视频讲解

第3章 程序流程控制

在 Python 程序中，对于语句的执行有三种基本的控制结构，即顺序结构、选择结构、循环结构。另外，当程序出错时，Python 使用异常处理流程进行处理。

3.1 程序的流程

3.1.1 输入、处理和输出(IPO)

无论程序的规模如何，每个程序都可以分为以下三个部分：程序通过输入接收待处理的数据(input))；执行相应的处理(process)；通过输出(output)返回处理的结果。该过程通常称为 IPO 程序编写方法。其示意图如图 3-1 所示。

图 3-1 程序的输入、处理和输出示意图

(1) 输入数据。输入是一个程序的开始。程序要处理的数据有多种来源，形成了多种输入方式，包括交互输入、参数输入、随机数据输入、文件输入、网络输入等。

(2) 处理数据。处理是程序对输入数据进行计算产生输出结果的过程。计算问题的处理方法统称为“算法”。

(3) 输出结果。输出是程序输出结果的方式。程序的输出方式包括控制台输出、图形输出、文件输出、网络输出等。

【例 3.1】 计算球体的表面积和体积的程序的 IPO 描述。

输入(I)：输入 r(球体的半径)

处理(P)：计算球体的表面积 s = 4 * math.pi * r * r

计算球体的体积 v = 4 * math.pi * r * r * r / 3

输出(O)：输出 s 和 v

3.1.2 算法和数据结构

程序还可以使用以下公式描述：

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。算法通常与数据结构紧密相关。算法可以描述为："建立一个特定的数据结构，然后采用某种方式使用该数据结构"。

描述算法的最简单方法是使用自然语言描述。对于较复杂的算法，为了描述其细节，往往采用伪代码进行描述。伪代码是一种类似于程序设计语言的文本，其目的是为读者提供在代码中实现算法所需的结构和细节，而无须将算法局限于特定的程序设计语言。

【例 3.2】 求解两个整数最大公约数的算法的自然语言描述。

求解两个整数的最大公约数(Great Common Divisor，GCD)的一种算法是辗转相除法，又称欧几里得算法。辗转相除法算法的自然语言描述如下。

(1) 对于已知的两个正整数 m 和 n，使得 m>n。

(2) m 除以 n 得到余数 r。

(3) 若 r≠0，则令 m←n，n←r，重复步骤(2)，继续 m 除以 n 得到新的余数 r。若仍然 r≠0，则重复此过程，直到 r=0 为止。最后的 m 就是最大公约数。

【例 3.3】 求解两个整数最大公约数的辗转相除算法的伪代码描述。

```
//求解 m 和 n 的最大公约数.GCD(m, n) = GCD(n, m Mod n).
GCD(m, n)
    While (n != 0)
        remainder = m Mod n      //计算余数
        m = n
        n = remainder
    End While
    Return m
End Gcd
```

伪代码没有特定对应的语言规则。本书采用本例中类似 Python 缩进规则的伪代码描述。

【例 3.4】 求解两个整数的最大公约数的辗转相除算法的 Python 代码实现(gcd.py)。

```
#求解 m 和 n 的最大公约数.GCD(m, n) = GCD(n, m Mod n)
def gcd(m,n):
    if (m < n):
        m, n = n, m
    while (n != 0):
        remainder = m % n          #计算余数
        m = n
        n = remainder
    return m
```

```
if __name__ == '__main__':
    print(24,36,"的最大公约数为：",gcd(24,36))
```

程序运行结果如下：

```
24 36 的最大公约数为：12
```

3.1.3　程序流程图

程序流程图(flow chart)又称为程序框图，是描述程序运行具体步骤的图形表示。通过标准符号详细描述程序的输入、处理和输出过程，可以作为程序设计的最基本依据。

流程图的基本元素主要包括以下几种。

(1) 开始框和结束框()。表示程序的开始和结束。

(2) 输入/输出框()。表示输入和输出数据。

(3) 处理框()。表示要执行的流程或处理。

(4) 判断框()。表示条件判断，根据判断的结果执行不同的分支。

(5) 箭头线()。表示程序或算法的走向。

【例 3.5】　使用程序流程图(如图 3-2 所示)描述计算所输入数据 a 的平方根的程序。

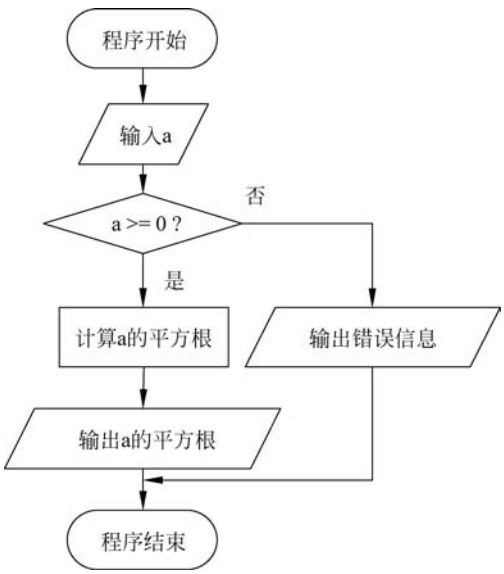


图 3-2　计算所输入数据平方根的流程图

常用的程序结构包括顺序结构、选择结构和循环结构。本章将陆续展开阐述。

3.2 顺序结构

程序中语句执行的基本顺序按各语句出现位置的先后次序执行，称之为顺序结构，参见图 3-3。在图 3-3 中先执行语句块 1，再执行语句块 2，最后执行语句块 3。三个语句块之间是顺序执行关系。

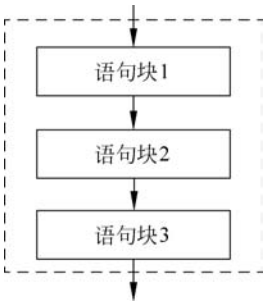


图 3-3 顺序结构示意图

【例 3.6】 顺序结构示例(area.py)：输入三角形三条边的边长(为简单起见，假设这三条边可以构成三角形)，计算三角形的面积。提示：三角形面积 $=\sqrt{h*(h-a)*(h-b)*(h-c)}$，其中，a、b、c 是三角形三边的边长，h 是三角形周长的一半。

```
import math
a = float(input("请输入三角形的边长 a: "))
b = float(input("请输入三角形的边长 b: "))
c = float(input("请输入三角形的边长 c: "))
h = (a + b + c) / 2                                    #三角形周长的一半
area = math.sqrt(h*(h-a)*(h-b)*(h-c));                 #三角形面积
print(str.format("三角形三边分别为: a={0},b={1},c={2}", a, b, c))
print(str.format("三角形的面积 = {0}", area))
```

程序运行结果如下：

```
请输入三角形的边长 a: 3
请输入三角形的边长 b: 4
请输入三角形的边长 c: 5
三角形三边分别为: a=3.0,b=4.0,c=5.0
三角形的面积 = 6.0
```

