第 1 2 章

错误和异常

现实生活并不是一帆风顺的，总会遇到各种突发情况。例如，飞机延误、火车晚点，公交 车堵车等，这些情况会引起我们约会赶不上、会议错过、上班迟到。在程序中也会遇到各种各 样的问题，比如，访问一个格式损坏了的文件、连接一个断开的网络、将字符串赋值给一个整 形变量等。此时,Python会检测到程序出现错误,无法继续执行。

因此，我们需要一个专门处理错误的机制，在程序中对可能出现的错误进行处理，有些错 误可以在程序中进行修复，不能修复的错误也可以提供详细具体的诊断信息，帮助程序员尽快 解决问题，恢复程序的运行。

简命

程序中的错误分为语法错误和逻辑错误。语法错误是指软件的编写不符合Python语言的 语法规定，导致无法被解释器解释或者编译器编译。这些错误必须经过修正，程序才能运行。

例如：

»> if 2 > 1

File n<stdin>"r line 1

if 2 > 1

SyntaxError: invalid syntax

上述代码在判断语句后少了冒号（:）,不符合Python的语法,因此语法分析器检测到错误后, 显示错误信息。错误信息包含了错误的行号，错误名称和具体信息，错误信息还用小箭头*（»* 指出语法错误的具体位置，方便程序员定位和改正。语法错误的类型是Synt宓

程序运行以后出现的错误就是逻辑错误，逻辑错误可能是由于外界条件引起的（比如网络 断开、文件格式损坏、输入的字符串格式不正确等），也可能是程序本身设计不严谨导致的， 比如用0当作除数。

综上所述，不管是那种错误，只要被Python检测到，程序都会发生异常。

12JJ異囂

在程序运行期间检测到的错误称之为异常。如果异常不被处理，默认的方式是程序崩溃， 并给出错误的提示。比如下列代码：

>>> open ("123 o txt°)

Traceback (most recent call last):

File "<stdin>n z line 1in <module>

FileNotFoundError: [Errno 2] No such file or directory: v123.txt \*

上述代码试图打开一个名为"23.txt”的文件，但是该文件并不存在,因此出现异常,程 序终止。解释器会发送错误提示信息，其中第3行可以看到岀现错误发生的行号和模块，第4 行显示了错误的名称FileNotFoundError,以及错误编号和详细信息，表示找不到名称为“123. txt"的文件或者目录。

程序在运行时崩溃是很糟糕的用户体验，所以要在程序中尽量对异常进行处理，能够解决 的异常要在程序中解决，不能解决的也要用友好的方式提示用户，然后收集异常的信息，以供 程序员诊断错误时使用。

12J.2 Pyt怖◎麻囂圍昜囂

Python使用异常对象来表示不同的异常,并且已经为常见的异常建立了异常类,在写程序 时可以直接使用这些内建的异常类。Python中常见的异常类型如表12-1所示。

表心常见内建异常

|  |  |  |  |
| --- | --- | --- | --- |
| 类 名 | 描 述 | 类 名 | 描 述 |
| Exception | 所有异常类的基类 | IndexError | 当使用超出列表范围的索引时引发 |
| SyntaxError | 发生语法错误的异常类 | KeyEn\*or | 当使用映射不存在的键时引发 |
| F ileN otF oundError | 未找到指定文件或目录时引发 | AttributeError | 当尝试访问未知对象属性时引发 |
| NameError | 找不到指定名称的变量时引发 | TypeError | 当试图在使用a类型的场合使用b类型时引发 |
| ZeroDivisionError | 除数为0时的异常 |  |  |

下面对上述常用的几种异常类进行介绍，具体如下：

1. NameError

尝试访问一个未声明的变量，会引发NameError异常。例如:

>>> print(abc)

Traceback (most recent call last):

File H<stdin>Hr line *1f* in <module>

NameError: name ' abc \* is not defined

上述信息表明，解释器在任何名称空间里面都没有找到abco

1. ZeroDivisionError

当除数为零的时候,会引发ZeroDivisionError异常。例如:

