 次。每次判断 aInt 是否为偶数，若是偶数就累加到和变量 sumInt 中。

程序的执行流程如图 3.6 所示。

程序源代码如下：

```
#example3_6.py
#coding = gbk
aInt = 1
sumInt = 0

while aInt <= 200:
    if aInt % 2 == 0:
        sumInt = sumInt + aInt
    aInt = aInt + 1

print('1～200 的偶数和:',sumInt)
```

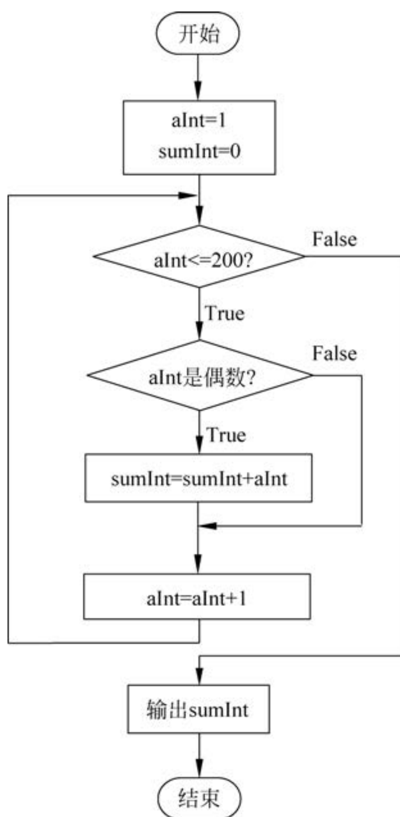

图 3.6 例 3.6 程序的执行流程

如上代码中,aInt 是循环控制变量,其初始值设为 1,每次循环步进为 1,其变化直接控制着循环的推进和次数。sumInt 的初始值为 0,用来累加 1～200 的偶数之和。

思考:在循环结束后,aInt 的值是多少?如果想要降低循环的次数,应该怎样修改程序?

如上代码的循环体共执行了 200 次。在循环结束时,aInt 的值是 201。如果要降低循环的次数,可以通过修改循环控制变量的初始值和每次的增长值来实现。

修改后的源代码如下:

```
#question3_2.py
#coding = gbk
aInt = 2
sumInt = 0

while aInt <= 200:
   sumInt = sumInt + aInt
   aInt = aInt + 2

print('1～200 的偶数和:',sumInt)
```

测试与思考:如果省略了语句 aInt＝aInt＋2,会出现什么运行结果?请总结该条语句的作用。如果省略了语句 sumInt＝0,会出现什么运行结果?将语句 sumInt＝0 放到循环体内,会产生怎样的结果?并请总结该条语句的作用。

如果要求 1～200 奇数的和,可以怎样修改程序?

2. 利用信号值解决循环次数不确定的问题

循环次数不确定的问题是指在编写程序或程序运行前无法预知循环将要执行的次数。为了控制循环,一般在程序中设置一个类似触发器的变量(循环控制变量)。每次循环,该变量接收一个新值,当该变量值达到某信号值时,循环结束。

【例 3.7】 从键盘输入公司某商品的所有订单销售额,编程实现对输入的销售额累加求和。当输入的值小于或等于 0 时终止该操作。

分析:在编写程序时或程序运行之前均无法预知用户将从键盘连续输入多少个大于 0 的值,只要还没有输入小于或等于 0 的值,就利用循环一直累加相应的输入值,直到输入小于或等于 0 的值(信号量),循环才终止。

程序源代码如下:

```
#example3_7.py
#coding = gbk
fSale = float(input('请输入订单的销售额:'))
sumSales = 0

while fSale > 0 :
```

```
    sumSales += fSale
    fSale = float(input('请输入下一个订单的销售额: '))

print('商品的销售总额为:',sumSales)
```

程序 example3_7.py 的运行结果如下：

```
请输入订单的销售额:12
请输入下一个订单的销售额: 56.8
请输入下一个订单的销售额: 30
请输入下一个订单的销售额: 98.2
请输入下一个订单的销售额: 0
商品的销售总额为: 197.0
```

