

# Chapter 6 (second part): Files and Exceptions

These slides have been prepared based on the book titled "Starting Out with Python" by Tony Gaddis.

Lecturer: F. Kuzey Edes-Huyal

Bahcesehir University

November 21, 2023

## Using Python's for Loop to Read Lines

---

The Python language also allows you to write a **for loop** that automatically reads the lines in a file without testing for any special condition that signals the end of the file. The loop does not require a priming read operation, and it automatically stops when the end of the file has been reached. When you simply want to read the lines in a file, one after the other, this technique is simpler and **more elegant** than writing a **while loop** that explicitly tests for an end of the file condition. Here is the general format of the loop:

```
for variable in file_object:
```

```
    statement
```

```
    statement
```

```
    etc.
```

In the general format, `variable` is the name of a variable, and `file_object` is a variable that references a file object. The loop will iterate once for each line in the file. The first time the loop iterates, `variable` will reference the first line in the file (as a string), the second time the loop iterates, `variable` will reference the second line, and so forth. Program 6-10 provides a demonstration. It reads and displays all of the items in the `sales.txt` file.

### Program 6-10 (read\_sales2.py)

```
1 # This program uses the for loop to read
2 # all of the values in the sales.txt file.
3
4 def main():
5     # Open the sales.txt file for reading.
6     sales_file = open('sales.txt', 'r')
7
8     # Read all the lines from the file.
9     for line in sales_file:
10         # Convert line to a float.
11         amount = float(line)
12         # Format and display the amount.
13         print(f'{amount:.2f}')
14
15     # Close the file.
16     sales_file.close()
17
18 # Call the main function.
19 if __name__ == '__main__':
20     main()
```

### Program Output

```
1000.00
2000.00
3000.00
4000.00
5000.00
```

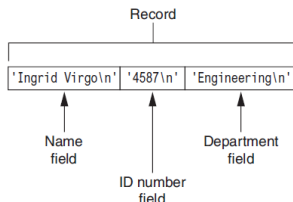
## Processing Records

---

When data is written to a file, it is often organized into records and fields. A **record** is a complete set of data that describes one item, and a **field** is a single piece of data within a record. For example, suppose we want to store data about employees in a file. The file will contain a record for each employee. Each record will be a collection of fields, such as name, ID number, and department. This is illustrated in Figure 6-18.

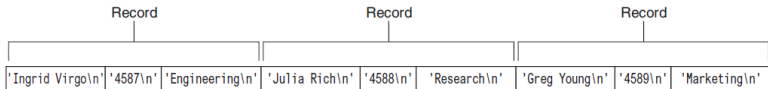
**Figure 6-18** Fields in a record

---



Each time you write a record to a sequential access file, you write the fields that make up the record, one after the other. For example, Figure 6-19 shows a file that contains three employee records. Each record consists of the employee's name, ID number, and department.

**Figure 6-19** Records in a file



```

1 # This program gets employee data from the user and
2 # saves it as records in the employee.txt file.
3
4 def main():
5     # Get the number of employee records to create.
6     num_emps = int(input('How many employee records ' +
7                           'do you want to create? '))
8
9     # Open a file for writing.
10    emp_file = open('employees.txt', 'w')
11
12    # Get each employee's data and write it to the file.
13    for count in range(1, num_emps + 1):
14        # Get the data for an employee.
15        print(f'Enter data for employee #{count}')
16        name = input('Name: ')
17        id_num = input('ID number: ')
18        dept = input('Department: ')
19
20        # Write the data as a record to the file.
21        emp_file.write(f'{name}\n')

```

(program continues)

## Chapter 6 Files and Exceptions

### Program 6-13 (continued)

```

22    emp_file.write(f'{id_num}\n')
23    emp_file.write(f'{dept}\n')
24
25    # Display a blank line.
26    print()
27
28    # Close the file.
29    emp_file.close()
30    print('Employee records written to employees.txt.')
31
32 # Call the main function.
33 if __name__ == '__main__':
34     main()

```

### Program Output (with input shown in bold)

```

How many employee records do you want to create? 3 Enter
Enter the data for employee #1
Name: Ingrid Virgo Enter
ID number: 4587 Enter
Department: Engineering Enter

Enter the data for employee #2
Name: Julia Rich Enter
ID number: 4588 Enter
Department: Research Enter

Enter the data for employee #3
Name: Greg Young Enter
ID number: 4589 Enter
Department: Marketing Enter

Employee records written to employees.txt.