»> 2/0

Traceback (most recent call last):

242 虹 Pjthan SMSOffl: AA饕雾 PyBwn

File "<stdin>"f line 1f in <module> ZeroDivisionError: division by zero

(3 ) IndexError

当使用序列中不存在的索引时，会引发IndexError异常。例如:

>>> my\_list = [0]

>>> my\_list[1]

Traceback (most recent call last):

File n<stdin>"f line 1z in <module> IndexError: list index out of range

上述示例中，myjist列表中只有一个元素，当使用索引1访问列表的第二个元素时，就引 发IndexError异常,并提示列表的索引值超出了列表的范围。

(4) KeyError

当使用映射中不存在的键时，会引发KRyEnw异常。例如：

>>> my\_dict = { ? name 3: ? xiaoMing , !age v: v18 \*}

>>> my\_\_dict [ ? g㊀ndeb]

Traceback (most recent call last):

File K<stdin>"z line 1e in <module> KeyError: \* gender \*

上述示例中,my dict字典中只有name和age两个键,当尝试获取gender键对应的值时, 就出现KeyError异常。

(5 ) AttributeError

当尝试访问未知的对象属性时,会引发AttributeError异常。例如:

class Car(object):

pass

car = Car()

car . color ="黑色 ° print(car.color) print(car«name)

上述示例中,Car类没有定义任何属性和方法,在创建Car类的实例以后,动态地给car 引用的实例添加了 color属性，然后访问它的color和name属性时，出现如下错误信息:

Traceback (most recent call last):

File "D:/PythonCode/Chapterl2/ »pyf\ line 6, in <module> print(car.name)

AttributeError: J Car , obj ect has no attribute ? name \*

上述信息表明，在Cair的实例中定义了 cokr属性，可以使用点访问；如果没有名称为 name的属性，就会引发上述异常。

(6) TypeError

当试图将与变量类型不相符的值赋给变量时会引发TypeError,例如：

»> i = 5

»> print ("i ° + 5)

Traceback (most recent call last):

File "<stdin>° r line 1, in <module>

TypeError: must be strz not int

上述代码在priiM)函数中，试图将字符串和整数用符号“+”连接在一起，引发了 TypeError。由于操作的左边是字符串类型，所以右边应该也是字符串类型，在应该使用字符串 类型的场合使用了整型，这是错误的用法。

虞提攔藐异常

前面已经说过，不能在程序中放任异常出现「否则会出现糟糕的用户体验。为此，有必要 捕获异常并作出相应的处理。

Python中使用try-except语句处理异常9其中try语句检测异常，except语句捕获异常。

12n2n1

try-except语句的格式如下: ..

try: 可能出错的代码

except ［错误类型］:

出错后的处理语句

try语句后面是可能出错的代码, except语句后是捕获的异常类型，并指明捕获到异常后的 处理方法。

try-except语句的工作方式如下:

(1 )首先,执行try子句，即在try和except之间的代码;

(2 )如果try子句没有发生异常，则忽略except后的子句；

1. 如果try子句发生异常，则忽略该子句的剩余语句，此时有两种可能：

-如果发生的异常类型与except后指定的异常类型一致，则执行except子句须然后继续执 行try™except语句后面的代码。

•如果发生的异常类型与except后指定的异常类型不一致，则该异常会被抛出到上一级代 码，由上一级代码处理。如果最终都没有得到处理，就会使用默认的处理方式：程序崩溃, 终止运行，并显示提示信息。

下面是一个异常处理的例子:

»> try:

|  |  |
| --- | --- |
| ... print （ ° 我的年龄是:" + 18） | #出错代码 |
| ... print （ °我叫小明"*）* | #被忽略的代码 |
| …except TypeError: |  |
| …. print （ ° 出错了 "） | #错误处理代码 |