3.4.2 简单 for 循环结构

for 循环结构通过遍历一个序列（如字符串、列表、元组、range）等可迭代对象中的每个元素来建立循环。

简单 for 循环结构语句的语法形式如下：

```
for 变量 in 序列或迭代器等可迭代对象:
    循环体
```

简单 for 循环结构的执行流程如图 3.7 所示。

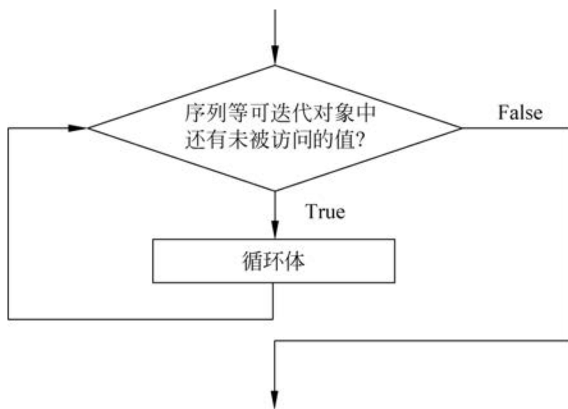

图 3.7 简单 for 循环结构的执行流程

循环开始时，for 关键词后面的变量从 in 关键词后面的序列或迭代器等可迭代对象中取其元素值，如果没有取到值，则不进入循环；如果可迭代对象中有值可取，则取到最前面的值，接着执行循环体。循环体执行完成后，for 关键词后面的变量继续取可迭代对象的下一个元素值，当没有值可取时，则终止循环；否则取到下一个元素值后继续执行循环体。然后重复以上过程，直到可迭代对象中没有新的值可取时，循环终止。

【例 3.8】 用列表存储若干城市的名称，利用 for 循环逐一输出城市名称。

程序源代码如下：

```
#example3_8.py
#coding = gbk
```

```
nameList = ['Beijing','Shanghai','Hangzhou','Nanjing','Taizhou','Wuhan']

print('城市名称列表:',end = " ")

for name in nameList:
   print(name,end = ' ')
```

程序 example3_8.py 的运行结果如下：

```
>>>
 === RESTART: D:\test\example3_8.py ===
城市名称列表: Beijing Shanghai Hangzhou Nanjing Taizhou Wuhan
>>>
```

在程序 example3_8.py 的每次循环过程中，变量 name 依次访问 nameList 列表中的一个字符串元素，然后执行循环体中的 print 语句，打印当前 name 变量值。print()函数输出结束时不换行，而是添加一个空格。

可以用 for 循环直接遍历 range 整数序列对象。例如：

```
>>> for i in range(0,10):
   print(i,end = ' ')
```

以上代码的运行结果为 0 1 2 3 4 5 6 7 8 9。

```
>>> for i in range(10):
   print(i,end = ' ')
```

以上代码的运行结果为 0 1 2 3 4 5 6 7 8 9。

```
>>> for i in range(3,15,1):
   print(i,end = ' ')
```

以上代码的运行结果为 3 4 5 6 7 8 9 10 11 12 13 14。

```
>>> for i in range(3,15):
    print(i,end = ' ')
```

以上代码的运行结果为 3 4 5 6 7 8 9 10 11 12 13 14。

```
>>> for i in range(3,15,2):
    print(i,end = ' ')
```

以上代码的运行结果为 3 5 7 9 11 13。

```
>>> for i in range(15,3, - 2):
   print(i,end = ' ')
```

以上代码的运行结果为 15 13 11 9 7 5。

range 对象经常被用到 for 循环结构中，用于遍历序列的索引值。例 3.8 也可以使用以下方法实现。

```
#example3_8_1.py
#coding = gbk
nameList = ['Beijing','Shanghai','Hangzhou','Nanjing','Taizhou','Wuhan']
```

```
print('城市名称列表:',end = " ")

for i in range(len(nameList)):
    print(nameList[i],end = ' ')
```

程序 example3_8_2.py 的运行结果如下：

```
>>>
 == RESTART: D:\test\example3_8_2.py ==
城市名称列表: Beijing Shanghai Hangzhou Nanjing Taizhou Wuhan
>>>
```

语句 range(len(nameList))先求 len(nameList)的值为 6；然后执行 range(6)，生成元素为 0、1、2、3、4、5 的可迭代对象。在 for 循环中，i 依次取可迭代对象中的值。将这个值作为访问列表 nameList 中元素的索引(即元素在列表中所处的位置)。通过 nameList[i]语句获取索引 i 对应的列表中的元素。

3.4.3 break 语句和 continue 语句

视频讲解

break 语句可以在 while 和 for 循环中用于提前终止循环。在循环的进行过程中，如果执行了 break 语句，则循环体中该 break 语句之后的部分不再执行并终止循环；如果 break 语句在具有两层循环嵌套的内层循环中，则只终止内层循环，进入外层循环的下一条语句继续执行。在多层嵌套的循环结构中，break 语句只能终止其所在层的循环。循环体中 break 语句是否执行，通常由分支结构来判断。

图 3.8 给出了循环体中含 break 语句的 while 循环结构执行流程。图 3.9 给出了循环体中含 break 语句的 for 循环结构执行流程。其中循环体 1、break 语句和循环体 2 三部分共同构成循环体。

