3

# 程序结构与控制流

本章将详细介绍程序结构和控制流，主题包括条件、循环、异常与上下文管理器。

## 3.1 程序结构与执行

Python程序由一系列语句构成。所有语言特性，包括变量赋值、表达式、函数定义、类和模块导入，都是语句，并与其他语句具有同等地位——这意味着任何语句几乎都可以放在程序中的任何位置（尽管某些语句如return只能出现在函数中）。例如，下面的代码在条件语句中定义了两个不同版本的函数：

if debug:  
 def square(x):  
 if not isinstance(x,float):  
 raise TypeError('Expected a float')  
 return x \* x  
else:  
 def square(x):  
 return x \* x

当加载源文件时，解释器按照语句出现的顺序来执行语句，直到没有语句可执行为止。这个执行模型既适用于作为主程序运行的文件，也适用于通过import加载的库文件。

## 3.2 条件执行

if、else和elif语句控制条件代码的执行。条件语句的一般格式如下：

if expression:  
 statements  
elif expression:  
 statements  
elif expression:  
 statements  
...  
else:  
 statements

如果不需要，则可以省略条件语句的else和elif子句。如果某个子句里没有语句，则使用pass语句来表示：

if expression:  
 pass # To do: 请实现  
else:  
 statements

## 3.3 循环与迭代

使用for和while语句实现循环，如下所示：

while expression:  
 statements  
  
for i in s:  
 statements

while语句会一直执行语句，直到相关表达式的计算结果为False。for语句遍历s中的所有元素，直到没有可用的元素。for语句适用于任何支持迭代的对象。这包括内置的序列类型，如列表、元组和字符串，也包括任何实现迭代器协议的对象。

在for i in s语句中，变量i被称作迭代变量。在循环的每次迭代中，它从s接收一个新值。迭代变量的作用域不是for语句私有的。如果之前定义的变量具有相同的名称，则该值将被覆盖。此外，迭代变量在循环完成后保留最后一个值。

如果迭代生成的元素是相同尺寸的迭代对象，则可以使用如下语句将该迭代对象解构为单独的迭代变量：

s = [ (1, 2, 3), (4, 5, 6) ]  
  
for x, y, z in s:  
 statements

在这个例子中，s必须包含或生成迭代对象，每个迭代对象都有3个元素。在每次迭代中，变量x、y和z被赋值为对应迭代对象的项目。虽然这种情况在s是一个元组序列时最常见，但当s中的项目是任何类型的迭代对象（包括列表、生成器和字符串）时，解构也可以工作。

有时会在解构时使用像\_这样的可任意丢弃的变量。例如：

for x, \_, z in s:  
 statements

这里，一个值仍然被放置在\_变量中，但是变量的名称意味着它在后面的语句中没有意义或者没用。

如果迭代对象的项目的尺寸不统一，可以使用通配符的解构方式将多个值放在一个变量中。例如：

s = [ (1, 2), (3, 4, 5), (6, 7, 8, 9) ]  
  
for x, y, \*extra in s:  
 statements # x = 1, y = 2, extra = []  
 # x = 3, y = 4, extra = [5]  
 # x = 6, y = 7, extra = [8, 9]  
 # ...

这个例子至少需要两个值x和y。而\*extra接收任何可能出现的额外值，这些值总是放在一个列表中。在一个解构中最多只能出现一个带星号的变量，但该变量可以出现在任何位置。所以，这两种变体都是合法的：

for \*first, x, y in s:  
 ...  
  
for x, \*middle, y in s:  
 ...

在循环时，除跟踪数据值外，有时跟踪数值索引也是很有用的。这里有一个例子：

i = 0  
for x in s:  
 statements  
 i += 1

Python提供了一个内置函数enumerate()，可以用来简化这段代码：

for i, x in enumerate(s):  
 statements

enumerate(s)创建一个迭代器，生成元组(0,s[0])、(1, s[1])、(2, s[2])，以此类推。可以使用enumerate()的start关键字参数为计数提供一个不同的起始值：

for i, x in enumerate(s, start=100):  
 statements

这将生成形式诸如(100, s[0])、(101, s[1])的元组。

另一个常见的循环问题是并行迭代两个或多个迭代对象——例如，编写一个循环，每次迭代都从不同的序列中获取元素：

# s与t是两个序列  
i = 0  
while i < len(s) and i < len(t):  
 x = s[i] # 从s中获取一个项目  
 y = t[i] # 从t中获取一个项目  
 statements  
 i += 1