出错了

这个例子中，在试图使用符号“+”将字符串和数字连接在一起时发生了类型异常 (TypeError),但是程序并没有崩溃。Exc项语句捕获到了这个异常，告诉解释器程序已经做 了相应的处理，于是解释器忽略异常后的代码，执行错误处理代码(这个例子是打印一条消息)。

由此可以看出，经过异常处理后的代码更健壮。

12.2.2

有些异常可以在程序中纠正，有些不能，但是不论是否可以在程序中处理，程序员都有必 要全面了解程序的运行情况，包括程序岀现的所有异常信息。一般在程序中会维护一个日志， 记录程序中出现的所有异常，此时可以使用舔获取系统反馈的异常具体信息。例如：

»> try:

…. print ("我的年龄是:” + 18)

… print (°我叫小明° )

。…except TypeError as error:

…o print ("错误信息:° )

.。. print(error)

错误信息：

must be str7 not int

得到具体信息后，就可以进行记录了。

12.2.3翳褻爹命嘉黨

现实情况是，一段代码可能出现多个异常，此时，可以将多个特定异常组成一个元组放在 一个except语句后处理，也可以多个except子句联合使用。比如下列代码：

>» try:

• 。 • print(abc)

... print (2 / 0)

"… except (NameError^ ZeroDivisionError) as error:

o". print(error)

name 5 abc5 is not defined

这段代码在try子句里试图访问未定义的abc变量,会引发NameError异常;而用0作为除数， 会引发ZeroDivisionError,所以在except语句中，将两种异常包含在一个元组中，并打印异常 的详细信息。

同时捕获多个异常，只能使用一种方式进行处理。如果希望对不同的异常采用不同的处理 方式，可以使用多个except子句，比如下面的写法：

»> try:

.。. print (2 / 0)

... print(abc)

... except NameError as error:

•.. print(error)

..o except ZeroDivisionError:

•… print ("不能使用0作为除数°)

不能使用。作为除数

这里将两个异常分别用except子句处理，方便对不同的异常使用特定的处理方式。

(分主意

捕获多个异常的写法，具体如下:

° Python 2里把多个异常用逗号隔开即可,例如：

except NameErrorf ZeroDivisionError as error:

◎ Python 3里需要将多个异常用元组组织起来，也就是多了个圆括号，例如：

except (NameErrorfZeroDivisionError) as error:

12.2.4 "

使用前面小节讲解的捕获异常的方式，可以捕获特定名称的异常并作出相应处理，那么怎 么保证我们罗列了所有的异常呢？万一有“漏网之鱼”，一旦发生就会引起程序崩溃，所以最 好能捕获所有的异常。

在Python中，捕获所有的异常有2种方式，一种是使用Exception,由于Exception是所有 异常的基类，所以可以包含所有的异常。比如下面的代码：

>>> try:

"… demo\_list = 传智"/黑马七°酷丁鱼°, °博学谷° ]

…。 print (demo\_list [5])

"…except Exception as error:

... print(error)

… print、程序出错，请联系管理员。“)

demo\_list index out of range

程序£错，请联系管理员。

在try子句里,demo list只有4个元素,当试图访问第5个元素时报出异常IndexError, 在except语句里没有指明异常的具体类型，而是使用了所有异常的基类Exception。从结果可以 看出，这个异常也被捕获到了，并且打印出了异常的信息。

另一种方式是只使用except,如下面的代码：

»> try: demo\_dict = { "name" : "xiaoMing" }

... print(demo\_dict[°age"])

o o o except:

.。. print ("出错了,具体错误未知。")