3.3 选择结构

选择结构可以根据条件来控制代码的执行分支，也称为分支结构。Python 使用 if 语句来实现分支结构。

3.3.1 分支结构的形式

分支结构包含单分支、双分支和多分支等多种形式，流程如图 3-4(a)～(c)所示。

3.3.2 条件表达式

条件表达式通常用于选择语句中，用于判断是否满足某种条件。在分支结构中，根据条件表达式的求值结果(True 或 False)执行程序不同的分支。

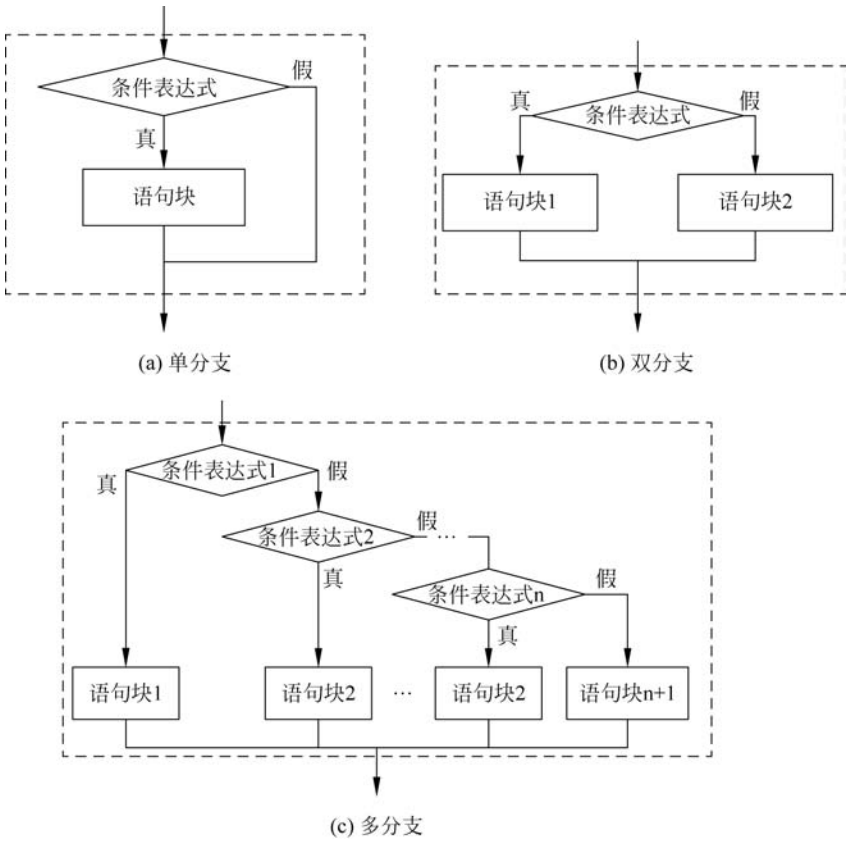


图 3-4　if 语句的选择结构

最简单的条件表达式可以是一个常量或变量，复杂的条件表达式包含关系比较运算符、测试运算符和逻辑运算符。条件表达式的最后评价为 bool 值 True(真)或者 False(假)。

Python 评价方法如下：如果条件表达式的结果为数值类型(0)、空字符串("")、空元组(())、空列表([])、空字典({})，则其 bool 值为 False(假)；否则其 bool 值为 True(真)。例如，123、"abc"、(1,2)均为 True。

3.3.3　关系和测试运算符与关系表达式

关系运算符用于比较两个对象的大小，测试运算符用于测试两个对象的关系。包含关系和测试运算符的表达式称之为关系表达式。若关系成立，则关系表达式的结果为 True，否则为 False。

关系和测试运算符是二元运算符。Python 支持连写多个比较运算符的关系表达式。例如：

```
>>> 2 > 1                    # 输出：True
>>> 1 < 2 < 3                # 输出：True
```

原则上，关系运算符应该是两个相同类型对象之间的比较。不同类型的对象也允许进行比较，但可能会导致错误。数值类型(包括布尔型，True 自动转换为 1，False 自动转换为

0)之间可以进行比较。例如：

```
>>> 1 > 1.23        # 输出：False
>>> 2 > True        # 输出：True
>>> 123 >"abc"      # 报错.TypeError: '>' not supported between instances of 'int' and 'str'
```

Python 语言的关系和测试运算符如表 3-1 所示。

表 3-1 Python 语言的关系和测试运算符

运算符	表达式	含义	实例	结果
==	x== y	x 等于 y	"ABCDEF" == "ABCD"	False
!=	x!= y	x 不等于 y	"ABCD" != "abcd"	True
>	x> y	x 大于 y	"ABC" > "ABD"	False
>=	x>= y	x 大于等于 y	123 >= 23	True
<	x< y	x 小于 y	"ABC" < "上海"	True
<=	x<= y	x 小于等于 y	"123" <= "23"	True
is	x is y	x 和 y 是同一个对象	x=y=1; x is y x=1; y=2; x is y	True False
is not	x is not y	x 和 y 不是同一个对象	x=1; y=2; x is not y	True
in	x in y	x 是 y 的成员(y 是容器，例如元组)	1 in (1, 2, 3) "A" in "ABCDEF"	True True
not in	x not in y	x 不是 y 的成员(y 是容器，例如元组)	1 not in (1, 2, 3)	False

注意

(1) 关系运算符的优先级相同。

(2) 对于两个预定义的数值类型，关系运算符按照操作数的数值大小进行比较。

(3) 对于字符串类型，关系运算符比较字符串的值，即按字符的 ASCII 码值从左到右一一比较：首先比较两个字符串的第一个字符，其 ASCII 码值大的字符串大，若第一个字符相等，则继续比较第二个字符，以此类推，直至出现不同的字符为止。

3.3.4 逻辑运算符和逻辑表达式

逻辑运算符，即布尔运算符。用于检测两个以上条件的情况，即多个 bool 值的逻辑运算，其结果为 bool 类型值。

逻辑运算符除逻辑非(not)是一元运算符，其余均为二元运算符，用于将操作数进行逻辑运算，结果为 True 或 False。表 3-2 按优先级从高到低的顺序列出了 Python 中的逻辑运算符。

表 3-2 Python 中的逻辑运算符

运算符	含义	说明	优先级	实例	结果
not	逻辑非	当操作数为 False 时返回 True，当操作数为 True 时返回 False	1	not True not False	False True

续表

运算符	含义	说　明	优先级	实　例	结果
and	逻辑与	两个操作数均为 True 时，结果才为 True，否则为 False	2	True and True True and False False and True False and False	True False False False
or	逻辑或	两个操作数中有一个为 True 时，结果即为 True，否则为 False	3	True or True True or False False or True False or False	True True True False

注意

(1) Python 的任意表达式都可以评价为布尔逻辑值，故均可以参与逻辑运算。例如：

```
>>> not 0              #输出：True
>>> not 'a'            #输出：False
```

(2) C = A or B。如果 A 不为 0 或者不为空或者为 True，则返回 A；否则返回 B。仅在必要时才计算第二个操作数，即如果 A 不为 0 或者不为空或为 True，则不用计算 B。即“短路”计算。例如：

```
>>> 1 or 2             #输出：1
>>> 0 or 2             #输出：2
>>> False or True      #输出：True
>>> True or False      #输出：True
```

(3) C = A and B。如果 A 为 0 或者为空或者为 False，则返回 A；否则返回 B。仅在必要时才计算第二个操作数，即如果 A 为 0 或者为空或者为 False，则不用计算 B。即“短路”计算。例如：

```
>>> 1 and 2            #输出：2
>>> 0 and 2            #输出：0
>>> False and 2        #输出：False
>>> True and 2         #输出：2
```