这段代码可以使用zip()函数来简化。例如：

# s与t是两个序列  
for x, y in zip(s, t):  
 statements

zip(s, t)将迭代对象s和t组合成一个由(s[0], t[0])、(s[1], t[1])、(s[2], t[2])这样的元组形式所构成的迭代对象。如果s和t的长度不相等，zip()的行为在较短序列的元素消耗光时停止。zip()的结果是一个迭代器，用于迭代时生成结果。如果希望将结果转换为列表，请使用list(zip(s, t))。

要跳出循环，请使用break语句。例如，这段代码从文件中读取文本行，直到遇到空行：

with open('foo.txt') as file:  
 for line in file:  
 stripped = line.strip()  
 if not stripped:  
 break # 遇到空行，停止读取  
 # 处理非空行  
 ...

要跳转到循环的下一个迭代（跳过循环体的剩余部分），请使用continue语句。在放弃一个条件测试分支，以及缩进造成程序嵌套太深或不必要的复杂时，这个语句很有用。例如，下面的循环会跳过文件中的所有空行：

with open('foo.txt') as file:  
 for line in file:  
 stripped = line.strip()  
 if not stripped:  
 continue # 若遇到空行，则跳过循环体的剩余代码，开始循环的下一次迭代  
 # 处理非空行  
 ...

break和continue语句只适用于正在执行的最内层循环。如果需要跳出深度嵌套的多层循环结构，可以考虑异常处理方式。Python没有提供“goto”语句。还可以将else语句附加到循环结构中，如下例所示：

# for-else  
with open('foo.txt') as file:  
 for line in file:  
 stripped = line.strip()  
 if not stripped:  
 break  
 # 处理非空行  
 ...  
 else:  
 raise RuntimeError('Missing section separator')

循环的else子句只在循环运行结束时才执行。这要么立即发生（如果循环根本不执行），要么在最后一次迭代后发生。如果使用break语句提前中止循环，则会跳过else子句。

循环else子句的主要应用场景是，在遍历数据时，为了应对循环过早中断，需要设置或者检查某种标志或条件的情况。例如，如果不使用else子句，前面的代码可能需要借助一个标志变量来重写，如下所示：

found\_separator = False  
  
with open('foo.txt') as file:  
 for line in file:  
 stripped = line.strip()  
 if not stripped:  
 found\_separator = True  
 break  
 # 处理非空行  
 ...  
 if not found\_separator:  
 raise RuntimeError('Missing section separator')

## 3.4 异常

异常代表错误和程序正常控制流的中断。通过raise语句来引发异常。raise语句的一般格式是raise Exception([value])。其中，Exception是异常类型；value是一个可选值，给出关于异常的特定细节。这里有一个例子：

raise RuntimeError('Unrecoverable Error')

要捕获异常，请使用try和except语句，如下所示：

try:  
 file = open('foo.txt', 'rt')  
except FileNotFoundError as e:  
 statements

当异常发生时，解释器停止执行try块中的语句，并寻找与所发生异常类型匹配的except子句。如果找到，控制权就传递给except子句中的第一个语句。完成except子句的执行后，继续执行整个try-except块之后出现的第一个语句。

try语句没有必要匹配所有可能发生的异常。如果没有找到匹配的except子句，异常将继续传播，并可能由另一个在别处定义的可以处理该异常的try-except块捕获。作为良好的编程风格，应该只捕获可以恢复的异常。如果异常无法恢复，通常最好让异常往上传播。

如果一个异常一直传播到程序的顶层都没有被捕获，解释器就会带着一条错误消息终止程序。

如果调用raise语句，则会再次抛出最后生成的异常。这种情况只在处理之前抛出的异常时有效。例如：

try:  
 file = open('foo.txt', 'rt')  
except FileNotFoundError:  
 print("Well, that didn't work.")  
 raise # 重新抛出当前异常

每个except子句都可以与as var修饰符一起使用，该修饰符给出一个变量的名称，如果发生异常，则将异常类型的实例放入该变量中。异常处理程序可以检查此变量的值，以了解有关异常原因的更多信息。例如，可以使用isinstance()来检查异常类型。

异常有一些标准特性，在需要执行进一步操作以响应错误时可能很有用。

e.args  
异常抛出时提供的参数元组。在大多数情况下，这是一个包含错误描述的字符串的单项目元组。对于OSError异常，该值为二元组或三元组，包含整型错误号、错误消息字符串和可选的文件名。

e.\_\_cause\_\_  
如果本异常（B）是为了响应处理另一个异常（A）而特意抛出的，则值为之前的那个异常（A）。请参阅后面关于链式异常的内容。

e.\_\_context\_\_  
如果在处理另一个异常（A）时抛出了本异常（B），则值为之前的那个异常（A）。

e.\_\_traceback\_\_  
与本异常相关的栈回溯对象。

用来保存异常值的变量只能在关联的except块中访问。一旦控制权离开块，变量就变为了未定义的。例如：

try:  
 int('N/A') # 抛出ValueError  
except ValueError as e:  
 print('Failed:', e)  
  
print(e) # 失败信息 -> NameError. 'e' not defined.