出错了,具体错误未知。

在except后空缺异常类型，也会将所有的异常都捕获，不过这种方式并不推荐，原因如下：

•不能获取错误的具体信息。

魁它还会捕捉我们并不想捕获的异常，比如用户终止执行的Ctrl+C操作，以及用sys,exit() 函数终止程序的操作，等等。

E注意

*t - "*

1. 异常处理的主要目的是防止因外部环境的变化导致的程序无法控制的错误，而不应 该用于处理程序的设计错误。所以将所有的代码都用try语句包含起来的做法是不正确的。
2. 另外，捕获了异常之后应作出对应处理，如果在异常后用pass直接忽略掉，会隐藏 掉导致程序出错的原因，不利于程序的正常运行和维护，也是不推荐的。

12d2q5蔔屈字倒 ’

else已经除了可以与if语句和循环语句配合使用之外，还可以与try^except语句配合使用, 表示当try子句没有出现错误时应执行的语句。

它的格式如下:

try:

可能出错的语句

except:

出错后的执行语句

else :

未出错时的执行语句

例如，在分页显示数据时，从用户输入获取每页显示多少条数据，此时需要先将用户输入 的字符串转化为数字。如果转化出错，则使用预设的数据量(例如每页显示20条)；如果转 化成功,则使用新的每页数据量重新加载数据。代码如下：

用户输入为字符串

>>> word = ° 10 °

>>> try:

•… page\_size = int (word)

*.。*。except:

。e. page\_size = 20

o <, . else :

• \* o reload\_data (page\_size)

将字符串转化为数字

如果转化出错，则使用预设的数据量

重新加载数据

12D2n(B fmaiy

finally子句与try语句联合使用,表示无论try语句是否出错都会执行的语句。 它的格式如下所示。

try:

可能出错的语句

finally:

无论是否出错都会执行的语句

在实际开发中，缶ially子句用于做一些善后工作。比如从数据库读取数据，首先要连接数 据库，然后对数据库进行操作。由于数据库是外部因素，不受程序的控制，所以对数据库的操 作容易出异常。但是数据库连接已经创建，占用了系统资源，所以，无论对数据库的操作成功 与否，都要将数据库连接关闭，并释放相应的资源，这个时候就可以在fin浏y子句中进行。

同样的还有访问本地或者网络文件等。

例如，以下代码实现了打开本地文件（alxt） 9并向该文件写入字符串的功能： try:

file = open （ 1 a «txt? f 1 r \* ） file . write （° 人生苦短 f 我用 Python"*）* except Exception as error:

print （"写入文件失败", error） finally:

file。close （*）*

在上述代码中，使用了 try语句将打开本地文件（&txt）和向文件中写入字符串的操作包 含在内，并将关闭文件对象的语句放在了 finally子句中，这样无论文件读写执行是否成功，都 会将该文件对象占用的资源（如果占用了资源）释放。

在Python 2.5之前,finally子句曾经不能与try-except或try-except-else子句一起使用,只能使 用try-finally,但是这并不符合大部分程序员的习惯。从Python 2.5开始,finally子句可以与except 子句和©屐子句自由组合，与tiy语句联合使用。一个最全面的捕获异常的语法格式如下所示。

try:

可能出错的语句

except: 异常发生后的处理语句

else： 异常未发生的执行语句

finally: 无论异常是否发生都会执行的语句

主意 •如果在finally子句引发了异常,则该异常无法捕捉。

•如果在finally子句中使用return, break, continue语句而终止,则原来的异常也会丢失, 并且无法重新引发。

如果捕获到的异常在本级无法处理，或者不应该由本级处理：也可以将异常抛出，交给上 一级代码处理。

12o3J raise 福匍

raise语句用于抛岀特定的异常，它的基本格式如下：

第1种：创建类的实例对象，引发异常。

raise异常类名 #创建类的实例对象，并引发异常

第2种9引发异常实例对象对应的异常。

raise异常类实例对象 #引发异常类实例对象对应的异常

第3种：重新引发刚才发生的异常。

raise #重新引发刚刚发生的异常

在上述格式中，第1种和第2种方式是对等的，都会引发指定异常类的实例。但是，第1 种形式隐式地创建了异常类的实例，而第2种形式是最常见的，直接提供一个实例。第3种用 于重新引发刚刚发生的异常。

