第六讲
文件操作与异常处理
Lecture 6
File Operations and Exception Handling

明玉瑞 Yurui Ming

yrming@gmail.com

声明
Disclaimer

- 本讲义在准备过程中由于时间所限，所用材料来源并未规范标示引用来源。所引材料仅用于教学所用，作者无意侵犯原著者之知识版权，所引材料之知识版权均归原著者所有；若原著者介意之，请联系作者更正及删除。

  The time limit during the preparation of these slides incurs the situation that not all the sources of the used materials (texts or images) are properly referenced or clearly manifested. However, all materials in these slides are solely for teaching and the author is with no intention to infringe the copyright bestowed on the original authors or manufacturers. All credits go to corresponding IP holders. Please address the author for any concern for remedy including deletion.

文件操作
File Operations

- 在许多非平凡程序中都有对文件的操作，本讲介绍Python中涉及文件操作的知识及发生异常时的处理。

  In non-trivial programs operations to files is an indispensable part. This lecture covers the file operations in Python and how to handle the exceptions upon occurring.

- 操作文件之前，首先要打开指定的文件。打开文件的操作在Python中用open()函数。open函数会返回一个文件对象，可以调用该对象的方法，如读取操作，进行文件读取。在文件操作完成之后，关闭文件以释放该文件对象所占用的资源。

  One needs to open the file before applying operation to it. In Python to open a file you need to call open() given the file path, and extra permissions which are optional. Upon success a file object will return, of which methods of it can be called, for example, reading a file. After finish the operation, it is obligatory to close the file in order to release the resource occupied by the file object.

文件操作
File Operations

```
C:\Users\yrming\Desktop\备课\Python程序设计>cat lec_6_pi_digits.txt
3.1415926535
8979323846
2643383279
C:\Users\yrming\Desktop\备课\Python程序设计>cat lec_6.py
file_object = open('lec_6_pi_digits.txt')
contents = file_object.read()
print(contents)
file_object.close()

C:\Users\yrming\Desktop\备课\Python程序设计>python lec_6.py
3.1415926535
8979323846
2643383279
```

文件操作
File Operations

- 在 Python 中处理文件时，须在对文件执行读取或写入操作后关闭文件，不然会造成资源如内存的泄漏。但由于跟踪内存使用并不是一件容易的事，为帮助程序员书写正确的程序，推荐在处理文件时使用with语句。

  When working with files in Python, it‘s essential to close the file after performing read or write operations on it. Otherwise, it will cause resource such as memory leaking. However, considering tracking the memory usage is not a trivial task, it is recommended to use with statement to help maintain the correctness and robustness when implementation.

- 下面例子展示了如何用with语句重写上面的程序。

  The example below demonstrates how to use with statement to rewrite the above program.

```
>>> with open('lec_6_pi_digits.txt') as file_object:
...     contents = file_object.read()
...     print(contents)
...
3.1415926535
8979323846
2643383279
>>>
```

文件操作
File Operations

- 在Python中执行with语句时，会创建了一个运行时上下文，随后的语句都是在该上下文管理器的控制下运行的。with语句可以使代码更清晰、更安全和可重用，因此，标准库中的许多类都支持 with 语句。那么什么是上下文管理器呢？上下文管理器是具有实现了__enter__和__exit__两个特殊方法的对象。open()函数之所以可以与with语句一起使用的原因是，因为它返回的文件对象实现了上下文管理器协议。实际上，只要实现了上下文管理器协议，任何用户定义的对象都可以使用 with语句。

  In Python, execution of with statement actually creates a runtime context that allows one to run a group of statements under the control of a context manager. The with statement can make the code clearer, safer, and reusable. Hence, many classes in the standard library support the with statement. The reader might ask that's context manager. A context manager is an object that has __enter__ and __exit__ methods implemented. The open() function works with the with statement because the file object it returns implements the context manager protocol. Any user-defined object can work with the with statement as long as it implements the context manager protocol, namely, the __enter__ and __exit__ methods.

文件操作
File Operations

- with 语句一般具有以下语法形式：

  with statement needs to follow general syntax below:

```
with expression as target_var:
    do_something(target_var)
```

- 上下文管理器对象由计算with之后表达式得出，即表达式须返回一个实现上下文管理协议的对象。其中，with 语句调__enter__()以进入运行时上下文，在执行离开 with 代码块时调用__exit__() 以释放资源。

  The context manager object results from evaluating the expression after with, which means expression must return an object that implements the context management protocol. In detail, __enter__() is called by the with statement to enter the runtime context, and __exit__() is called when the execution leaves the with code block.