```

## Adding and Displaying Records

---

Midnight Coffee Roasters, Inc. is a small company that imports raw coffee beans from around the world and roasts them to create a variety of gourmet coffees. Julie, the owner of the company, has asked you to write a series of programs that she can use to manage her inventory. After speaking with her, you have determined that a file is needed to keep inventory records. Each record should have two fields to hold the following data:

- Description. A string containing the name of the coffee
- Quantity in inventory. The number of pounds in inventory, as a floating-point number

Your first job is to write a program that can be used to add records to the file. Program 6-15 shows the code. Note the output file is opened in append mode. Each time the program is executed, the new records will be added to the file's existing contents.



**Program 6-15** (add\_coffee\_record.py)

```
1 # This program adds coffee inventory records to
2 # the coffee.txt file.
3
4 def main():
5     # Create a variable to control the loop.
6     another = 'y'
7
8     # Open the coffee.txt file in append mode.
9     coffee_file = open('coffee.txt', 'a')
10
11     # Add records to the file.
12     while another == 'y' or another == 'Y':
13         # Get the coffee record data.
14         print('Enter the following coffee data:')
15         descr = input('Description: ')
```

6.3 Pr

```
16         qty = int(input('Quantity (in pounds): '))
17
18         # Append the data to the file.
19         coffee_file.write(descr + '\n')
20         coffee_file.write(f'{qty}\n')
21
22         # Determine whether the user wants to add
23         # another record to the file.
24         print('Do you want to add another record?')
25         another = input('Y = yes, anything else = no: ')
26
27     # Close the file.
28     coffee_file.close()
29     print('Data appended to coffee.txt.')
30
31 # Call the main function.
32 if __name__ == '__main__':
33     main()
```

### Program Output (with input shown in bold)

Enter the following coffee data:

Description: **Brazilian Dark Roast**

Quantity (in pounds): **18**

Do you want to enter another record?

Y = yes, anything else = no: **y**

Enter the following coffee data:

Description: **Sumatra Medium Roast**

Quantity (in pounds): **25**

Do you want to enter another record?

Y = yes, anything else = no: **n**

Data appended to coffee.txt.

Your next job is to write a program that displays all of the records in the inventory file. Program 6-16 shows the code.

```

3
4 def main():
5     # Open the coffee.txt file.
6     coffee_file = open('coffee.txt', 'r')
7
8     # Read the first record's description field.
9     descr = coffee_file.readline()
10
11    # Read the rest of the file.

```

*(program continues)*

## Chapter 6 Files and Exceptions

### Program 6-16 *(continued)*

```

12    while descr != '':
13        # Read the quantity field.
14        qty = float(coffee_file.readline())
15
16        # Strip the \n from the description.
17        descr = descr.rstrip('\n')
18
19        # Display the record.
20        print(f'Description: {descr}')
21        print(f'Quantity: {qty}')
22
23        # Read the next description.
24        descr = coffee_file.readline()
25
26    # Close the file.
27    coffee_file.close()
28
29    # Call the main function.
30    if __name__ == '__main__':
31        main()

```

### Program Output

```

Description: Brazilian Dark Roast
Quantity: 18.0
Description: Sumatra Medium Roast
Quantity: 25.0

```

An exception is an error that occurs while a program is running. In most cases, an exception causes a program to abruptly halt. For example, look at Program 6-20. This program gets two numbers from the user then divides the first number by the second number. In the sample running of the program, however, an exception occurred because the user entered 0 as the second number. (Division by 0 causes an exception because it is mathematically impossible.)