这种写法常用于不确定 A 是否为空值时把 B 作为候补来赋值给 C。

3.3.5 单分支结构

if 语句单分支结构的语法形式如下：

```
if (条件表达式):
    语句/语句块
```

其中：

（1）条件表达式：可以是关系表达式、逻辑表达式、算术表达式等。

（2）语句/语句块：可以是单个语句，也可以是多个语句。多个语句的缩进必须对齐一致。

当条件表达式的值为真(True)时，执行 if 后的语句(块)，否则不做任何操作，控制将转到 if 语句的结束点。其流程如图 3-4(a)所示。

【例 3.7】 单分支结构示例(if_2desc.py)：输入两个整数 a 和 b，比较两者大小，使得 a 大于 b。

```
a = int(input("请输入第 1 个整数: "))
b = int(input("请输入第 2 个整数: "))
print(str.format("输入值: {0}, {1}", a, b))
if (a < b):       #a 和 b 交换
    t = a
    a = b
    b = t
print(str.format("降序值: {0}, {1}", a, b))
```

程序运行结果如下：

```
请输入第 1 个整数: 23
请输入第 2 个整数: 34
输入值: 23, 34
降序值: 34, 23
```

如果 a 和 b 不满足降序关系，本程序中还可以使用如下更简单的语句实现数据交换。

```
if (a < b): a,b = b,a
```

3.3.6 双分支结构

if 语句双分支结构的语法形式如下：

```
if (条件表达式):
    语句/语句块 1
else:
    语句/语句块 2
```

当条件表达式的值为真(True)时，执行 if 后的语句(块)1，否则执行 else 后的语句(块)2，其流程如图 3-4(b)所示。

Python 提供了下列条件表达式来实现等价于其他语言的三元条件运算符((条件)? 语

句 1:语句 2)的功能:

```
条件为真时的值 if (条件表达式) else 条件为假时的值
```

例如,如果 x>=0,则 y=x,否则 y=0,可以表述为:

```
y = x if (x >= 0) else 0
```

【例 3.8】　计算分段函数:$y=\begin{cases}\sin x+2\sqrt{x+e^4}-(x+1) & x\geqslant 0\\ \ln(-5x)-\dfrac{|x^2-8x|}{7x}+e & x<0\end{cases}$

此分段函数有以下几种实现方式,请读者自行编程测试。

(1) 利用单分支结构实现。

```
if (x >= 0):
    y = math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1, 3)
if (x < 0):
    y = math.log(-5 * x) - math.fabs(x * x - 8 * x) / (7 * x) + math.e
```

(2) 利用双分支结构实现。

```
if (x >= 0):
    y = math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1, 3)
else:
    y = math.log(-5 * x) - math.fabs(x * x - 8 * x) / (7 * x) + math.e
```

(3) 利用条件运算语句实现。

```
y = (math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1, 3))\
    if ((x >= 0)) else (math.log(-5 * x) - math.fabs(x * x - 8 * x) / (7 * x) + math.e)
```

3.3.7　多分支结构

if 语句多分支结构的语法形式如下:

```
if (条件表达式 1) :
    语句/语句块 1
elif (条件表达式 2) :
    语句/语句块 2
…
elif (条件表达式 n) :
```

```
    语句/语句块 n
[else:
    语句/语句块 n+1;]
```

该语句的作用是根据不同条件表达式的值确定执行哪个语句(块),其流程如图 3-4(c)所示。

【例 3.9】 已知某课程的百分制分数 mark,将其转换为五级制(优、良、中、及格、不及格)的评定等级 grade。评定条件如下:

$$\text{成绩等级}=\begin{cases}\text{优} & \text{mark}\geqslant 90\\ \text{良} & 80\leqslant \text{mark}<90\\ \text{中} & 70\leqslant \text{mark}<80\\ \text{及格} & 60\leqslant \text{mark}<70\\ \text{不及格} & \text{mark}<60\end{cases}$$

根据评定条件,有以下四种不同的方法实现。

方法一:

```
mark = int(input("请输入分数: "))
if (mark >= 90):
    grade = "优"
elif (mark >= 80):
    grade = "良"
elif (mark >= 70):
    grade = "中"
elif (mark >= 60):
    grade = "及格"
else:
    grade = "不及格"
```

方法二:

```
mark = int(input("请输入分数: "))
if (mark >= 90):
    grade = "优"
elif (mark >= 80 and mark < 90):
    grade = "良"
elif (mark >= 70 and mark < 80):
    grade = "中"
elif (mark >= 60 and mark < 70):
    grade = "及格"
else:
    grade = "不及格"
```

方法三:

```
mark = int(input("请输入分数: "))
if (mark >= 90):
    grade = "优"
elif (80 <= mark < 90):
    grade = "良"
elif (70 <= mark < 80):
    grade = "中"
elif (60 <= mark < 70):
    grade = "及格"
else:
    grade = "不及格"
```

方法四:

```
mark = int(input("请输入分数: "))
if (mark >= 60):
    grade = "及格"
elif (mark >= 70):
    grade = "中"
elif (mark >= 80):
    grade = "良"
elif (mark >= 90):
    grade = "优"
else:
    grade = "不及格"
```

其中,方法一中使用关系运算符“>=”,按分数从大到小依次比较;方法二和方法三使用关系运算符和逻辑运算符表达完整的条件,即使语句顺序不按分数从大到小依次书写,也可以得到正确的等级评定结果;方法四使用关系运算符“>=”,但按分数从小到大依次

比较。

上述四种方法中，方法一、方法二、方法三正确，而且方法一最简捷明了，方法二和方法三虽然正确，但是存在冗余条件；方法四虽然语法没有错误，但是判断结果错误：根据mark分数所得等级评定结果只有“及格”和“不及格”两种，请读者根据程序流程自行分析原因。

【例 3.10】　已知坐标点(x,y)，判断其所在的象限(if_coordinate.py)。

```
x = int(input("请输入 x 坐标: "))
y = int(input("请输入 y 坐标: "))
if (x == 0 and y == 0): print("位于原点")
elif (x == 0): print("位于 y 轴")
elif (y == 0): print("位于 x 轴")
elif (x > 0 and y > 0): print("位于第一象限")
elif (x < 0 and y > 0): print("位于第二象限")
elif (x < 0 and y < 0): print("位于第三象限")
else: print("位于第四象限")
```

程序运行结果如下：

```
请输入 x 坐标: 1
请输入 y 坐标: 2
位于第一象限
```

3.3.8　if 语句的嵌套

在 if 语句中又包含一个或多个 if 语句称为 if 语句的嵌套。一般形式如下：

```
if (条件表达式 1):
    if (条件表达式 11):
        语句 1                    }内嵌 if
    [else:                        }
        语句 2]                   }
[else:
    if (条件表达式 21):
        语句 3                    }内嵌 if
    [else:                        }
        语句 4]]                  }
```

【例 3.11】　计算分段函数：$y=\begin{cases}1 & x>0\\0 & x=0\\-1 & x<0\end{cases}$

此分段函数有以下几种实现方式，请读者判断哪些是正确的？并自行编程测试正确的实现方式。

方法一(多分支结构)：

```
if (x > 0): y = 1
elif (x == 0): y = 0
else: y = -1
```

方法二(if 语句嵌套结构)：

```
if (x >= 0):
    if (x > 0): y = 1
    else: y = 0
else: y = -1
```