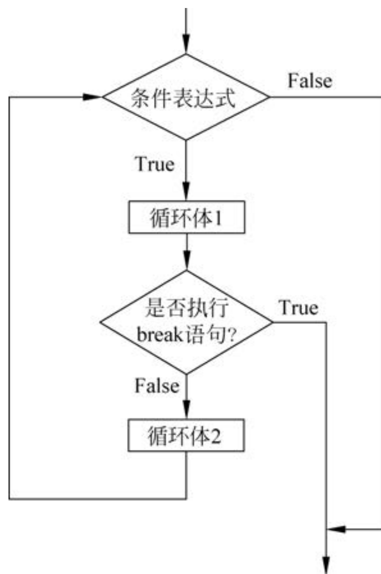

图 3.8 循环体中含 break 语句的 while 循环

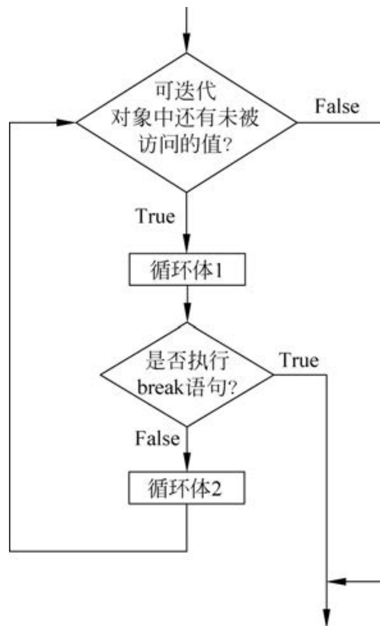

图 3.9 循环体中含 break 语句的 for 循环

【例3.9】 求一个大于1的自然数除了自身以外的最大约数(因子)。

分析：为了寻找一个自然数num除自身以外的最大因子，可以使用循环结构。让循环控制变量count的初值为num－1，只要count的值大于或等于1，进行如下循环。

循环体步骤1：判断num除以count的余数是否为0，若为0，该count值就是num的最大因子，利用break语句提前终止循环，否则进入循环体步骤2。

循环体步骤2：让count的值减1。

程序源代码如下：

```
#example3_9_1.py
#coding = gbk
num = int(input('请输入一个大于1的自然数:'))
count = num - 1

while count > 0:
    if num % count == 0:
        print(count,'is the max factor of ',num)
        break

    count = count - 1
```

一个大于1的自然数除了自身以外的最大因子不会超过其除以2的整数商。因此可以将其除以2后的整数商作为循环控制变量的初始值，从而减少循环次数。

程序源代码如下：

```
#example3_9_2.py
#coding = gbk
num = int(input('请输入一个大于1的自然数:'))
count = num//2

while count > 0:
    if num % count == 0:
        print(count,'is the max factor of ',num)
        break

    count = count - 1
```

程序example3_9_2.py的运行结果如下：

```
请输入一个大于1的自然数:27
9 is the max factor of  27
```

程序example3_9_2.py的执行过程如下：

```
num = 27
count = 13
```

进入循环体：

因为27除以13的余数不为0，break不会执行，count减1后的值为12；

因为27除以12的余数不为0，break不会执行，count减1后的值为11；

因为27除以11的余数不为0，break不会执行，count减1后的值为10；

因为 27 除以 10 的余数不为 0，break 不会执行，count 减 1 后的值为 9；

因为 27 除以 9 的余数为 0，执行 break，提前终止循环。

输出：

```
9 is the max factor of  27
```

continue 语句可以用在 while 和 for 循环中。循环体中如果执行了 continue 语句，本次循环跳过循环体中 continue 语句之后的剩余语句，回到循环开始的地方重新判断是否进入下一次循环。在嵌套循环中，continue 语句只对其所在层的循环起作用。

图 3.10 给出了循环体中含 continue 语句的 while 循环结构执行流程。图 3.11 给出了循环体中含 continue 语句的 for 循环结构执行流程。其中循环体 1、continue 语句和循环体 2 三部分共同构成循环体。