**Program 6-20** (division.py)

```
1 # This program divides a number by another number.
2
3 def main():
4     # Get two numbers.
5     num1 = int(input('Enter a number: '))
6     num2 = int(input('Enter another number: '))
7
8     # Divide num1 by num2 and display the result.
9     result = num1 / num2
10    print(f'{num1} divided by {num2} is {result}')
11
12 # Call the main function.
13 if __name__ == '__main__':
14     main()
```

**Program Output** (with input shown in bold)

Enter a number: **10**   
Enter another number: **0**

(program continues)

## Chapter 6 Files and Exceptions

**Program 6-20** (continued)

```
Traceback (most recent call last):
  File "C:\Python\division.py," line 13, in <module>
    main()
  File "C:\Python\division.py," line 9, in main
    result = num1 / num2
ZeroDivisionError: integer division or modulo by zero
```

The lengthy error message that is shown in the sample run is called a **traceback**. The traceback gives information regarding the line number(s) that caused the exception. (When an exception occurs, programmers say that an exception was raised.) The last line of the error message shows the name of the **exception** that was raised (`ZeroDivisionError`) and a description of the error that caused the exception to be raised (integer division or modulo by zero). You can prevent many exceptions from being raised by carefully coding your program. For example, Program 6-21 shows how division by 0 can be prevented with a simple if statement. Rather than allowing the exception to be raised, the program tests the value of `num2`, and displays an error message if the value is 0. This is an example of gracefully avoiding an exception.

**Program 6-21** (division2.py)

```
1 # This program divides a number by another number.
2
3 def main():
4     # Get two numbers.
5     num1 = int(input('Enter a number: '))
6     num2 = int(input('Enter another number: '))
7
8     # If num2 is not 0, divide num1 by num2
9     # and display the result.
10    if num2 != 0:
11        result = num1 / num2
12        print(f'{num1} divided by {num2} is {result}')
13    else:
14        print('Cannot divide by zero.')
15
16 # Call the main function.
17 if __name__ == '__main__':
18     main()
```

**Program Output** (with input shown in bold)

```
Enter a number: 10 Enter
Enter another number: 0 Enter
Cannot divide by zero.
```



### Program 6-22 (gross\_pay1.py)

```
1 # This program calculates gross pay.
2
3 def main():
4     # Get the number of hours worked.
5     hours = int(input('How many hours did you work? '))
6
7     # Get the hourly pay rate.
8     pay_rate = float(input('Enter your hourly pay rate: '))
9
10    # Calculate the gross pay.
11    gross_pay = hours * pay_rate
12
13    # Display the gross pay.
14    print(f'Gross pay: ${gross_pay:,.2f}')
15
16 # Call the main function.
17 if __name__ == '__main__':
18     main()
```

### Program Output (with input shown in bold)

How many hours did you work? **forty**

Traceback (most recent call last):

```
File "C:\Users\Tony\Documents\Python\Source
Code\Chapter 06\gross_pay1.py", line 17, in <module>
    main()
```

```
File "C:\Users\Tony\Documents\Python\Source
Code\Chapter 06\gross_pay1.py", line 5, in main
    hours = int(input('How many hours did you work? '))
ValueError: invalid literal for int() with base 10: 'forty'
```

Look at the sample running of the program. An exception occurred because the user entered the string 'forty' instead of the number 40 when prompted for the number of hours worked. Because the string 'forty' cannot be converted to an integer, the `int()` function raised an exception in line 5, and the program halted. Look carefully at the last line of the traceback message, and you will see that the name of the exception is `ValueError`, and its description is: `invalid literal for int() with base 10: 'forty'`.

Python, like most modern programming languages, allows you to write code that responds to exceptions when they are raised, and prevents the program from abruptly crashing. Such code is called an exception handler and is written with the `try/except` statement.