文件操作
File Operations

- 注意，一些上下文管理器的__enter__()实现并没有有实际意义的对象可供返回，因此返回给调用者的为None。在这种情况下，指定target_var并没有意义。

  Note: Some context managers return None from __enter__() because there is no useful object to give back to the caller. In these cases, specifying a target_var makes no sense.

```
C:\Users\yrming\Desktop\备课\Python程序设计>cat lec_6_with.py
import sqlite3

class Conn:
    def __init__(self, db_name="test.db"):
        self._db_name = db_name

    def __enter__(self):
        try:
            self._conn = sqlite3.connect(self._db_name)
            print(f"Open database {self._db_name} successfully")
        except:
            print(f"Failed to open database {self._db_name}")

        return self._conn

    def __exit__(self, type, value, traceback):
        self._conn.close()

with Conn() as conn:
    c = conn.cursor()
    c.execute('''CREATE TABLE COMPANY
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
    conn.commit()
    print ("数据表创建成功")
C:\Users\yrming\Desktop\备课\Python程序设计>python lec_6_with.py
Open database test.db successfully
数据表创建成功
```

文件操作
File Operations

- 在之前的例子中，利用read()读取文件时，默认情况下，会读取整个文件的内容。尽管可以指定一次读取的大小，有时候可能按行读取，特别是处理文本文件时，会很方便，下面改进的程序：

  In the previous example, using read() to read content from file results in all the content to be retrieved from the file by default. Although the size for each time read can be specified, now and then that read and process line by line is a better way especially for text files. The following example program demonstrates the idea:

```
>>> with open('lec_6_pi_digits.txt') as file_object:
...     for line in file_object.readlines():
...         print(line)
...
3.1415926535

8979323846

2643383279
>>>
```

文件操作
File Operations

- 在上面例子中，执行打印操作时，每一行会有额外的空白行。这些空白行的出现是由于在文本文件中每行的末尾，都有一个不可见的换行符。每次调用print函数时，其会添加自己的换行符，因此每行末尾的实际上有两个换行符：一个来自文件，一个来自print()。可以在调用print()的每一行上使用rstrip()，消除这些额外的空白行：

  Even more blank lines appear when we print each line. These blank lines appear because an invisible newline character is at the end of each line in the text file. However, the print function adds its own newline each time we call it, so we end up with two newline characters at the end of each line: one from the file and one from print(). Using rstrip() on each line in the print() call eliminates these extra blank lines:

```
>>> with open('lec_6_pi_digits.txt') as file_object:
...     for line in file_object:
...         print(line.rstrip())
...
3.1415926535
8979323846
2643383279
```

文件操作
File Operations

- 当文件对象支持for循环时，说明文件对象为可迭代对象。可迭代对象为实现了__iter__方法的对象。下面例子展示了这种用法：

  Since the file object is supporting for loop, which means the file object is an iterable object. Iterable objects are the ones which have the special method __iter__ implemented. The follow example illustrate this:

```
>>> class TrivialIterable:
...     def __init__(self, begin=0.0, end=10.0, step=1.0):
...         self._begin = begin
...         self._end = end
...         self._step = step
...         self._incr = begin
...     def __iter__(self):
...         while True:
...             if self._incr >= self._end:
...                 return None
...             else:
...                 yield self._incr
...                 self._incr += self._step
...
>>> for i in TrivialIterable(end=5.0):
...     print(i)
...
0.0
1.0
2.0
3.0
4.0
>>>
```

文件操作
File Operations

- 实现了__iter__方法，同时又实现__next__方法的对象称为迭代器。通常，出于实用的原因，可迭代对象一般公同时实现__iter__()和__next__()，并让__iter__()返回self，即对象自身，这使得该类既是可迭代的，本身又是迭代器。