可以使用多个except子句指定多个异常处理块：

try:  
 do something  
except TypeError as e:  
 # 处理Type error  
 ...  
except ValueError as e:  
 # 处理Value error  
 ...

一个处理子句可以捕获多种异常类型，如下所示：

try:  
 do something  
except (TypeError, ValueError) as e:  
 # 处理Type error或Value error  
 ...

要忽略异常，请使用pass语句，如下所示：

try:  
 do something  
except ValueError:  
 pass # 什么都不做

无视错误通常是危险的，这也是难以发现代码问题的根源之一。即使要忽略错误，明智的做法通常也是在日志或其他地方报告该错误，以便日后检查。

要捕获与程序退出相关异常之外的所有异常，请像这样使用Exception：

try:  
 do something  
except Exception as e:  
 print(f'An error occurred : {e!r}')

在捕获所有异常时，应该非常小心地向用户报告准确的错误信息。例如，在前面的代码中，将打印一条错误消息和相关的异常值。如果不包含任何异常值信息，那么调试因莫名原因而失败的代码就会变得非常困难。

try语句还支持else子句，它必须跟在最后一个except子句后面。如果try块中的代码没有引发异常，则执行此代码。这里有一个例子：

try:  
 file = open('foo.txt', 'rt')  
except FileNotFoundError as e:  
 print(f'Unable to open foo : {e}')  
 data = ''  
else:  
 data = file.read()  
 file.close()

finally语句定义了一个无论try-except块中发生什么都必须执行的清理操作。举个例子：

file = open('foo.txt', 'rt')  
try:  
 # 一些处理  
 ...  
finally:  
 file.close()  
 # 无论发生了什么，都要关闭文件

finally子句不是用来捕获错误的。相反，它用于定义无论是否发生错误，都必须执行的代码。如果没有引发异常，finally子句中的代码将在try块中的代码之后立即执行。如果发生异常，首先执行匹配的except块（如果有的话），然后将控制权传递给finally子句的第一个语句。如果finally子句中的代码执行完毕后，异常仍处于挂起状态，则该异常将被重新抛出，以便由另一个异常处理程序捕获。

### 3.4.1 异常层次

处理异常的一项挑战是管理程序中可能发生的大量异常。例如，仅内置异常就有60多个。再考虑到标准库的其他部分，异常可能会增加到数百个。此外，通常无法事先轻松确定代码随处可能引发的异常类型。异常不会被记录为函数调用签名的一部分，也没有哪种编译器可以验证代码中的正确异常处理。因此，异常处理有时会让人感觉杂乱无章。

针对这种挑战，应该认识到Python的异常是一种通过继承组织而成的层次结构。相对于考虑特定错误，关注更通用的错误类别可能会让事情变得更容易。现在来看看在容器中查找值时可能出现的不同错误：

try:  
 item = items[index]  
except IndexError: # 如果items是一个序列，将抛出此异常  
 ...  
except KeyError: # 如果items是一个映射，将抛出此异常  
 ...

与其编写这么多的代码处理两个如此具体的异常，不如这么写来得简单：

try:  
 item = items[index]  
except LookupError:  
 ...

LookupError是一个表示异常的更高级别分组。IndexError和KeyError都继承自LookupError，所以这个except子句将捕获其中的任何一个问题。然而，LookupError的范围并不涉及与查找无关的错误。

表3.1描述了最常见的内置异常类别。

BaseException类很少直接用于异常处理，因为它匹配所有可能的异常，包括影响控制流的特殊异常，比如SystemExit、KeyboardInterrupt和StopIteration。通常很少需要捕获这些异常。相反，所有与程序相关的普通错误都继承自Exception。ArithmeticError是所有与数学相关的错误的基础类型，比如ZeroDivisionError、FloatingPointError和OverflowError。ImportError是所有与导入相关的错误的基础类型。LookupError是所有与容器查找相关的错误的基础类型。OSError是源自操作系统和环境的所有错误的基础类型，它包含与文件、网络连接、权限、管道、超时等相关的各种异常。ValueError异常通常在为操作提供错误输入值时抛出。UnicodeError是ValueError的子类，对所有与Unicode相关的编码和解码错误进行分组。