There are several ways to write a try/except statement, but the following general format shows the simplest variation:

```
try:
```

```
    statement
```

```
    statement
```

```
    etc.
```

```
except ExceptionName:
```

```
    statement
```

```
    statement
```

```
    etc.
```

First, the keyword `try` appears, followed by a colon. Next, a code block appears which we will refer to as the try suite. The try suite is one or more statements that can potentially raise an exception. After the try suite, an `except` clause appears. The `except` clause begins with the keyword `except`, optionally followed by the name of an exception, and ending with a colon. Beginning on the next line is a block of statements that we will refer to as a handler.

When the try/except statement executes, the statements in the try suite begin to execute. The following describes what happens next:

- If a statement in the try suite raises an exception that is specified by the `ExceptionName` in an except clause, then the handler that immediately follows the except clause executes. Then, the program resumes execution with the statement immediately following the try/except statement.
- If a statement in the try suite raises an exception that is not specified by the `ExceptionName` in an except clause, then the program will halt with a traceback error message.
- If the statements in the try suite execute without raising an exception, then any except clauses and handlers in the statement are skipped, and the program resumes execution with the statement immediately following the try/except statement. Program 6-23 shows how we can write a try/except statement to gracefully respond to a `ValueError` exception.

**Program 6-23** (gross\_pay2.py)

```
1 # This program calculates gross pay.
2
3 def main():
4     try:
5         # Get the number of hours worked.
6         hours = int(input('How many hours did you work? '))
7
8         # Get the hourly pay rate.
9         pay_rate = float(input('Enter your hourly pay rate: '))
10
11        # Calculate the gross pay.
12        gross_pay = hours * pay_rate
13
14        # Display the gross pay.
```

## 6.4 Exceptions

```
15        print(f'Gross pay: ${gross_pay:.2f}')
16    except ValueError:
17        print('ERROR: Hours worked and hourly pay rate must')
18        print('be valid numbers.')
19
20 # Call the main function.
21 if __name__ == '__main__':
22     main()
```

**Program Output** (with input shown in bold)

```
How many hours did you work? forty Enter
ERROR: Hours worked and hourly pay rate must
be valid numbers.
```

**Figure 6-20** Handling an exception

```
# This program calculates gross pay.

def main():
    try:
        # Get the number of hours worked.
        hours = int(input('How many hours did you work? '))

        # Get the hourly pay rate.
        pay_rate = float(input('Enter your hourly pay rate: '))

        # Calculate the gross pay.
        gross_pay = hours * pay_rate

        # Display the gross pay.
        print(f'Gross pay: ${gross_pay:,.2f}')
    except ValueError:
        print('ERROR: Hours worked and hourly pay rate must')
        print('be valid integers.')

# Call the main function.
if __name__ == '__main__':
    main()
```

If this statement raises a `ValueError` exception...

The program jumps to the `except ValueError` clause and executes its handler.

Let's look at another example in Program 6-24. This program, which does not use exception handling, gets the name of a file from the user then displays the contents of the file. The program works as long as the user enters the name of an existing file. An exception will be raised, however, if the file specified by the user does not exist. This is what happened in the sample run.

### Program 6-24 (display\_file.py)

```
1 # This program displays the contents
2 # of a file.
3
4 def main():
5     # Get the name of a file.
6     filename = input('Enter a filename: ')
7
8     # Open the file.
9     infile = open(filename, 'r')
10
11     # Read the file's contents.
12     contents = infile.read()
13
14     # Display the file's contents.
15     print(contents)
16
17     # Close the file.
18     infile.close()
19
20 # Call the main function.
21 if __name__ == '__main__':
22     main()
```

### Program Output (with input shown in bold)

```
Enter a filename: bad_file.txt Enter
Traceback (most recent call last):
File "C:\Python\display_file.py," line 21, in <module>
main()
File "C:\Python\display_file.py," line 9, in main
infile = open(filename, 'r')
IOError: [Errno 2] No such file or directory: 'bad_file.txt'
```



The statement in line 9 raised the exception when it called the open function. Notice in the traceback error message that the name of the exception that occurred is `IOError`. This is an exception that is raised when a file I/O operation fails. You can see in the traceback message that the cause of the error was No such file or directory: 'bad\_file.txt'.