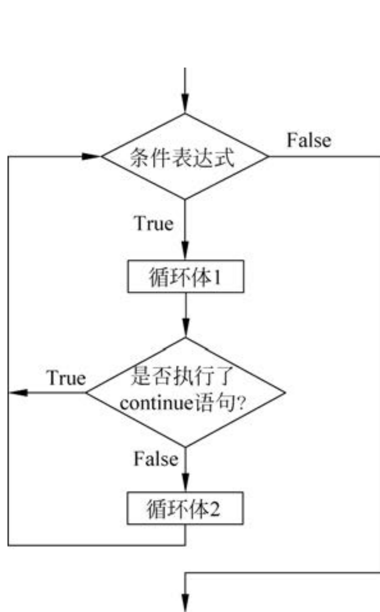

图 3.10 循环体中含 continue 语句的 while 循环

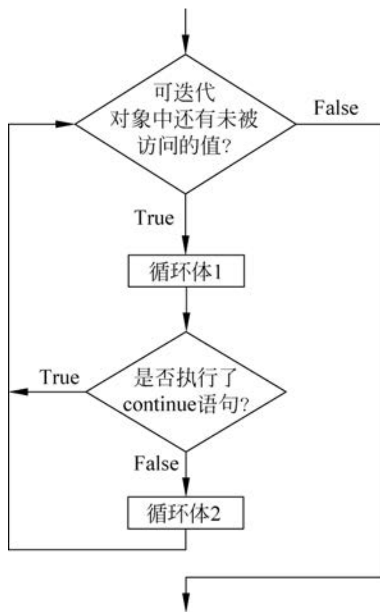

图 3.11 循环体中含 continue 语句的 for 循环

break 语句与 continue 语句的主要区别如下。

(1) break 语句一旦被执行，循环体中 break 语句之后的部分不再执行，且终止该 break 所在层的循环。

(2) continue 语句的执行不会终止整个当前循环，只是提前结束本次循环，本次循环跳过循环体中 continue 语句之后的剩余语句，提前回到循环开始的地方，重新判断是否进入下一次循环。

【例 3.10】 阅读以下两个程序，理解 break 语句和 continue 语句的区别。

程序源代码如下：

```
#example3_10_1.py
strs = ['Mike','Tom','Null','Apple','Betty','Null','Amy','Dick']

for astr in strs:
```

```
    if astr == 'Null':
        break               #遇到单词 Null,则终止循环
    print(astr)

print('End')
```

程序 example3_10_1.py 的运行结果如下：

```
Mike
Tom
End
```

程序源代码如下：

```
#example3_10_2.py
strs = ['Mike','Tom','Null','Apple','Betty','Null','Amy','Dick']
for astr in strs:
    if astr == 'Null':
        continue            #遇到单词 Null,则跳过该单词
    print(astr)
print('End')
```

程序 example3_10_2.py 的运行结果如下：

```
Mike
Tom
Apple
Betty
Amy
Dick
End
```

第一种情况下,if 语句里面是 break 语句。当触发了条件(即取到的字符串是'Null')则执行 break 语句,直接终止了循环,因此只输出了两个姓名 Mike 和 Tom。

第二种情况下,if 语句里面是 continue 语句。当触发了条件(即取到的字符串是'Null')则执行 continue 语句,只终止了当次循环,本次循环跳过循环体中 continue 语句之后的部分,提前进入下一次循环(即取得下一个字符串),因此输出了所有不是 Null 的姓名 Mike、Tom、Apple、Betty、Amy、Dick。

视频讲解

3.4.4 带 else 的循环结构

前面介绍了简单的 while 和 for 循环结构。与很多程序设计语言不同,Python 中的 while 和 for 语句后面还可以带有 else 语句块。

1. 带 else 的 while 循环

带 else 的 while 循环结构语法形式如下：

```
while 条件表达式:
    循环体
else:
    else 语句块
```

当条件表达式为真(True、非空、非零)时,反复执行循环体。当循环因为 while 后面的条件表达式为假(False、零、空)而导致不能进入循环或循环终止,else 语句块执行一次,然后结束该循环结构。如果该循环是因为执行了循环体中的 break 语句而导致循环终止,else 语句块不会执行,直接结束该循环结构。

如果循环体中没有 break 语句,带 else 语句块的 while 循环执行流程可以用图 3.12(a)表示。如果循环体中包含 break 语句,带 else 语句块的 while 循环执行流程可以用图 3.12(b)表示。图 3.12(b)中,循环体 1、break 语句部分、循环体 2 共同构成循环体。