下面通过实例来对raise语句进行介绍。

们使用类客引裝异常

当raise语句指定异常的类名时，会创建该类的实例对象，然后引发异常。例如：

raise IndexError

程序运行的结果如下：

Traceback (most recent call last):

File "D: /PythonCode/Chapter 12/ 异常.pyn f line 1 z in <module> raise IndexError

IndexError

2.瘻照劈常类的实例引綬异常

通过显式地创建异常类的实例，直接使用该实例对象来引发异常。例如：

index = IndexError()

raise index

上述7K例创建了一个IndexError类的实例index,然后使用raise让index实例引发异常。 程序运行的结果如下：

Traceback (most recent call last):

File "D: / PythonCode/Chapter 12/ 异常.pyH, line 2, in <module> raise index

IndexError

& ft禳舅鬻

不带任何参数的函院语句，可以再次引发刚刚发生过的异常，作用就是向外传递异常。 例如：

try:

raise IndexError

except:

print (° 出错了 w)

raise

上述示例中,try里面使用raise抛出了 IndexError异常,程序会跳转到except子句中执行, 输出打印语句，然后使用函se再次引发刚刚发生的异常，导致程序出现错误而终止运行。程序 运行的结果如下：

出错了

File °D:/PythonCode/Chapterl2/ .py°z line 27 in <module> raise IndexError

IndexError

4b指定异常的撇述億息

当使用raise语句抛出异常时，还能给异常类指定描述信息。例如：

raise IndexError （"索引下标超出范围° ）

上述示例中，在抛出异常类时传入了自定义的描述信息。程序的运行结果如下x

Traceback （most recent call last）:

File °D: /PythonCode/Chapterl2/ 异常.py"r line 1 r in <module> raise IndexError （"索引下标超出范围"）

IndexError :索引下标超出范围

凱异鬻引叢異鬻

如果要在异常中抛出另外一个异常，可以使用raise^from…语句实现。例如：

try:

num

except Exception as exception:

raise IndexError （° 下标超出 范围 "） from exception

上述示例中，try里面只定义了变量num,并没有为其赋值，所以会引发NameEnw异常， 使得程序跳转到except子句中执行。铤顷子句能捕捉所有的异常，并且使用raise.^from…语 句抛出NameError异常后再抛出“下标超出范围”的异常。程序的运行结果如下：

Traceback (most recent call last):

File °D: /PythonCode/Chapterl2/ 异常.py°, line 2, in <module>

num

NameError: name \* num \* is not defined

The above exception was the direct cause of the following exception: Traceback (most recent call last):

File °D: /PythonCode/Chapterl2/ 异常 .py", line 4 f in <module> raise IndexError (n 下标超出范围 ") from exception

IndexError :下标超出范围

12.3.2異囂齣後還 "’

如果异常没有被处理，默认情况下会将异常传递给上层代码，逐级向上传递。如果最上层 的代码也没有处理，则会使用系统默认的方式。下面的例子用三个函数实现计算正方形面积的 功能，代码如下：

>>> def get width():

... print ( ㊀t\_width 计算开始")

... a = 10 / 0 #发生异常

…。 print ("--get\_width 计算开始")

... return a

>>> def square():

…. print (" —square 计算开始 ° ) b = get\_width ()

。… print ( "--square 计算结束")

... return b \* b

»> def area ():

…。 try:

。… print (”一 area计算开始° )

... c = square ()

… print ("--area 计算结束 11)

... except ZeroDivisionError as error:

…. print ("捕捉到异常")

»> area ()

--area计算开始

--square 计算开始

--get\_width计算开始 捕捉到异常

上述代码，定义了三个函数，其中，取s()函数调用了 square()函数，square()函数又调用 T get\_width()函数。在 get\_width()函数中，语句 “10/0” 发生了 ZeroDivisionError 异常，但是 该函数没有进行任何处理,异常默认传递给了函数square()osquare()函数同样没有进行任何处理, 该异常又被传递给了 area()函数。area()函数内部捕捉到了 ZeroDivisionError异常，并打印提示 信息。