Program 6-25 shows how we can modify Program 6-24 with a try/except statement that gracefully responds to an `IOError` exception. In the sample run, assume the file `bad_file.txt` does not exist.

**Program 6-25** (display\_file2.py)

```
1 # This program displays the contents
2 # of a file.
3
4 def main():
5     # Get the name of a file.
6     filename = input('Enter a filename: ')
7
8     try:
9         # Open the file.
10        infile = open(filename, 'r')
11
12        # Read the file's contents.
13        contents = infile.read()
14
15        # Display the file's contents.
16        print(contents)
17
18        # Close the file.
19        infile.close()
20    except IOError:
21        print('An error occurred trying to read')
22        print('the file', filename)
23
24 # Call the main function.
25 if __name__ == '__main__':
26     main()
```

**Program Output** (with input shown in bold)

Enter a filename: **bad\_file.txt**   
An error occurred trying to read  
the file bad\_file.txt

## Handling Multiple Exceptions

---

In many cases, the code in a try suite will be capable of throwing more than one type of exception. In such a case, you need to write an except clause for each type of exception that you want to handle. For example, Program 6-26 reads the contents of a file named `sales_data.txt`. Each line in the file contains the sales amount for one month, and the file has several lines. Here are the contents of the file:

24987.62

26978.97

32589.45

31978.47

22781.76

29871.44

Program 6-26 reads all of the numbers from the file and adds them to an accumulator variable.

**Program 6-26** (sales\_report1.py)

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10        infile = open('sales_data.txt', 'r')
11
12        # Read the values from the file and
13        # accumulate them.
14        for line in infile:
15            amount = float(line)
16            total += amount
17
18        # Close the file.
19        infile.close()
20
21        # Print the total.
22        print(f'total:,.2f')
23
24    except IOError:
25        print('An error occurred trying to read the file.')
26
27    except ValueError:
28        print('Non-numeric data found in the file.')
29
30    except:
31        print('An error occurred.')
32
33 # Call the main function.
34 if __name__ == '__main__':
35     main()
```

The try suite contains code that can raise different types of exceptions. For example:

- The statement in line 10 can raise an `IOError` exception if the `sales_data.txt` file does not exist. The for loop in line 14 can also raise an `IOError` exception if it encounters a problem reading data from the file.
- The `float` function in line 15 can raise a `ValueError` exception if the line variable references a string that cannot be converted to a floating-point number (an alphabetic string, for example).

Notice the `try/except` statement has three `except` clauses:

- The `except` clause in line 24 specifies the `IOError` exception. Its handler in line 25 will execute if an `IOError` exception is raised.
- The `except` clause in line 27 specifies the `ValueError` exception. Its handler in line 28 will execute if a `ValueError` exception is raised.
- The `except` clause in line 30 does not list a specific exception. Its handler in line 31 will execute if an exception that is not handled by the other `except` clauses is raised.

## Using One except Clause to Catch All Exceptions

---

The previous example demonstrated how multiple types of exceptions can be handled individually in a `try/except` statement.

Sometimes you might want to write a `try/except` statement that simply catches any exception that is raised in the `try` suite and, regardless of the exception's type, responds the same way. You can accomplish that in a `try/except` statement by writing one `except` clause that does not specify a particular type of exception. Program 6-27 shows an example.

**Program 6-27** (sales\_report2.py)

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10        infile = open('sales_data.txt', 'r')
11
12        # Read the values from the file and
13        # accumulate them.
14        for line in infile:
```

(program continues)

---

Chapter 6 Files and Exceptions

**Program 6-27** (continued)

```
15         amount = float(line)
16         total += amount
17
18         # Close the file.
19         infile.close()
20
21         # Print the total.
22         print(f'{total:,.2f}')
23     except:
24         print('An error occurred.')
25
26 # Call the main function.
27 if __name__ == '__main__':
28     main()
```