  Objects that both implementing __iter__ and __next__ methods are called iterators. Often, for pragmatic reasons, iterable classes will implement both __iter__() and __next__() simultaneously and have __iter__() return self, which makes the class both an iterable and its own iterator.

```
>>> class TrivialIterator:
...     def __init__(self, begin=0.0, end=10.0, step=1.0):
...         self._begin = begin
...         self._end = end
...         self._step = step
...     def __iter__(self):
...         self._incr = self._begin
...         return self
...     def __next__(self):
...         if self._incr >= self._end:
...             raise StopIteration
...         else:
...             ret = self._incr
...             self._incr += self._step
...             return ret
...
>>> for i in TrivialIterator(end=5.0):
...     print(i)
...
0.0
1.0
2.0
3.0
4.0
>>>
```

文件操作
File Operations

- open()的第一个参数是要操作文件的路径，第二个参数mode表示打开文件时，对文件的操作模式。常见的模式有读取模式（‘r’）、写入模式（‘w’）、附加模式（‘a’）或同时允许读取和写入文件的模式（‘r+’）。如果省略 mode 参数，Python 默认以只读模式打开文件。当要将文本写入文件时，第二个参数需要设成‘w’以告诉Python我们要以写入模式打开文件。此时如果要写入的文件不存在，open() 函数会自动创建它。但是，以写入模式（’w’）打开文件需要小心，因为如果文件确实存在，Python 将在返回文件对象之前擦除文件的内容。

  The first parameter to the open() call is the name of the file we want to open; the second argument indicates the operating mode when opening the file. The general options are read mode ('r'), write mode ('w'), append mode ('a'), or a mode that allows to read and write to the file ('r+’). If the mode argument is omitted, Python opens the file in read-only mode by default. To write text to a file, we need to call open() with a second argument 'w', tells Python that we want to open the file in write mode. The open() function automatically creates the file if it doesn’t already exist. However, be careful opening a file in write mode ('w') because if the file does exist, Python will erase the contents of the file before returning the file object.

文件操作
File Operations

- 向文件对象所代表的文件中写入内容可用write()函数，下面例子展示了这个用法：

  One can use the write() function to write contents such as text to a file. The following example demonstrates the routine operation:

```
>>> filename = 'programming.txt'
>>> with open(filename, 'w') as file_object:
...     file_object.write("I love programming.")
...     file_object.write("I love creating new games.")
...
19
26
>>> from subprocess import check_output
>>> check_output("cat programming.txt", shell=True)
b'I love programming.I love creating new games.'
>>>
```

文件操作
File Operations

- 当用户如果是想将内容添加到文件而不是覆盖现有内容时，可以以附加模式打开文件。

  If the user wants to add content to a file instead of writing over existing content, then the file can be opened in append mode.

- 注意，write()对文件的操作为有缓冲的操作，意味着文件内容在调用返回之后未必真正会写入文件。用户可以调用flush()进行文件强制写入。

  Note that the operation of write() on the file is a buffered operation, which means that the content of the file may not actually be written to the file after the call returns. The user can call flush() to force write to the file.

- Windows平台用反斜杠来分割文件路径，但因为反斜杠用于转义字符串中的字符，用户可能因此收到错误消息。例如，在路径“C:\path\to\file.txt”中，序列\t 被解释为制表符。因此如果需要使用反斜杠，可以对路径中的每一个进行转义，如：“C:\\path\\to\\file.txt”。

  On Windows, backslashes are used to separate file path. However, since the backslash is also used to escape characters in strings, you might get an error. For example, in the path "C:\path\to\file.txt", the sequence \t is interpreted as a tab. If you need to use backslashes, you can escape each one in the path, like this: "C:\\path\\to\\file.txt".

异常
Exceptions

- 当在程序中发生错误，Python不能确定下一步怎么做时，它会首先创建一个异常对象，用该特殊对象来管理程序执行期间出现的错误。如果用户不处理异常，程序将停止并显示回溯，其中包括引发异常的报告。如果用户希望程序继续运行，则编写处理异常的代码。

  Whenever an error occurs that makes Python unsure what to do next, it creates an exception object first, and uses special objects called exceptions to manage errors that arise during a program's execution. If the user doesn't handle the exception, the program will halt and show a traceback, which includes a report of the exception that was raised. However, if you still wants the program continue to run, you have to write the code that handles the exception.

- 在Python中，使用try-except 块来处理异常。

  Exceptions are handled with try-except blocks in Python.