从运行结果可以看出，area()函数捕捉到了 get\_width()函数中传递过来的异常。如果area() 函数也不捕获异常，则默认的方式是程序崩溃，在控制栏显示异常的详细信息。

需要注意的是，根据打印信息可以看出，发生异常之后剩下的代码都没有被执行。三个函 数都只执行了发生异常之前的打印语句，而没有执行发生异常之后的语句。即使在ars()函数 中捕捉了异常，发生异常后的语句也被跳过，并没有被执行。

.12D3n3 assert .

龄sort断言语句用于判定一个表达式是否为真，如果表达式为True,则不做任何操作，如 果为False则会引发AssertionError异常。assert断言语句的格式通常为

assert表达式［r参数］

其中，表达式是assort语句的判定对象，参数通常是一个字符串，是自定义的异常参数， 用于显示异常的描述信息。

assert断言可以理解为函ssiFnot，,艮卩

if not表达式

raise AssertionError (［参数］)

例如，T会员管理系统，要求会员的年龄必须大于等于18岁，贝加以对年龄进行断言，如下所示。

>>> age = 17

>>> assert age〉= *18 r 11*年龄必须大于等于18岁"

Traceback (most recent call last): File "<stdin>"f line 1in <module>

AssertionError :年龄必须大于等于18岁

其中，age >=18就是assert语句要断言的表达式，“年龄必须大于等于18岁”是断 言的异常参数。程序运行时，由于age=179要断言的表达式值为珂獎，所以系统抛岀了 AssertionError异常，并在异常后显示了自定义的异常参数。

assert断言语句抛出的AssertionError异常可以通过try-except语句捕捉,比如将上例引发 的异常捕获，代码如下。

>>> age = 17

>>> try:

…. assert age >= 18f °年龄必须大于等于18岁°

...except AssertionError as error:

.。. print(n%sz %sH%(error. class . name r error))

AssertionErrorz年龄必须大于等于18岁

assert断言语句用于判定用户定义的约束条件，在程序未完成时，可以使用断言语句对条 件进行判定。与其让程序在运行中崩溃，不如在遇到不满足的条件时就报出异常。

1X4自定WB

虽然系统的内建异常很多，但有时候我们希望创建新的异常类。自定义异常类的方法很简 单9只需要创建一个类，让它继承自Exception类或者Exception类的子类即可。例如：

class myException(Exception):

pass

上述代码创建了一个自定义异常类，是不是非常简单呢？

那么如何使用自定义异常呢？直接使用函se语句抛出即可。例如：

>>> class myException(Exception): •.. pass

»> try:

。.. pass

。. raise myException ( n 出错啦 n )

…。except myException as error:

。。。 print(error)

出错啦

上述代码创建了一个自定义异常myException,在try语句中可通过raise语句将myException 抛岀歹还可以为异常指定异常提示信息。

在创建自定义异常时，可以像创建普通类一样赋予它属性和方法，但是我们通常不添加或 者只添加几个属性，用于描述异常的详细信息即可。

在实际开发中，可以创建自己的异常体系，先创建一个基类继承自Exception,然后针对不 同情况创建不同的子类。

例如，在实现用户上传图片的功能时，可以按不同的出错情况设计不同的异常：文件名称 不对,文件是空文件,文件格式不对等。代码如下所示。

>>> class FileUploadError(Exception):

#定义异常的基类

… def init (self, err=n 文件上传出错"): super() o init (selfz err)

class FileNameError (FileUploadError) : # 定义异常的子类

»>

»>

»>

def init (self z err^0 文件名应以.jpg,。png, . jpeg 作为后缀")： super() . init (s e1f z err)