## Displaying an Exception's Default Error Message

---

When an exception is thrown, an object known as an exception object is created in memory. The exception object usually contains a default error message pertaining to the exception. (In fact, it is the same error message that you see displayed at the end of a traceback when an exception goes unhandled.) When you write an `except` clause, you can optionally assign the exception object to a variable, as shown here:

```
except ValueError as err:
```

This `except` clause catches `ValueError` exceptions. The expression that appears after the `except` clause specifies that we are assigning the exception object to the variable `err`. (There is nothing special about the name `err`. That is simply the name that we have chosen for the examples. You can use any name that you wish.) After doing this, in the exception handler you can pass the `err` variable to the `print` function to display the default error message that Python

---

provides for that type of error. Program 6-28 shows an example of how this is done.


**Program 6-28** (gross\_pay3.py)

```
1 # This program calculates gross pay.
2
3 def main():
4     try:
5         # Get the number of hours worked.
6         hours = int(input('How many hours did you work? '))
7
```

## 6.4 Exceptions

```
8         # Get the hourly pay rate.
9         pay_rate = float(input('Enter your hourly pay rate: '))
10
11         # Calculate the gross pay.
12         gross_pay = hours * pay_rate
13
14         # Display the gross pay.
15         print(f'Gross pay: ${gross_pay:,.2f}')
16     except ValueError as err:
17         print(err)
18
19 # Call the main function.
20 if __name__ == '__main__':
21     main()
```

**Program Output** (with input shown in bold)

How many hours did you work? **forty**   
Invalid literal for int() with base 10: 'forty'

## The else Clause

---

The try/except statement may have an optional else clause, which appears after all the except clauses. Here is the general format of a try/except statement with an else clause:

```
try:
```

```
    statement
```

```
    statement
```

```
    etc.
```

```
except ExceptionName:
```

```
    statement
```

```
    statement
```

```
    etc.
```

```
else:
```

```
    statement
```

```
    statement
```

```
    etc.
```

The block of statements that appears after the else clause is known as the else suite. The statements in the else suite are executed after the statements in the try suite, only if no exceptions were raised. If an exception is raised, the else suite is skipped. Program 6-30 shows an example.

**Program 6-30** (sales\_report4.py)

```
1 # This program displays the total of the
2 # amounts in the sales_data.txt file.
3
4 def main():
5     # Initialize an accumulator.
6     total = 0.0
7
8     try:
9         # Open the sales_data.txt file.
10         infile = open('sales_data.txt', 'r')
```

## 6.4 Exceptions

```
11
12         # Read the values from the file and
13         # accumulate them.
14         for line in infile:
15             amount = float(line)
16             total += amount
17
18         # Close the file.
19         infile.close()
20     except Exception as err:
21         print(err)
22     else:
23         # Print the total.
24         print(f'total:,.2f')
25
26 # Call the main function.
27 if __name__ == '__main__':
28     main()
```

## The finally Clause

---

The try/except statement may have an optional finally clause, which must appear after all the except clauses. Here is the general format of a try/except statement with a finally clause:

```
try:
    statement
    statement
    etc.
except ExceptionName:
    statement
    statement
    etc.
finally:
    statement
    statement
    etc.
```

The block of statements that appears after the finally clause is known as the finally suite. The statements in the finally suite are always executed after the try suite has executed, and after any exception handlers have executed. The statements in the finally suite execute whether an exception occurs or not. The purpose of the finally suite is to perform cleanup operations, such as closing files or other resources. Any code that is written in the finally suite will always execute, even if the try suite raises an exception.

## Possible quiz questions after the midterm

1. What does "object-oriented programming" refer to?
2. What are the reasons for the necessity of object-oriented programming?



Kevin is a freelance video producer who makes TV commercials for local businesses. When he makes a commercial, he usually films several short videos. Later, he puts these short videos together to make the final commercial. He has asked you to write the following two programs.

1. A program that allows him to enter the running time (in seconds) of each short video in a project. The running times are saved to a file.