方法三：

```
y = 1
if (x != 0):
    if (x < 0): y = -1
else: y = 0
```

方法四：

```
y = 1
if (x != 0):
    if (x < 0): y = -1
    else: y = 0
```

请读者画出每种方法相应的流程图，并进行分析测试。其中，方法一、方法二和方法三是正确的，而方法四是错误的。

3.3.9 if 语句典型示例代码

if 语句的典型示例代码如表 3-3 所示。当 if 或 else 的语句块仅包含一条语句时，该语句也可以直接写在关键字 if 或者 else 语句的同一行后面，以实现紧凑代码。

表 3-3 if 语句的典型示例代码

程序功能	代码片段
求绝对值	if a < 0: a = -a
a 和 b 按升序排序	if a > b: a,b = b,a
求 a 和 b 的最大值	if a > b: maximum = a else: maximum = b
计算两个数相除的余数，如果除数为 0，则给出报错信息	if b == 0:print("除数为 0") else:print("余数为: " + a % b)

续表

程 序 功 能	代 码 片 段

计算并输出一元二次方程的两个根。如果判别式 $b^2-4ac<0$，则显示“方程无实根”的提示信息	delta = b * b - 4.0 * a * c
	if delta < 0.0:
	　　print("方程无实根")
	else:
	　　d = math.sqrt(delta)
	　　print((-b + d)/(2.0 * a))
	　　print((-b - d)/(2.0 * a))

3.3.10　选择结构综合举例

【例 3.12】　输入三个整数，要求按从大到小的顺序排序(if_3desc.py)。

先 a 和 b 比较，使得 a＞b；然后 a 和 c 比较，使得 a＞c，此时 a 最大；最后 b 和 c 比较，使得 b＞c。

```
a = int(input("请输入整数 a: "))
b = int(input("请输入整数 b: "))
c = int(input("请输入整数 c: "))
if (a < b):a,b = b,a                    #使得 a>b
if (a < c):a,c = c,a                    #使得 a>c
if (b < c):b,c = c,b                    #使得 b>c
print("排序结果(降序): ", a, b, c)
```

程序运行结果如下：

```
请输入整数 a: 3
请输入整数 b: 2
请输入整数 c: 5
排序结果(降序): 5 3 2
```

【例 3.13】　编程判断某一年是否为闰年(leapyear.py)。判断闰年的条件是年份能被 4 整除但不能被 100 整除，或者能被 400 整除，其判断流程参见图 3-5 所示。

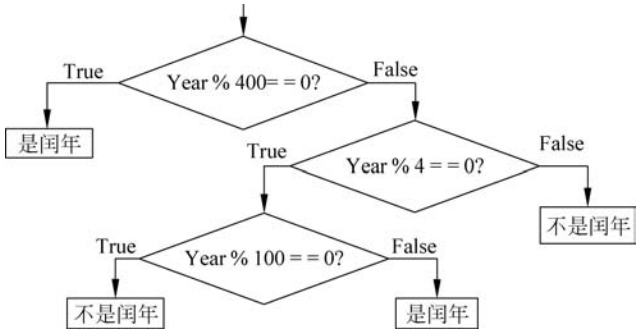


图 3-5　闰年的判断条件

方法一。使用一个逻辑表达式包含所有的闰年条件，相关语句如下：

```
if ((y % 4 == 0 and y % 100 != 0) or y % 400 == 0):
    print("是闰年")
else: print("不是闰年")
```

方法二。使用嵌套的 if 语句，相关语句如下：

```
if (y % 400 == 0): print("是闰年")
else:
    if (y % 4 == 0):
        if (y % 100 == 0): print("不是闰年")
        else: print("是闰年")
    else: print("不是闰年")
```

方法三。使用 if-elif 语句，相关语句如下：

```
if (y % 400 == 0): print("是闰年")
elif (y % 4 != 0): print("不是闰年")
elif (y % 100 == 0): print("不是闰年")
else: print("是闰年")
```

方法四。使用 calendar 模块的 isleap()函数来判断闰年，相关语句如下：

```
if (calendar.isleap(y)): print("是闰年")
else: print("不是闰年")
```

3.4 循环结构

循环结构用来重复执行一条或多条语句。使用循环结构，可以减少源程序重复书写的工作量。许多算法需要使用到循环结构。Python 使用 for 语句和 while 语句来实现循环结构。

3.4.1 可迭代对象（iterable）

可迭代对象一次返回一个元素，因而适用于循环。Python 包括以下几种可迭代对象：序列（sequence），例如字符串（str）、列表（list）、元组（tuple）等；字典（dict）；文件对象；迭代器对象（iterator）；生成器函数（generator）。

迭代器是一个对象，表示可迭代的数据集合，包括方法__iter__()和__next__()，可以实现迭代功能。

生成器是一个函数，使用 yield 语句，每次产生一个值，也可以用于循环迭代。

3.4.2 range 对象

Python 3 内置对象 range 是一个迭代器对象，迭代时产生指定范围的数字序列。其格式如下：

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

range 返回的数值序列从 start 开始，到 stop 结束(不包含 stop)。如果指定了可选的步长 step，则序列按步长 step 增长。例如：

```
>>> for i in range(1,11): print(i, end = ' ')        #输出：1 2 3 4 5 6 7 8 9 10
>>> for i in range(1,11,3): print(i, end = ' ')      #输出：1 4 7 10
```

Python 2 中 range 的类型为函数，是一个生成器；Python 3 中 range 的类型为类，是一个迭代器。

3.4.3 for 循环

for 语句用于遍历可迭代对象集合(例如列表、字典等)中的元素，并对集合中的每个元素执行一次相关的嵌入语句。当集合中的所有元素完成迭代后，控制传递给 for 之后的下一个语句。for 语句的格式如下：

```
for 变量 in 对象集合:
    循环体语句/语句块
```

【例 3.14】 利用 for 循环求 1～100 中所有奇数的和以及偶数的和(for_sum1_100.py)。

```
sum_odd = 0; sum_even = 0
for i in range(1, 101):
    if i % 2 != 0:          #奇数
        sum_odd += i        #奇数和
    else:                   #偶数
        sum_even += i       #偶数和
print("1～100 中所有奇数的和:", sum_odd)
print("1～100 中所有偶数的和:", sum_even)
```

程序运行结果如下：

```
1～100 中所有奇数的和: 2500
1～100 中所有偶数的和: 2550
```

3.4.4 while 循环

与 for 循环一样，while 也是一个预测试的循环，但是 while 在循环开始前，并不知道重

复执行循环语句序列的次数。while 语句按不同条件执行循环语句(块)零次或多次。while 循环语句的格式为：

```
while (条件表达式):
    循环体语句/语句块
```

while 循环的执行流程如图 3-6 所示。

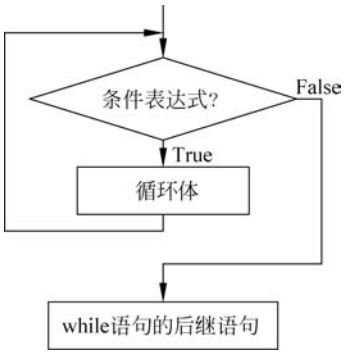


图 3-6　while 循环的执行流程

说明

(1) while 循环语句的执行过程如下：

① 计算条件表达式。

② 如果条件表达式结果为 True，控制将转到循环语句(块)，即进入循环体。当到达循环语句序列的结束点时，转①，即控制转到 while 语句的开始，继续循环。

③ 如果条件表达式结果为 False，退出 while 循环，即控制转到 while 循环语句的后继语句。

(2) 条件表达式是每次进入循环之前进行判断的条件，可以为关系表达式或逻辑表达式，其运算结果为 True(真)或 False(假)。条件表达式中必须包含控制循环的变量。