异常
Exceptions

- try-except块在指示Python做某事同时，也告诉 Python 如果引发异常应该做什么。当用户使用 try-except 块时，即使出现问题，编写的程序也会继续运行。并且代码使用者将看到特意编写的友好错误消息，而不是可能使用户难以阅读的回溯。

  While a try-except block asks Python to do something, it will simultaneously tell Python what to do if an exception is raised. When users use try-except blocks, the programs will continue running even if things start to go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you write.

- 通常，读者在try语句块中指示要做的事，用except语句捕获异常，并决定如何处理。用户可以什么也不做，或者接着抛出异常。下面程序展示了除零错误的处理。

  Usually, a try block instruct Python to do specific things, except statement captures the exception, and the following process. The user can choose to do nothing or continue to raise further exception. The code below illustrates how to process dividing by 0 errors.

异常
Exceptions

```
C:\Users\yrming\Desktop\备课\Python程序设计>cat lec_6_exception.py
print("Without try-catch block guard:")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break

    answer = float(first_number) / float(second_number)
    print(answer)


print("With try-catch block guard:")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break

    try:
        answer = float(first_number) / float(second_number)
    except ZeroDivisionError:
        print("You can't divide by 0!")
    else:
        print(answer)
```

异常
Exceptions

```
C:\Users\yrming\Desktop\备课\Python程序设计>python lec_6_exception.py
Without try-catch block guard:
Enter 'q' to quit.

First number: 1.0
Second number: 0.0
Traceback (most recent call last):
  File "lec_6_exception.py", line 11, in <module>
    answer = float(first_number) / float(second_number)
ZeroDivisionError: float division by zero

C:\Users\yrming\Desktop\备课\Python程序设计>python lec_6_exception.py
Without try-catch block guard:
Enter 'q' to quit.

First number: q
With try-catch block guard:
Enter 'q' to quit.

First number: 1.0
Second number: 0.0
You can't divide by 0!

First number: 1.0
Second number: 2.0
0.5

First number:
```

异常
Exceptions

- 从上面的例子可以看出，我们可以通过将可能产生错误的代码包裹在try-except块中来使该程序更具鲁棒性。另外，此示例还包括一个else块。任何依赖于try块成功执行的代码都进入else块：在本例中，如果除法操作成功，我们使用else块打印结果。except块告诉Python在出现ZeroDivisionError时如何响应。如果try块由于被零除错误而没有成功，我们会打印一条友好的消息，告诉用户如何避免这种错误。程序继续运行，用户则不会看到晦涩难懂的栈回溯。

  Exemplified by the program, we can see that program can be more error-resistant by wrapping the line that might produce errors in a try-except block. This example also includes an else block. Any code that depends on the try block executing successfully goes in the else block: in this case if the division operation is successful, we use the else block to print the result w. The except block tells Python how to respond when a ZeroDivisionError arises. If the try block doesn’t succeed because of a division by zero error, we print a friendly message telling the user how to avoid this kind of error. The program continues to run, and the user never sees a traceback sometimes hard to understand.

异常
Exceptions

- 在文件操作中，比较常见的错误是要操作的文件不存在，下面例子展示了如何处理这种异常：

  One common mistake in file processing is the targeted file does not exist. The following code demonstrates how to process such exception:

```
>>> filename = 'alice.txt'
>>>
>>> try:
...     with open(filename, encoding='utf-8') as f:
...         contents = f.read()
... except FileNotFoundError:
...     print(f"Sorry, the file {filename} does not exist.")
...
Sorry, the file alice.txt does not exist.
```

异常
Exceptions

- 在异常处理中，如果不论是否发生异常，特定语句必须执行，则可将其放到finally语句块中，即完整的异常处理具有如下形式：

  During exception handling, if specific statements must be executed regardless of whether an exception occurs or not, it can be placed into a finally block. The most complete form of exception handling is as follows:

```
try:
        # Some Code....

except:
        # optional block
        # Handling of exception (if required)

else:
        # execute if no exception

finally:
      # Some code .....(always executed)
```

异常
Exceptions

- 关键字raise用于抛出异常，特别是用户不能处理时，其常见如法如右所示：

  The keyword raise is used to throw out exceptions, especially the exception cannot be handled by the user. There are generally two forms of use, listed as right:

```
if something:
    raise Exception('My error!')
```

```
try:
  generate_exception()
except SomeException as e:
  if not can_handle(e):
    raise
  handle_exception(e)
```