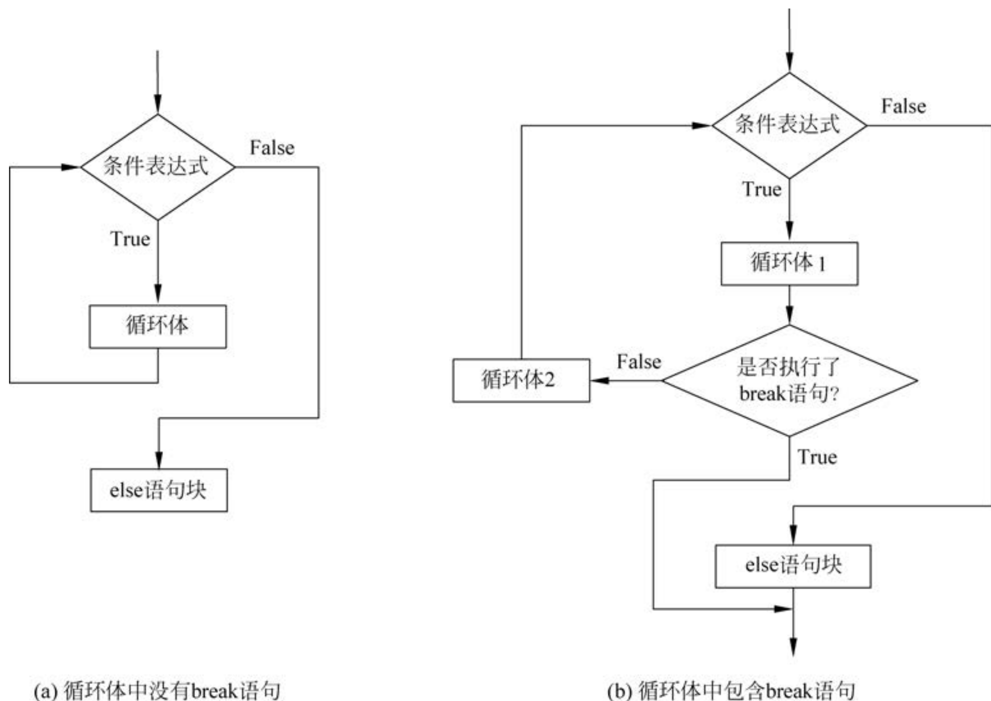

图 3.12 while/else 循环结构的执行流程

【例 3.11】 从键盘输入一个正整数 n,用 while 循环找出小于或等于该正整数 n 且能被 23 整除的最大正整数。如果找到了,输出该正整数;如果没有找到,则输出“未找到”。

程序源代码如下:

```
#example3_11.py
#coding = gbk
n = int(input('请输入一个正整数:'))
i = n
while i > 0:
    if i % 23 == 0:
        print("小于或等于",n,"且能被 23 整除的最大正整数是:",i)
        break
    i = i - 1
else:
    print("未找到。")
```

程序 example3_11.py 的一种运行结果如下:

```
请输入一个正整数:20
未找到。
```

程序 example3_11.py 的另一种运行结果如下：

```
请输入一个正整数:100
小于或等于 100 且能被 23 整除的最大正整数是: 92
```

2. 带 else 的 for 循环

带 else 的 for 循环结构语法形式如下：

```
for 变量 in 序列或迭代器等可迭代对象:
    循环体
else:
    else 语句块
```

当变量能够从 in 后面的序列或迭代器等可迭代对象中取到值，则执行循环体。循环体执行结束后，变量重新从序列或迭代器等可迭代对象中取值。当变量从 in 后面的序列或迭代器等可迭代对象中取不到新的值时，则循环终止，else 语句块执行一次，然后终止循环结构。当循环是因为循环体中执行了 break 语句而导致终止时，则 else 语句块不执行，直接终止循环结构。

如果循环体中没有 break 语句，带 else 语句块的 for 循环执行流程可以用图 3.13(a)表示，如果循环体中带有 break 语句，带 else 语句块的 for 循环执行流程可以用图 3.13(b)表示。图 3.13(b)中，循环体 1、break 语句部分、循环体 2 共同构成循环体。