表3.1 异常类别

|  |  |
| --- | --- |
| 异常类型 | 描 述 |
| BaseException | 所有异常的根类 |
| Exception | 所有与程序相关的错误的基础类型 |
| ArithmeticError | 所有与数学相关的错误的基础类型 |
| ImportError | 所有与导入相关的错误的基础类型 |
| LookupError | 所有与容器查找相关的错误的基础类型 |
| OSError | 所有与系统相关的错误的基础类型。IOError和EnvironmentError是别名 |
| ValueError | 与值相关的错误的基础类型，包括Unicode |
| UnicodeError | 与Unicode字符串编码相关的错误的基础类型 |

表3.2展示了一些常见的内置异常，它们直接继承自Exception，而非其他的异常分组。

表3.2 其他内置异常

|  |  |
| --- | --- |
| 异常类型 | 描 述 |
| AssertionError | 失败的断言语句 |
| AttributeError | 错误地查找对象的特性 |
| EOFError | 文件结束错误 |
| MemoryError | 可恢复的内存不足错误 |
| NameError | 在本地或全局命名空间中找不到名称 |
| NotImplementedError | 未实现的功能 |
| RuntimeError | 通用的“something bad happened”错误 |
| TypeError | 应用于错误类型对象的操作 |
| UnboundLocalError | 在赋值之前使用了局部变量 |

### 3.4.2 异常与控制流

通常，异常是为处理错误而准备的。不过，也有一些异常用于更改控制流。这些异常如表3.3所示，直接继承自BaseException。

表3.3 用于控制流的异常

|  |  |
| --- | --- |
| 异常类型 | 描 述 |
| SystemExit | 抛出，以指示程序退出 |
| KeyboardInterrupt | 当程序通过Ctrl-C组合键中断时抛出 |
| StopIteration | 抛出，以表示迭代的结束 |

SystemExit异常用于让程序故意终止。可以提供整数退出码或字符串消息作为参数。如果提供一个字符串，字符串将打印到sys.stderr，同时程序以退出码1终止。下面是一个典型的例子：

import sys  
if len(sys.argv) != 2:  
 raise SystemExit(f'Usage: {sys.argv[0]} filename)  
  
filename = sys.argv[1]

当程序接收到SIGINT信号（通常通过在终端中按下Ctrl-C组合键进行）时，会抛出KeyboardInterrupt异常。这个异常有点不寻常，因为它是异步的——这意味着它几乎可以在程序中的任何时间和任何语句上发生。在这种情况发生时，Python的缺省行为是，简单地终止程序的运行。如果想控制SIGINT的传递，可以使用signal库模块（参见第9章）。

StopIteration异常是迭代协议的一部分，表示迭代结束。

### 3.4.3 定义新异常

所有内置异常都是通过类型来定义的。要创建一个新异常，可创建一个继承自Exception的新类定义，如下所示：

class NetworkError(Exception):  
 pass

像这样通过raise语句来使用我们的新异常：

raise NetworkError('Cannot find host')

当抛出异常时，随raise语句提供的可选值被用作异常的类构造函数的参数。在大多数情况下，这是一个包含某种错误消息的字符串。然而，用户定义的异常可以写入一个或多个异常值，如下例所示：

class DeviceError(Exception):  
 def \_\_init\_\_(self, errno, msg):  
 self.args = (errno, msg)  
 self.errno = errno  
 self.errmsg = msg  
  
# 抛出一个异常（多个参数）  
raise DeviceError(1, 'Not Responding')

当创建一个重新定义了\_\_init\_\_()的自定义异常类时，如上述代码所示，将包含了\_\_init\_\_()参数的元组分配给特性self.args是很重要的。self.args特性在打印异常回溯消息时使用。如果将self.args保留为未定义，则发生错误时用户无法看到有关异常的任何有用信息。

可以使用继承将异常组织成一个层次结构。例如，前面定义的NetworkError异常可以作为各种更具体错误的基类。这里有一个例子：

class HostnameError(NetworkError):  
 pass  
  
class TimeoutError(NetworkError):  
 pass  
  
def error1():  
 raise HostnameError('Unknown host')  
  
def error2():  
 raise TimeoutError('Timed out')  
  
try:  
 error1()  
except NetworkError as e:  
 if type(e) is HostnameError:  
 # 对这类错误执行特定操作  
 ...