#定义异常的子类

class EmptyFileError(FileUploadError): def init (seIf z err=n 文件为空"): super(). init (selfz err)

class FileFormatError (FileUploadError) : # 定义异常的子类

def init (self z err=H文件已损坏或者格式不对f无法打开"):

super(). init (selfz err)

>>> file\_name = °a»gif"

»> try:

• if file\_name . split (". °) [ 1 ] in [ ° jpg° r "png" 7 "jpeg"]:

*。*。. pass

o.» else:

raise FileNameError ()

。…except Exception as error:

.•. print(error)

文件名应以.jpg, .png, . jpeg作为后缀

上述代码中,首先定义了 一个异常基类FileUploadError,直接继承自Exception类。 FileUploadError类重写了 \_init\_方法,将默认的错误信息传递进去,这样的好处是,在引发 异常时可以指定异常描述信息，也可以使用默认的信息，使用起来更灵活。

然后定义了 FileUploadError 的三个子类:FileNameError^ EmptyFileError 和 FileFormatError,分 别用于表示文件名称错误、空文件错误和文件格式错误。

后面的代码获取了文件名的后扩展名，判断扩展名名是否为指定的图片后扩展名，如果是 则继续其他操作*（*这里用pass语句省略），如果不是则使用raise语句引发FileNameError,在 引发FileNameError时没有传入参数，所以会使用默认的异常提示信息。

最后捕获异常，将异常提示给用户。

除了创建自己的异常体系之外，还可能创建继承自Exception类的子类的自定义异常类，

第12章 错误和异常 w 253 为异常添加更详细的描述信息。例如，创建两个继承自FileNotFoundError的异常，一个用于 找不到文件的异常，一个用于找不到目录的异常。

；12o5 with^甸建耀异常

通过前面的章节9我们已经学习了使用try-except-finally语句等处理异常的方法。除此之夕卜， 还可以使用with语句处理与异常相关的工作。with语句支持创建资源，抛出异常，释放资源等 操作，并且代码更简洁。接下来，就针对with语句进行详细的介绍。

12o5a1 woth 福匐 '

with语句适用于对资源进行访问的场合，无论资源使用过程中是否发生异常，都会执行必 要的释放资源的操作，比如文件使用后自动关闭、线程中锁的自动获取和释放等。

with语句的语法格式如下:

with上下文表达式［as资源对象］:

对对象的操作

其中，上下文表达式返回一个上下文管理对象。如果指定了踮子句，该对象并不赋值给 as子句中的资源对象，而是将上下文管理器的\_en传」()方法的返回值赋值给资源对象。资源 对象可以是单个变量，也可以是元组。

例如,使用with语句操作文件对象的示例如下:

with open ( \* /test. txt1 ) as file :

for aline in file:

print(aline)

这段代码使用with语句打开文件，如果顺利打开，则将文件对象赋值给切饥然后用for 语句遍历和打印文件的每一行。当对文件的操作结束后，with语句关闭文件。如果在这段代码 运行过程中发生了异常,with语句也会将文件关闭。

这段代码使用finally语句也可以实现：

try:

file = open (! /test. txt \* )

try:

for aline in file:

print(aline)

except Exception as error: print(error)

finally: file \* close () except FileNotFoundError as err:

print(err)

我们也可以给with语句加上异常处理，代码如下:

try:

with open ( ' /test. txt , ) as file :

for aline in file: print(aline) except Exception as error: print(error)

这段代码和上一段代码的效果是一样的，但是显然使用With语句更简洁，减少了代码量， 而且能避免手动释放资源时的遗漏情况。

with 语句从 Python 2.5 开始引入(需通过 from future import with statement 导入),从 2.6版本开始可以缺省。

/源"注意：不是所有对象都可以使用with语句，只有支持上下文管理协议的对象才可以，目 前支持该协议的对象如下：

° fileo

* decimal. Context o

° thread.LockType o

® threading.BoundedSemaphore o

° threading.Conditiono

® threaciing.Locko

® threading.RLock0

* threading. Semaphore o

12.5.2上TF黨簷聽器

要理解with语句，一个重要的概念是上下文管理器。前面已经说过，只有支持了上下文管 理协议的对象才能使用with语句。

上下文管理协议与with语句有关的概念包括：

1 上TF支管理协议(Context Management ProtocoE)