(3) 循环语句序列可以是一条语句，也可以是多条语句。

(4) 在循环语句序列中至少应包含改变循环条件的语句，以使循环趋于结束，避免“死循环”。

【例 3.15】　利用 while 循环求 $\sum_{i=1}^{100} i$，以及 1～100 中所有奇数的和及偶数的和(while_sum.py)。

```
i = 1; sum_all = 0; sum_odd = 0; sum_even = 0
while (i <= 100):
    sum_all += i          #所有数之和
    if (i % 2 == 0):      #偶数
        sum_even += i     #偶数和
    else:                 #奇数
        sum_odd += i      #奇数和
```

```
    i += 1
print("和 = %d、奇数和 = %d、偶数和 = %d" % (sum_all, sum_odd, sum_even))
```

程序运行结果如下：

```
和 = 5050、奇数和 = 2500、偶数和 = 2550
```

【例 3.16】　用如下近似公式求自然对数的底数 e 的值，直到最后一项的绝对值小于 10^{-6} 为止(while_e.py)。

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$$

```
i = 1; e = 1; t = 1
while (1/t >= pow(10, -6)):
    t *= i
    e += 1 / t
    i += 1
print("e =", e)
```

程序运行结果如下：

```
e = 2.7182818011463845
```

3.4.5　循环的嵌套

在一个循环体内又包含另一个完整的循环结构，则称之为循环的嵌套。这种语句结构称为多重循环结构。内层循环中还可以包含新的循环，形成多层循环结构。

在多层循环结构中，两种循环语句(for 循环、while 循环)可以相互嵌套。多重循环的循环次数等于每一重循环次数的乘积。

【例 3.17】　利用嵌套循环打印运行结果如图 3-7 所示的九九乘法表(nest_for.py)。

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

图 3-7　九九乘法表运行结果图

```
for i in range(1, 10):                  #外循环
    s = ""
    for j in range(1, 10):              #内循环
        s += str.format("{0:1}*{1:1}={2:<2} ", i, j, i * j)
    print(s)
```

思考：请修改程序，分别打印如图 3-8(a)和图 3-8(b)所示的九九乘法表。

```
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
```

(a) 下三角

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

(b) 上三角

图 3-8　九九乘法表的另外两种显示效果

3.4.6　break 语句

break 语句用于退出 for、while 循环，即提前结束循环，接着执行循环语句的后继语句。

注意　当多个 for、while 语句彼此嵌套时，break 语句只应用于最里层的语句，即 break 语句只能跳出最近的一层循环。

【例 3.18】　使用 break 语句终止循环(break.py)。

```
while True:
    s = input('请输入字符串(按 Q 或者 q 结束): ')
    if s.upper() == 'Q':
        break
    print('字符串的长度为: ', len(s))
```

程序运行结果如下：

```
请输入字符串(按 Q 或者 q 结束): Hello, World!
字符串的长度为: 13
请输入字符串(按 Q 或者 q 结束): 您好!
字符串的长度为: 3
请输入字符串(按 Q 或者 q 结束): q
```

【例 3.19】　编程判断所输入的任意一个正整数是否为素数(prime1.py 和 prime2.py)。

所谓素数(或称质数)，是指除了 1 和该数本身，不能被任何整数整除的正整数。判断一个正整数 m 是否为素数，只要判断 m 可否被 2～$\sqrt{m}$之中的任何一个整数整除，如果 m 不能被此范围中任何一个整数整除，m 即为素数，否则 m 为合数。

方法一(利用 for 循环和 break 语句):

```
import math
m = int(input("请输入一个整数(>1): "))
k = int(math.sqrt(m))
for i in range(2, k +2):
    if m % i == 0:
        break                          #可以整除,肯定不是素数,结束循环
if i == k+1 : print(m, "是素数!")
else: print(m, "是合数!")
```

方法二(利用 while 循环和 bool 变量):

```
import math
m = int(input("请输入一个整数(>1): "))
k = int(math.sqrt(m))
flag = True                            #先假设所输整数为素数
i = 2
while (i <= k and flag == True):
    if (m % i == 0): flag = False      #可以整除,肯定不是素数,结束循环
    else: i += 1
if (flag == True): print(m, "是素数!")
else: print(m, "是合数!")
```

3.4.7　continue 语句

continue 语句类似于 break,也必须在 for、while 循环中使用。但它结束本次循环,即跳过循环体内自 continue 下面尚未执行的语句,返回到循环的起始处,并根据循环条件判断是否执行下一次循环。

continue 语句与 break 语句的区别在于:continue 语句仅结束本次循环,并返回到循环的起始处,循环条件满足的话就开始执行下一次循环;而 break 语句则是结束循环,跳转到循环的后继语句执行。

与 break 语句相类似,当多个 for、while 语句彼此嵌套时,continue 语句只应用于最里层的语句。

【例 3.20】 使用 continue 语句跳过循环(continue_score.py)。要求输入若干学生成绩(按 Q 或 q 结束),如果成绩<0,则重新输入。统计学生人数和平均成绩。

```
num = 0; scores = 0;               #初始化学生人数和成绩总分
while True:
    s = input('请输入学生成绩(按 Q 或 q 结束): ')
    if s.upper() == 'Q':
        break
    if float(s) < 0:               #成绩必须>=0
        continue
    num += 1                       #统计学生人数
```

```
        scores += float(s)              #计算成绩总分
    print('学生人数为：{0},平均成绩为：{1}'.format(num,scores / num))
```

程序运行结果如下：

```
请输入学生成绩(按 Q 或 q 结束)：65
请输入学生成绩(按 Q 或 q 结束)：87
请输入学生成绩(按 Q 或 q 结束)：-40
请输入学生成绩(按 Q 或 q 结束)：q
学生人数为：2,平均成绩为：76.0
```

【例 3.21】 显示 100～200 之间不能被 3 整除的数(continue_div3.py)。要求一行显示 10 个数。程序运行结果如图 3-9 所示。

100~200之间不能被3整除的数为：

100	101	103	104	106	107	109	110	112	113
115	116	118	119	121	122	124	125	127	128
130	131	133	134	136	137	139	140	142	143
145	146	148	149	151	152	154	155	157	158
160	161	163	164	166	167	169	170	172	173
175	176	178	179	181	182	184	185	187	188
190	191	193	194	196	197	199	200		

图 3-9 显示 100～200 之间不能被 3 整除的数

```
j = 0                                   #控制一行显示的数值个数
print('100～200 之间不能被 3 整除的数为：')
for i in range(100, 200 + 1):
    if (i % 3 == 0): continue           #跳过能被 3 整除的数
    print(str.format("{0:<5}",i), end="") #每个数占 5 个位置,不足后面加空格,并且不换行
    j += 1
    if (j % 10 == 0): print()           #一行显示 10 个数后换行
```