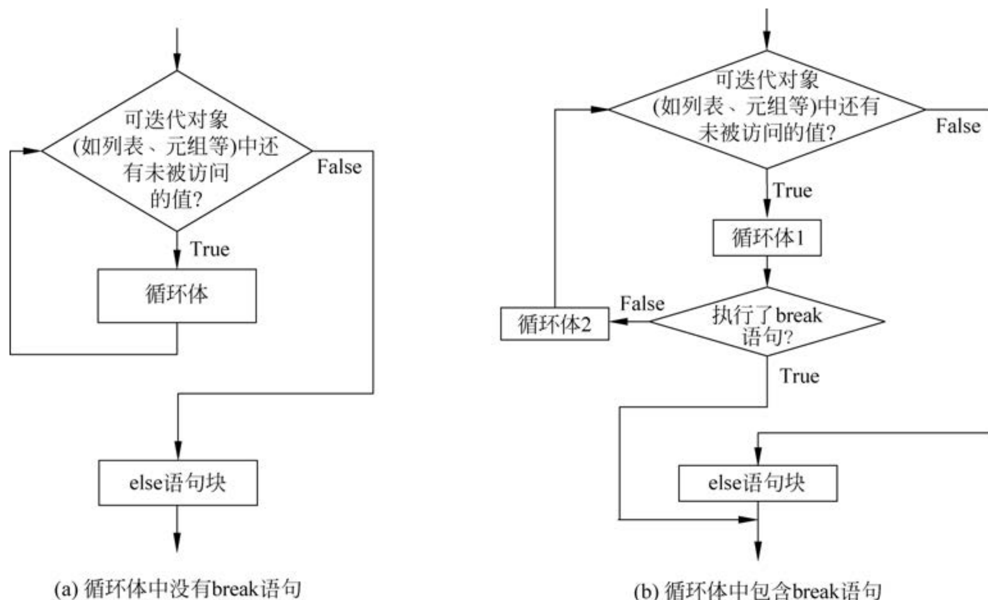

图 3.13　for/else 结构执行流程图

【例 3.12】　有一个列表 sales=[5000,3000,8000,10600,6000,5000]。该列表中的元素依次表示某产品 1～6 月的销售额。请用 for 循环遍历该列表，找到第一个销售额大于或等于 6000 的元素，并打印该元素的值。如果没有找到，则输出“未找到”。

程序源代码如下：

```
#example3_12.py
#coding = gbk
sales = [5000, 3000, 8000, 10600, 6000, 5000]

for i in sales:
    if i >= 6000:
        print("第一个大于或等于 6000 的销售额是:",i)
        break
else:
    print('未找到。')
```

程序 example3_12.py 的运行结果如下：

```
第一个大于或等于 6000 的销售额是: 8000
```

3.4.5 循环的嵌套

视频讲解

循环的嵌套是指在一个循环中又包含另外一个完整的循环，即循环体中又包含循环结构。循环嵌套的执行的过程是，先进入外层循环第 1 轮，然后执行完所有内层循环，接着进入外层循环第 2 轮，然后再次执行完内层循环，以此类推，直到外层循环执行完毕。

while 循环里面可以嵌套 while 循环，for 循环里面可以嵌套 for 循环。同时，while 循环和 for 循环也可以相互嵌套。循环的嵌套可以有很多层。典型的语法形式如下：

```
while 条件表达式 1 :
    语句体 1 - 1
    while 条件表达式 2 :
        循环体 2
    语句体 1 - 2
```

```
for 变量 1 in 可迭代对象 1 :
    语句体 1 - 1
    for 变量 2 in 可迭代对象 2:
        循环体 2
    语句体 1 - 2
```

```
while 条件表达式 :
    语句体 1 - 1
    for 变量 in 可迭代对象 :
        循环体 2
    语句体 1 - 2
```

```
for 变量 in 可迭代对象 :
    语句体 1 - 1
    while 条件表达式 :
        循环体 2
    语句体 1 - 2
```

【例 3.13】 利用 $e=1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\cdots+\frac{1}{n!}$，编写程序计算 e 的近似值。要求直到最后一项的值小于 10^{-8} 时，计算终止。输出最后一个 n 的值及 e 的值。

分析： 将第一项 1 设为 e 的初始值。其他项为$\frac{1}{n!}$，其中 n 的值为从 1 开始的自然数，直到$\frac{1}{n!}<10^{-8}$。while 循环的条件表达式用 True，自动进入下一轮循环。计算 n!，并将当前$\frac{1}{n!}$项加入 e 中。如果当前$\frac{1}{n!}$的值小于 10^{-8}，则利用 break 语句终止循环；否则让 n 递增 1，进入下一轮循环。

程序源代码如下：

```
#example3_13.py
#coding = gbk
e = 1
n = 1

while True:                    #始终循环,直到执行 break 语句终止循环
    s = 1                      #计算新的阶乘之前,初值重新设置为 1
    for i in range(1,n + 1):
        s = s * i
    e = e + 1/s
    if 1/s < 1e - 8 :
        break                  #终止循环
    n = n + 1

print("n = ",n)
print("e = ",e)
```

程序 example3_13.py 的运行结果如下：

```
n =  12
e =  2.7182818282861687
```

【思考题】 改写程序,去除内层循环,提高程序的执行效率。

分析：利用 n！ *（n+1）==（n+1）！这个等式,循环第 n 轮结束时变量 s 保存了 n！的值,下一轮循环计算(n+1)！时,只需 s=s*(n+1)即可。

程序源代码如下：

```
#question3_3.py
#coding = gbk
e = 1
n = 1
s = 1

while True:                    #始终循环,直到执行 break 语句终止循环
    s =  s * n
    e = e + 1/s
    if 1/s < 1e - 8 :
        break                  #终止循环
    n = n + 1

print("n = ",n)
print("e = ",e)
```

视频讲解

3.4.6 嵌套循环中的 break 语句和 continue 语句

前面介绍的 break 语句只在一重循环中使用。在例 3.13 的程序 example3_13.py 中,虽然出现了嵌套循环,但是这个例子中的 break 语句实际上放在外层循环中,与内层的 for 循环一样,共同作为外层循环的循环体,所以该 break 语句明显与内层循环没有关系。如果执行了该 break 语句,将直接终止外层 while 循环。