上下文管理协议包含了方法\_enter\_()和\_exit\_()9支持该协议的对象需要实现这两个 方法。这两个方法的含义如下：

* \_enter\_(self)：进入上下文管理器时调用此方法，其返回值被放入withes语句中曲说 明符指定的变量中。

® \_\_exit\_(self5 type, value, tb)：离幵上下文管理器调用此方法。如果有异常出现,type, value> tb分别为异常的类型、值和追踪信息；如果没有异常貝3个参数均设为None。 此方法返回值为True或者False,分别指示被引发的异常得到了还是没有得到处理，如 果返回False,引发的异常会被传递出上下文。

1. 上TXWIiH (Context Manager)

支持了上下文管理协议的对象就是上下文管理器，这种对象实现了 \_enter\_()和\_exit\_() 方法。使用with语句即可调用上下文管理器，它负责建立运行时上下文。

1. 运行肘上下交 「

运行时上下文由上下文管理器创建，通过上下文管理器的 enter ()和 \_exit\_()方法实现。

4D ±下支表达式(Context Expression):

with语句中在关键字with之后的表达式，该表达式要返回一个支持了上下文管理协议的对象， 也就是返回一个上下文管理器。

0语旬体

由with语句包括起来的包括块，表示对资源对象的操作。在执行语句体之前会执行上下文 管理器的 enter () 9执行完后调用 exit ()方法。

了解上下文管理器之后，就能理解with语句的执行过程了，具体如下：

(1 )先执行上下文表达式，生成上下文管理器对象。

1. 接着，调用上下文管理器的\_皿脸\_()方法，如果使用了 as子句，就把\_皿切\_() 方法的返回值赋值给as子句中的变量。
2. 然后执行语句体。
3. 无论在执行的过程中是否发生异常，都会执行上下文管理器对象的\_exit\_()方法。 该方法负责执行程序的“清理”工作，如释放资源等。

(5 )如果执行过程中没有岀现异常，或者语句体中执行了 break, continue或者Mtum语句， 则以None作为参数调用—exit\_()方法;如果执行过程中出现异常,则会使用sys.exc\_info得 到的异常信息为参数调用\_exit\_()方法。

(6)出现异常时，如果\_exit\_()方法返回的结果为F/Isr,则会重新抛出异常，让with 之外的语句逻辑来处理异常，这是通用做法；如果返回Tr械，则忽略异常，不再对异常进行处理。

12.5.3圓竊製上芋夜簷蹲器

在开发中，我们也可以自己定义一个上下文管理器，只需要让它支持上下文管理协议，并 实现该协议规定的 enter ()和 exit ()两个方法即可。

接下来，'我们通过一个简单的示例来演示如何构建自定义的上下文管理器，示例如下所示。 class A(object):

def enter (self):

print(? enter () called1)

return self

def print\_description(self): print("my name is A!")

def exit (selff e\_t7 e\_vz t\_b):  
print(1 exit () called')

#首先会执行 enter 方法

with A () as a : # a 为 enter 的返回对象

a。print\_description() print ( 1 a is called!)

#结美会执行 exit 方法

在上述代码中,第1~8行定义了一个上下文管理器A,它实现了 \_enter\_()方法和\_exit\_() 方法，并在\_«iter\_\_()方法中返回对自身的引用。

256釦,Python襄毓疆濫 風零專Pythoo

运行这段代码0它的输出结果如下所示。

enter () called

my name is A!

a is called

exit () called

从结果可以看出，使用with语句，在生成上下文管理器之后先调用了该对象的\_\_enter\_() 方法，然后执行语句体，最后执行了上下文管理器对象的\_\_exit\_\_()方法。