3.4.8 死循环(无限循环)

如果 while 循环结构中循环控制条件一直为真，则循环将无限继续，程序将一直运行下去，从而形成死循环。

程序死循环时，会造成程序没有任何响应；或者造成不断输出(例如控制台输出、文件写入、打印输出等)。

在程序的循环体中插入调试输出语句 print，可以判断程序是否为死循环。

有的程序算法十分复杂，可能需要运行很长时间，但并不是死循环。

在大多数计算机系统中，用户可以使用 Ctrl+C 组合键中止当前程序的运行。

【例 3.22】 死循环示例(infinite.py)。

```
import math
while True:         #循环条件一直为真
   num = float(input("请输入一个正数:"))
```

```
        print(str(num), "的平方根为: ", math.sqrt(num))
    print("Good bye!")
```

本程序因为循环条件为“while True”,所以将一直重复提示用户输入一个正数,计算并输出该数的平方根,从而形成死循环。所以,最后一句“print("Good bye!")”语句将没有机会执行。

3.4.9　else 子句

for、while 语句可以附带一个 else 子句(可选)。如果 for、while 语句没有被 break 语句中止,则会执行 else 子句,否则不执行。其语法如下:

```
for 变量 in 对象集合:
    循环体语句(块)1
else:
    语句(块)2
```

或者:

```
while (条件表达式):
    循环体语句(块)1
else:
    语句(块)2
```

【例 3.23】　使用 for 语句的 else 子句(for_else.py)。

```
hobbies = ""
for i in range(1, 3 + 1):
    s = input('请输入爱好之一(最多三个,按 Q 或 q 结束): ')
    if s.upper() == 'Q':
        break
    hobbies += s + ''
else:
    print('您输入了三个爱好.')
print('您的爱好为: ', hobbies)
```

程序运行结果如下:

```
>>>
请输入爱好之一(最多三个,按 Q 或 q 结束): 旅游
请输入爱好之一(最多三个,按 Q 或 q 结束): 音乐
请输入爱好之一(最多三个,按 Q 或 q 结束): 运动
您输入了三个爱好.
您的爱好为:  旅游 音乐 运动
>>>
```

```
请输入爱好之一(最多三个,按 Q 或 q 结束): 音乐
请输入爱好之一(最多三个,按 Q 或 q 结束): q
您的爱好为:  音乐
```

3.4.10 循环语句典型示例代码

使用 for 语句和 while 语句都能实现循环功能，选择不同的语法构造取决于程序员的偏好。循环语句的典型示例如表 3-4 所示。

表 3-4 for 语句和 while 语句的典型示例

功能示例	实现代码
输出 n 个数(0～n－1)的 2 的乘幂的值列表	power = 1 for i in range(n): 　　print(str(i) + " " + str(power)) 　　power * = 2
输出小于或等于 n 的最大的 2 的乘幂的值	power = 1 while 2 * power <= n: 　　power * = 2 print(power)
计算并输出 1 ＋ 2 ＋…＋ n 的累积和	total = 0 for i in range(1, n+1): 　　total += i print(total)
计算并输出 n 的阶乘(n! ＝ 1 × 2 ×…× n)	factorial = 1 for i in range(1, n+1): 　　factorial * = i print(factorial)
输出半径为 1～n 的圆的周长列表	for r in range(1, n+1): 　　print("r = " + str(r), end = " ") 　　print("p = " + str(2.0 * math.pi * r))

3.4.11 循环结构综合举例

【例 3.24】 使用牛顿迭代法求解平方根(sqrt.py)。运行结果如图 3-10 所示。

```
请输入正实数a: 2
1.4142135623730951
```

图 3-10 使用牛顿迭代法求解平方根

计算一个正实数 a 的平方根可以使用牛顿迭代法实现：首先假设 t＝a，开始循环。如果 t＝a/t(或小于容差)，则 t 等于 a 的平方根，循环结束并返回结果；否则，将 t 和 a/t 的平均值赋值给 t，继续循环。

```
EPSILON = 1e-15                          #容差
a = float(input("请输入正实数 a: "))      #正实数 a
t = a                                    #假设平方根 t = a
while abs(t - a/t) > (EPSILON * t):
    t = (a/t + t) / 2.0                  #将 t 和 a/t 的平均值赋值给 t
print(t)                                 #输出 a 的平方根
```

【例 3.25】　显示 Fibonacci 数列(for_fibonacci.py)：1、1、2、3、5、8、…的前 20 项。

即
$$\begin{cases} F_1 = 1 & n = 1 \\ F_2 = 1 & n = 2 \\ F_n = F_{n-1} + F_{n-2} & n >= 3 \end{cases}$$

要求每行显示 4 项。运行结果如图 3-11 所示。

```
    1     1     2     3
    5     8    13    21
   34    55    89   144
  233   377   610   987
 1597  2584  4181  6765
```

图 3-11　显示 Fibonacci 数列

相关语句如下：

```
f1 = 1; f2 = 1
for i in range(1, 11):
    print(str.format("{0:6}{1:6}", f1, f2), end = " ")    #每次输出 2 个数,每个数占 6 位
                                                         #空格分隔
    if i % 2 == 0: print()                               #显示 4 项后换行
    f1 += f2; f2 += f1
```

3.5　错误和异常处理

3.5.1　程序的错误

Python 程序的错误通常可以分为三种类型，即语法错误、运行时错误和逻辑错误。

1. 语法错误

Python 程序的语法错误是指其源代码中拼写语法错误，这些错误导致 Python 编译器无法把 Python 源代码转换为字节码，故也称之为编译错误。程序中包含语法错误时，编译器将显示 SyntaxError 错误信息。

通过分析编译器抛出的运行时错误信息，仔细分析相关位置的代码，可以定位并修改程序错误。

【例 3.26】　Python 语法错误示例(syntax_error.py)。

```
print("Good Luck!")
print("你今天的幸运随机数是：", random.choice(range(10)))
```

程序运行结果如图 3-12 所示。

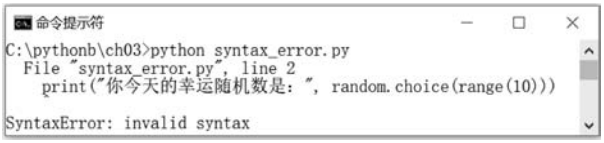


图 3-12　Python 语法错误运行示意图

编译器显示错误行号为 2，这是因为第一行的 print()函数需要结束括号，编译器编译到第二行时发现错误。一般情况下，需要根据提示错误行号和信息，在其附近判断和定位具体的错误。

2. 运行时错误

Python 程序的运行时错误是在解释执行过程中产生的错误。例如，如果程序中没有导入相关的模块(例如，import random)时，解释器将在运行时抛出 NameError 错误信息；如果程序中包括零除运算，解释器将在运行时抛出 ZeroDivisionError 错误信息；如果程序中试图打开不存在的文件，解释器将在运行时抛出 FileNotFoundError 错误信息。