如果 break 语句在具有两层循环嵌套的内层循环中，则只终止内层循环，然后进入外层循环的下一条语句继续执行。

【例 3.14】 程序反复接收自然数的输入，直到输入－1 为止，计算并输出输入的自然数除了自身以外的最大约数。

分析：程序提示用户输入一个自然数，如果该数不为－1，则根据例 3.9 的方法计算并输出该自然数除自身外的最大约数；然后提示用户再输入一个自然数，利用循环重新计算并输出该自然数除自身外的最大约数；直到输入为－1 时终止该程序。

程序源代码如下：

```
#example3_14.py
#coding = gbk
num = int(input('请输入一个自然数:'))
while num!= -1:
    count = num//2
    while count > 0:
        if num % count == 0:
            break
        count = count - 1

    print(count,'is the max factor of ',num)
    num = int(input('请再输入一个自然数:'))

print('程序结束')
```

程序 example3_14.py 的一次运行结果如下：

```
请输入一个自然数:15
5 is the max factor of 15
请再输入一个自然数:27
9 is the max factor of 27
请再输入一个自然数:28
14 is the max factor of 28
请再输入一个自然数:36
18 is the max factor of 36
请再输入一个自然数: - 1
程序结束
```

程序 example3_14.py 的运行过程如下：

num = 15

进入外层循环:

count = 7

进入内层循环:

因为 15 除以 7 的余数不为 0,所以 count = 6

因为 15 除以 6 的余数不为 0,所以 count = 5

因为 15 除以 5 的余数为 0,所以输出"5 is the max factor of 15",然后遇到 break,内层循环结束。进入外层循环的下一条语句。

输入另一个 num = 27,回到外层循环起始语句:

count = 13

进入内层循环:

```
        因为27除以13的余数不为0,所以count = 12
        因为27除以12的余数不为0,所以count = 11
        因为27除以11的余数不为0,所以count = 10
        因为27除以10的余数不为0,所以count = 9
因为27除以9的余数为0,所以输出"9 is the max factor of 27",然后遇到break,内层循环结束。进
入外层循环的下一条语句。
    输入另一个num = 28,回到外层循环起始语句:
    …
    输入另一个num = 36,回到外层循环起始语句:
    …
    输入另一个num = -1,结束外层循环
输出"程序结束"
```

【例3.15】 寻找并打印输出所有三位数的素数。

分析:可以用循环取出100～999的每个数赋给变量i,对每个给定的i,用循环语句判断它是否为素数,若为素数,则打印输出。需要使用循环嵌套。

程序源代码如下:

```
#example3_15.py
# -*- coding: cp936 -*-
print("所有三位数的素数如下:",end = " ")
for i in range(100,1000):
    j = 2
    flag = 1
    while j < i:
        if i % j == 0:
            flag = 0
            break
        j += 1
    if flag == 1:
        print(i,end = " ")
```

程序example3_15.py的运行结果如下:

```
所有三位数的素数如下: 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331
337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467
479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631
641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797
809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967
971 977 983 991 997
```

实际上,不管有多少层的循环嵌套,循环体中一个break语句的执行,只是终止该break语句所在的循环体,并且从break语句往外层搜索离该break语句最近的while或for循环,然后终止该循环。

如果执行循环结构else语句块中的break语句,将终止其上一层循环,而不仅是终止该else语句块。此else语句块所在的循环已经执行到else语句块,所以其所在的循环自然已经终止,否则不会执行到该语句块。

【例 3.16】　编写程序，寻找大于或等于 500 且小于 1000 的最小素数。

程序源代码如下：

```
#example3_16_1.py
#coding = utf - 8
for i in range(500,1000):
    for j in range(2,i):
        if i % j == 0:
            break           #该 break 语句终止内层循环
    else:
        print('求解范围内的最小素数为:',i,sep = "")
        break               #该 break 语句终止外层循环
```

程序 example3_16_1.py 的运行结果如下：

```
>>>
============= RESTART: D:\example3_16_1.py =============
求解范围内的最小素数为:503
>>>
```

在多层循环中，continue 语句的作用范围与 break 语句类似。不管有多少层的循环嵌套，一个 continue 语句的执行，只是跳过 continue 语句所在本层循环中循环体的剩余语句。

【例 3.17】　修改例 3.10 中的要求，对列表['Mike','Tom','Null','Apple','Betty','Null','Amy','Dick']中每个非 Null 的单词依次输出其组成的字母，如果遇到字母 i，则不输出。