在本例中，except NetworkError子句捕获派生自NetworkError的任何异常。要找出所抛出错误的特定类型，请使用type()检查执行值的类型。

### 3.4.4 链式异常

有时，为了响应一个异常，我们可能希望抛出一个不同的异常。要达成此目标，可以抛出一个链式异常：

class ApplicationError(Exception):  
 pass  
  
def do\_something():  
 x = int('N/A') # 抛出ValueError异常  
  
def spam():  
 try:  
 do\_something()  
 except Exception as e:  
 raise ApplicationError('It failed') from e

如果发生未捕获的ApplicationError，我们将得到包含这两个异常的消息。例如：

>>> spam()  
Traceback (most recent call last):  
 File "c.py", line 9, in spam  
 do\_something()  
 File "c.py", line 5, in do\_something  
 x = int('N/A')  
ValueError: invalid literal for int() with base 10: 'N/A'  
  
The above exception was the direct cause of the following exception:  
  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
 File "c.py", line 11, in spam  
 raise ApplicationError('It failed') from e  
\_\_main\_\_.ApplicationError: It failed  
>>>

如果捕获一个ApplicationError，所产生异常的\_\_cause\_\_特性将包含之前那个异常（本例中的ValueError异常）。例如：

try:  
 spam()  
except ApplicationError as e:  
 print('It failed. Reason:', e.\_\_cause\_\_)

如果想在不包含其他异常链的情况下抛出一个新的异常，请像这样从None中抛出一个错误：

def spam():  
 try:  
 do\_something()  
 except Exception as e:  
 raise ApplicationError('It failed') from None

出现在except块中的编码错误也会导致链式异常，但这种情况下工作方式略有不同。例如，假设我们有一些这样的错误代码：

def spam():  
 try:  
 do\_something()  
 except Exception as e:  
 print('It failed:', err) # err未定义（打字错误）

产生的异常回溯消息就略有不同：

>>> spam()  
Traceback (most recent call last):  
 File "d.py", line 9, in spam  
 do\_something()  
 File "d.py", line 5, in do\_something  
 x = int('N/A')  
ValueError: invalid literal for int() with base 10: 'N/A'  
  
During handling of the above exception, another exception occurred:  
  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
 File "d.py", line 11, in spam  
 print('It failed. Reason:', err)  
NameError: name 'err' is not defined  
>>>

如果在处理另一个异常（上例中的ValueError异常）时抛出未预期的异常（上例中的NameError异常），则\_\_context\_\_特性（而非\_\_cause\_\_）保存错误发生时正在处理的异常（ValueError异常）的信息。例如：

try:  
 spam()  
except Exception as e:  
 print('It failed. Reason:', e)  
 if e.\_\_context\_\_:  
 print('While handling:', e.\_\_context\_\_)

在异常链中，预期异常和意外异常之间有一个重要的区别。在第一个示例中，编写代码是为了预料到出现异常的可能性。例如，代码被显式包装在一个try-except块中：

try:  
 do\_something()  
except Exception as e:  
 raise ApplicationError('It failed') from e

在第二种情况下，在except块中有一个编码错误：

try:  
 do\_something()  
except Exception as e:  
 print('It failed:', err) # err未定义

这两种情况之间的区别很微妙，也很重要。这种区别就是异常链信息为什么放在\_\_cause\_\_或\_\_context\_\_特性中的原因。\_\_cause\_\_特性在预料到失败可能性时使用。\_\_context\_\_特性在这两种情况下都可以使用，但\_\_context\_\_是在处理一个异常过程中抛出的意外异常的唯一信息回溯源。

### 3.4.5 异常回溯

异常有一个对应的栈回溯结构，提供错误发生的位置信息。回溯结构存储在异常的\_\_traceback\_\_特性中。出于报告或调试的目的，我们可能会希望自己生成回溯消息。可以使用traceback模块来完成这一任务。例如：

import traceback  
  
try:  
 spam()  
except Exception as e:  
 tblines = traceback.format\_exception(type(e), e, e.\_\_traceback\_\_)  
 tbmsg = ''.join(tblines)  
 print('It failed:')  
 print(tbmsg)

在这段代码中，format\_exception()返回一个字符串列表，其中包含通常的异常回溯消息。参数提供了异常类型、异常值和回溯结构。