通过分析解释器抛出的运行时错误信息，仔细分析相关位置的代码，可以定位并修改程序错误。

【例 3.27】　Python 运行时错误(没有导入相关的模块)示例(name_error.py)。

```
print("Good Luck!")
print("你今天的幸运随机数是：", random.choice(range(10)))
```

程序运行结果如图 3-13 所示。

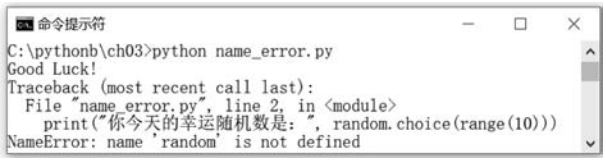


图 3-13　Python 运行时错误(没有导入相关的模块)运行示意图

编译器显示错误行号为 2，这是因为程序中没有导入相关模块的语句(import random)，编译器编译到第二行时发现错误。一般情况下，需要根据提示错误行号和信息，在其附近判断和定位具体的错误。

【例 3.28】　Python 运行时错误(零除错误)示例(zero_division_error.py)。

```
a = 1
b = 0
c = a/b
```

程序运行结果如图 3-14 所示。

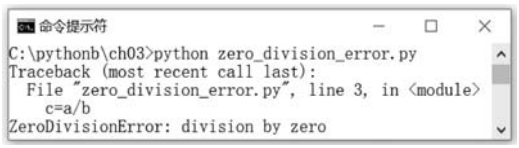


图 3-14　Python 运行时错误(零除错误)运行示意图

3. 逻辑错误

Python 程序的逻辑错误是程序可以执行(程序运行本身不报错),但运行结果不正确。对于逻辑错误,Python 解释器无能为力,需要读者根据结果来调试判断。

【例 3.29】　Python 逻辑错误示例(logic_error.py)。

```
import math
a = 1; b = 2; c = 1
x1 = -b + math.sqrt(b * b - 4 * a * c)/2 * a        #公式有误,故结果不正确
x2 = -b - math.sqrt(b * b - 4 * a * c)/2 * a        #公式有误,故结果不正确
print(x1, x2)                                       #输出: -2.0 -2.0
```

程序计算一元二次方程 $ax^2+bx+c=0$ 的两个根：$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$。方程 $x^2+2x+1=0$ 的正确解为 x1=x2=−1。但由于计算公式有误(正确公式为(−b + math.sqrt(b * b−4 * a * c))/(2 * a)以及(−b − math.sqrt(b * b−4 * a * c))/(2 * a)),故结果不正确。

3.5.2　异常处理概述

Python 语言采用结构化的异常处理机制。在程序运行过程中,如果产生错误,则抛出异常；通过 try 语句来定义代码块,以运行可能抛出异常的代码；通过 except 语句,可以捕获特定的异常并执行相应的处理；通过 finally 语句,可以保证即使产生异常(处理失败),也可以在事后清理资源等。例如,读取文件内容的伪代码一般如下：

```
def readfile():
    打开文件                          #可能产生错误: 文件不存在
    读取文件内容                      #可能产生错误: 无读取权限
    关闭文件
```

使用 Python 的结构化异常处理机制,其伪代码一般如下。

```
def read_file():
    try:
        打开文件                      #可能产生错误: 文件不存在
        读取文件内容                  #可能产生错误: 无读取权限
        关闭文件
```

```
except FileNotFoundError:          #捕获异常：无法打开文件
    #异常处理逻辑
except PermissionError:            #捕获异常：无读取权限
    #异常处理逻辑
```

从上面伪代码可以看出，异常处理机制可以把错误处理和正常代码逻辑分开，从而可以更加高效地实现错误处理，增加程序的可维护性。

异常处理机制已经成为许多现代程序设计语言处理错误的标准模式。

3.5.3 内置异常类和自定义异常类

在程序运行过程中，如果出现错误，Python 解释器会创建一个异常对象，并抛出给系统运行时(runtime)处理。即程序终止正常执行流程，转而执行异常处理流程。

在某种特殊条件下，代码中也可以创建一个异常对象，并通过 raise 语句，抛出给系统运行时处理。

异常对象是异常类的对象实例。Python 异常类均派生于 BaseException。常见的异常类包括 NameError、SyntaxError、AttributeError、TypeError 、ValueError、ZeroDivisionError、IndexError、KeyError 等。

在应用程序开发过程中，有时候需要定义特定于应用程序的异常类，表示应用程序的一些错误类型。

【例 3.30】 常见异常示例。

(1) NameError。尝试访问一个未申明的变量。

```
>>> noname        #报错。NameError: name 'noname' is not defined
```

(2) SyntaxError。语法错误。

```
>>> int a         #报错。SyntaxError: invalid syntax
```

(3) AttributeError。访问未知对象属性。

```
>>> a = 1
>>> a.show()      #报错。AttributeError: 'int' object has no attribute 'show'
```

(4) TypeError。类型错误。

```
>>> 11 + 'abc'   #报错。TypeError: unsupported operand type(s) for + : 'int' and 'str'
```

(5) ValueError。数值错误。

```
>>> int('abc')    #报错。ValueError: invalid literal for int() with base 10: 'abc'
```

（6）ZeroDivisionError。零除错误。

```
>>> 1/0          #报错。ZeroDivisionError: division by zero
```

（7）IndexError。索引超出范围。

```
>>> a = [10,11,12]
>>> a[3]        #报错。IndexError: list index out of range
```

（8）KeyError。字典关键字不存在。

```
>>> m = {'1':'yes', '2':'no'}
>>> m['3']      #报错。KeyError: '3'
```

3.5.4　引发异常

大部分由程序错误而产生的错误和异常，一般由 Python 虚拟机自动抛出。另外，在程序中，如果判断某种错误情况，则可以创建相应的异常类的对象，并通过 raise 语句抛出。

【例 3.31】　Python 虚拟机自动抛出异常示例。

```
>>> 1/0          #报错。ZeroDivisionError: division by zero
```

【例 3.32】　程序代码中通过 raise 语句抛出异常示例。

```
>>> if a<0: raise ValueError("数值不能为负数")
```

如果 a 小于 0，则程序运行后将显示“ValueError：数值不能为负数”的报错信息。

3.5.5　捕获和处理异常

当程序中的引发异常后，Python 虚拟机通过调用堆栈查找相应的异常捕获程序。如果找到匹配的异常捕获程序（即调用堆栈中某函数使用 try…except 语句捕获处理），则执行相应的处理程序（try…except 语句中匹配的 except 语句块）。

如果堆栈中没有匹配的异常捕获程序，则该异常最后会传递给 Python 虚拟机，Python 虚拟机通用异常处理程序在控制台打印出异常的错误信息和调用堆栈，并中止程序的执行。