程序源代码如下：

```
#example3_17.py
#coding = utf - 8
strs = ['Mike','Tom','Null','Apple','Betty','Null','Amy','Dick']

for astr in strs:
    if astr == 'Null':
        continue            #作用于外层循环

    for s in astr:
        if s == 'i':
            continue        #作用于内层循环
        print(s,end = ' ')

    print()

print('End')
```

程序 example3_17.py 的运行结果如下：

```
M k e
T o m
A p p l e
B e t t y
A m y
```

```
D c k
End
```

3.5 控制结构的应用实例

【例 3.18】 输入若干同学的计算机成绩，成绩分布在[0,120]区间内。求出这些同学的计算机成绩平均值、最小值和最大值。输入出现负数时终止输入，且该负数不计入统计范围。

分析：因为平均值是所有成绩之和再除以人数，所以设置总分变量 iSum 初始值为 0，计数总人数的变量 sCnt 为 0。因为需要求成绩的最大值和最小值，所以设置成绩最大值变量 sMax 在循环开始前是一个非常小的数，譬如是－100；设置成绩最小值变量 sMin 在循环开始前是一个非常大的数，譬如是 150。

在程序运行时依次输入若干同学的计算机成绩，存入变量 aScore，以输入负数结束输入。每输入一名同学的成绩就进行以下操作。

(1) 将该学生的计算机成绩累加到变量 iSum 中。

(2) 对人数计数变量 sCnt 增加 1。

(3) 判断该学生的成绩与成绩最大值的关系，如果该生成绩大于成绩最大值，则将成绩最大值修改为该生的成绩值，否则不做任何操作。

(4) 判断该学生的成绩与成绩最小值的关系，如果该生成绩小于成绩最小值，则将成绩最小值修改为该生的成绩值，否则不做任何操作。

(5) 输入下一名学生的成绩，继续做上述步骤(1)～步骤(4)的操作，直到输入负数结束。

通过上述分析可见，需要利用循环控制结构实现上述步骤(1)～步骤(5)操作，循环结束的条件是输入的成绩值为负数。而对变量 iSum、sCnt、sMax 和 sMin 的赋初值要放到循环体以外。步骤(3)和步骤(4)需要用分支控制结构实现。而步骤(5)的输入下一名学生的成绩，是推动程序进入下一轮循环的关键。

程序源代码如下：

```
#example3_18.py
#coding = gbk
iSum = 0
sCnt = 0
sMax = - 100
sMin = 150
aScore = int(input('请输入一名同学的成绩:'))

while aScore >= 0:
    iSum = iSum + aScore
    sCnt = sCnt + 1
    if aScore > sMax:
        sMax = aScore
    if aScore < sMin:
        sMin = aScore
```

```
    aScore = int(input('请输入下一名同学的成绩:'))
print('计算机平均成绩:',iSum/sCnt)
print('计算机成绩最高分:',sMax)
print('计算机成绩最低分:',sMin)
```

程序 example3_18.py 的一次运行结果：

```
请输入一名同学的成绩:65
请输入下一名同学的成绩:70
请输入下一名同学的成绩:56
请输入下一名同学的成绩:89
请输入下一名同学的成绩:100
请输入下一名同学的成绩:95
请输入下一名同学的成绩:78
请输入下一名同学的成绩:88
请输入下一名同学的成绩:94
请输入下一名同学的成绩:103
请输入下一名同学的成绩:7
请输入下一名同学的成绩: - 1
计算机平均成绩: 76.81818181818181
计算机成绩最高分: 103
计算机成绩最低分: 7
```

思考：如果正确成绩位于[0,100]这个区间，也就是最高分只能是100分，那么我们就输入了一个错误的分数103。那么如何修改程序，可以使我们在输错成绩时有提示出现，并可以继续输入其他成绩呢？

习题 3

1. 从键盘输入一个百分制的成绩(0～100)存放在变量score中，要求输出其对应的成绩等级A～E。其中，score >= 90，则输出'A'；80 <= score < 90，则输出'B'；70 <= score < 80，则输出'C'；60 <= score < 70，则输出'D'；score < 60，则输出'E'。

2. 某电商平台上销售不同规格包装、不同价格的水笔。编写程序，在不考虑运费的情况下，从键盘分别输入两种规格包装水笔的支数和价格，分别计算单根水笔的价格，根据价格就低原则打印输出选择购买哪种包装的产品。

3. 输出1000以内的素数以及这些素数之和(素数是指除了1和该数本身之外，不能被其他任何整数整除的数)。

4. 编写程序，按公式 $s=1^2+2^2+3^2+\cdots+n^2$ 求累加和s小于1000的最大项数n，程序的运行结果如下：

```
n    s
1    1
2    5
3    14
4    30
5    55
6    91
```

```
7    140
8    204
9    285
10   385
11   506
12   650
13   819
14   1015
```

累计和不超过1000的最大项是n=13。