【例 3.33】　Python 虚拟机捕获处理异常示例（pymexcept.py）。

```
i1 = 1;i2 = 0
print(i1/i2)
```

程序运行后将显示“ZeroDivisionError：division by zero”报错信息。

Python 语言采用结构化的异常处理机制。try 语句定义代码块，运行可能抛出异常的

代码；except 语句捕获特定的异常并执行相应的处理；else 语句执行无异常时的处理；finally 语句保证即使产生异常（处理失败），也可以在事后清理资源等。try…except…else…finally 语句的一般格式为：

```
try:
    可能产生异常的语句
except Exception1:                  #捕获异常 Exception1
    发生异常时执行的语句
except (Exception2, Exception3):    #捕获异常 Exception2、Exception3
    发生异常时执行的语句
except Exception4 as e:             #捕获异常 Exception4,其实例为 e
    发生异常时执行的语句
except:                             #捕获其他所有异常
    发生异常时执行的语句
else:                               #无异常
    无异常时执行的语句
finally:                            #不管发生异常与否,保证执行
    不管发生异常与否,保证执行的语句
```

except 块可以捕获并处理特定的异常类型（此类型称为“异常筛选器”），具有不同异常筛选器的多个 except 块可以串联在一起。系统自动自上而下匹配引发的异常：如果匹配（引发的异常为“异常筛选器”的类型或子类型），则执行该 except 块中的异常处理代码；否则继续匹配下一个 except 块，故用户需要将带有最具体的（即派生程度最高的）异常类的 except 块放在最前面。

【例 3.34】 try…except…else…finally 示例（try_except.py）。

```
try:
    f = open("testfile.txt", "w")
    f.write("这是一个测试文件,用于测试异常!!")
    f1 = open("testfile1.txt", "r")        #报错：没有找到文件或读取文件失败
except IOError:
    print("没有找到文件或读取文件失败")
else:
    print("文件写入成功!")
finally:
    f.close()
```

3.6 综合应用：turtle 模块的复杂图形绘制

3.6.1 绘制正方形（改进版）

【例 3.35】 修改例 2.39 的代码，使用循环结构绘制正方形。

```
import turtle                  #导入 turtle 模块
p = turtle.Turtle()            #创建海龟对象
p.color("red")                 #设置绘制时画笔的颜色
p.pensize(3)                   #定义绘制时画笔的线条宽度
turtle.speed(1)                #定义绘图的速度("slowest"或者 1)
p.goto(0,0)                    #移动海龟到坐标原点(0,0)
for i in range(4):             #绘制正方形的四条边
    p.forward(100)             #向前移动 100
    p.right(90)                #向右旋转 90 度
```

3.6.2　绘制圆形螺旋

【例 3.36】　使用海龟绘图分别绘制红、蓝、绿、黄四种颜色的圆形螺旋(spiral.py)。运行最终结果如图 3-15 所示。

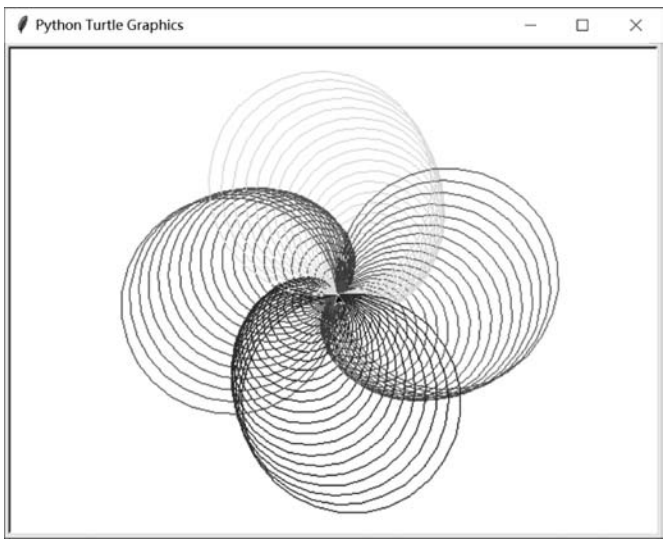
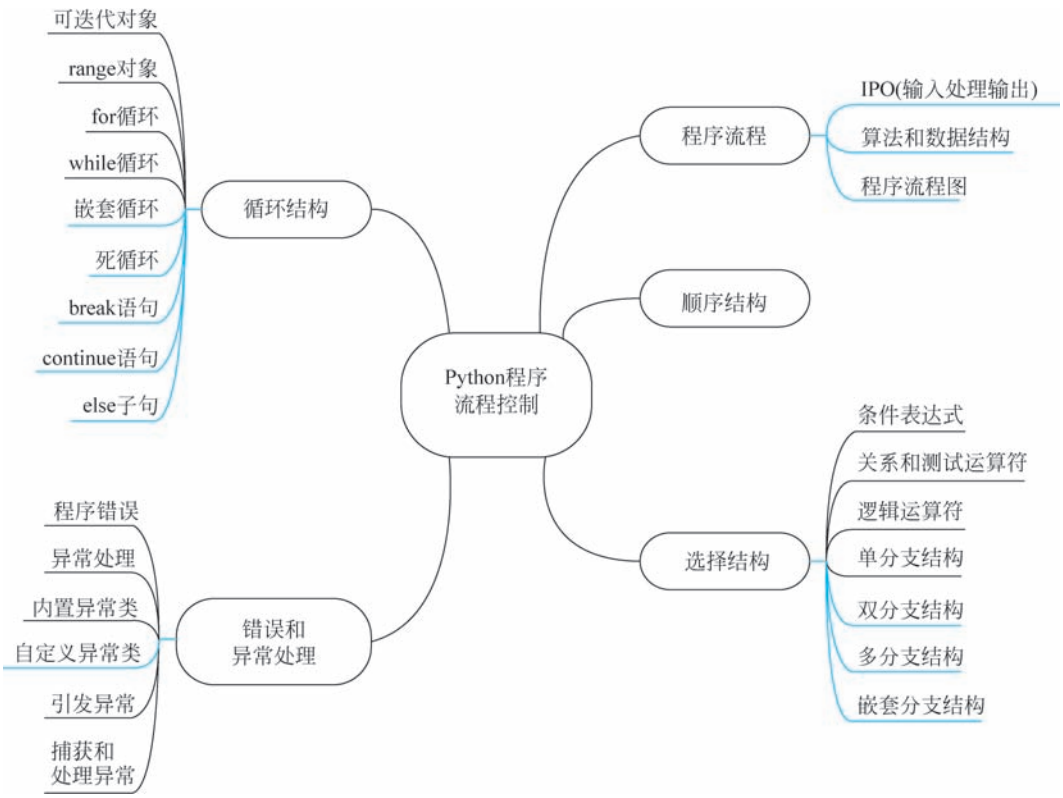


图 3-15　使用海龟绘图分别绘制四种颜色的圆形螺旋

```
import turtle                          #导入 turtle 模块
p = turtle.Turtle()                    #创建海龟对象
p.speed(0)                             #定义绘图的速度("fastest"或者 0 均表示最快)
colors = ["red", "blue", "green", "yellow"] #红、蓝、绿、黄四种颜色
for i in range(100):                   #i = 0～99
    p.pencolor(colors[i % 4])          #设置画笔颜色(红或蓝或绿或黄)
    p.circle(i)                        #画圆
    p.left(91)                         #向左旋转 91 度
```

本章小结


Python程序流程控制
程序流程
IPO(输入处理输出)
算法和数据结构
程序流程图
顺序结构
选择结构
条件表达式
关系和测试运算符
逻辑运算符
单分支结构
双分支结构
多分支结构
嵌套分支结构
循环结构
可迭代对象
range对象
for循环
while循环
嵌套循环
死循环
break语句
continue语句
else子句
错误和异常处理
程序错误
异常处理
内置异常类
自定义异常类
引发异常
捕获和处理异常