Computer Programming I

Binnur Kurt, PhD

BAU Bahçeşehir University

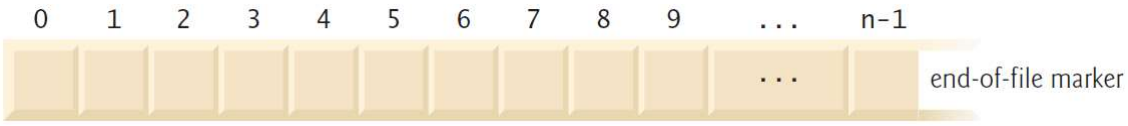
binnur.kurt@rc.bau.edu.tr

MODULE 9

EXCEPTIONS AND FILES

Files

- Python views a text file as a sequence of characters and a binary file (for images, videos and more) as a sequence of bytes.
- As in lists and arrays, the first character in a text file and byte in a binary file is located at position 0, so in a file of n characters or bytes, the highest position number is n – 1



End of File

- Every operating system provides a mechanism to denote the end of a file.
- Some represent it with an end-of-file marker, while others might maintain a count of the total characters or bytes in the file.
- Programming languages generally hide these operating-system details from you.

Standard File Objects

- When a Python program begins execution, it creates three standard file objects:
  - `sys.stdin`
    - the standard input file object
  - `sys.stdout`
    - the standard output file object, and
  - `sys.stderr`
    - the standard error file object.
- They do not read from or write to files by default

Writing to a Text File

- Many applications ***acquire*** resources, such as files, network connections, database connections and more.
- You should ***release*** resources as soon as they're no longer needed.
- This practice ensures that other applications can use the resources.
- Python's with `statement`:
  - acquires a resource and assigns its corresponding object to a variable
  - allows the application to use the resource via that variable
  - calls the resource object's `close` method to release the resource when program control reaches the end of the `with` statement's suite

Writing to a Text File

```
with open('accounts.txt', mode='w') as accounts:
    accounts.write('100, Jack Bauer, 10000.0\n')
    accounts.write('200, Kate Austen, 20000.0\n')
    accounts.write('300, James Sawyer, 30000.0\n')
    accounts.write('400, Ben Linus, 40000.0\n')
    accounts.write('500, Sun Kwon, 50000.0\n')
```

> The built-in **open** function opens the file **accounts.txt** and associates it with a file object.

> The mode argument specifies the file-open mode, indicating whether to open a file for reading from the file, for writing to the file or both.

> The mode **'w'** opens the file for writing, creating the file if it does not exist.

> If you do not specify a path to the file, Python creates it in the current folder

Reading from a Text File

```
with open('accounts.txt', mode='r') as accounts:
    print(f'{"Account":<10}{"Full Name":<20}{"Balance":>10}')
    for record in accounts:
        account, name, balance = record.split(',')
        print(f'{account:<10}{name.strip():<20}{balance.strip():>10}')
```

```
Account   Full Name           Balance
100       Jack Bauer          10000.0
200       Kate Austen         20000.0
300       James Sawyer        30000.0
400       Ben Linus           40000.0
500       Sun Kwon            50000.0
```

File open modes

Mode	Description
`'r'`	Open a text file for reading. This is the default if you do not specify the file-open mode when you call open.
`'w'`	Open a text file for writing. Existing file contents are deleted.
`'a'`	Open a text file for appending at the end, creating the file if it does not exist. New data is written at the end of the file.
`'r+'`	Open a text file reading and writing.
`'w+'`	Open a text file reading and writing. Existing file contents are deleted.
`'a+'`	Open a text file reading and appending at the end. New data is written at the end of the file. If the file does not exist, it is created.

File Method `readlines`

- The file object's **`readlines`** method also can be used to read an entire text file.
- The method returns each line as a string in a list of strings.
- For small files, this works well, but iterating over the lines in a file object can be more efficient.

File Method **readlines**

```
accounts = open('accounts.txt', mode='r')
print(f'{"Account":<10}{"Full Name":<20}{"Balance":>10}')
for line in accounts.readlines():
    account, name, balance = line.split(',')
    print(f'{account:<10}{name.strip():<20}{balance.strip():>10}')
accounts.close()
```

```
Account   Full Name              Balance
100       Jack Bauer             10000.0
200       Kate Austen            20000.0
300       James Sawyer           30000.0
400       Ben Linus              40000.0
500       Sun Kwon               50000.0
```

Seeking to a Specific File Position

- While reading through a file, the system maintains a file-position pointer representing the location of the next character to read.
- Sometimes it's necessary to process a file sequentially from the beginning several times during a program's execution.
- Each time, you must reposition the file-position pointer to the beginning of the file, which you can do either by
  - closing and reopening the file
  - calling the file object's seek method, as in
    - `file_object.seek(0)`

Updating Text Files

> Formatted data written to a text file cannot be modified without the risk of destroying other data.

> Copy and update to a temporary file

```
accounts = open('accounts.txt', 'r')
```

```
temp_file = open('temp_file.txt', 'w')
```

```
with accounts, temp_file:
    for account in accounts:
        iban, fullname, balance = account.split(',')
        if  iban.strip() != "300":
            temp_file.write(account)
        else:
            new_account = f'{iban.strip()}, James Sawyer Ford, {balance.strip()}\n'
            temp_file.write(new_account)
```

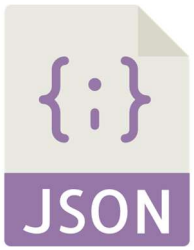
os Module File-Processing Functions

> The **os** module provides functions for interacting with the operating system, including several that manipulate your system's files and directories.

```
import os
```

```
os.remove("accounts.txt")
```

```
os.rename("temp_file.txt", "accounts.txt")
```



Serialization with JSON

- JSON (JavaScript Object Notation) is a text-based, human-and-computer-readable, data-interchange format used to represent objects (such as dictionaries, lists and more) as collections of name–value pairs.
- JSON can even represent objects
- JSON has become the preferred data format for transmitting objects across platforms.
- This is especially true for invoking cloud-based web services, which are functions and methods that you call over the Internet.

JSON Data Format

> JSON objects are like Python dictionaries.

> Each JSON object contains a comma-separated list of property names and values, in curly braces

```
{
  "iban": "TR1",
  "fullname": "Jack Bauer",
  "balance": 10000.0
}
```

> JSON also supports arrays which, like Python lists, are comma-separated values in square brackets.

```
[100, 200, 300]
```

JSON Data Format

> Values in JSON objects and arrays can be:
  - strings in ***double quotes*** (like "Jack"),
  - numbers (like 100 or 24.98),
  - JSON Boolean values (represented as true or false in JSON),
  - **`null`** (to represent no value, like **None** in Python),
  - arrays (like [100, 200, 300]), and
  - other JSON objects.

Python Standard Library Module **`json`**

> The **`json`** module enables you to convert objects to JSON (JavaScript Object Notation) text format.

> This is known as ***serializing*** the data.

```
import json
```

```
accounts = { 'accounts': [
    { "iban": "TR1", "fullname": "kate austen", "balance" : 100000.},
    { "iban": "TR2", "fullname": "jack shephard", "balance" : 200000.},
    { "iban": "TR3", "fullname": "ben linus", "balance" : 300000.}
]}
```

```
with open("accounts.json", "w") as json_file:
    json.dump(accounts, json_file)
```

Deserializing the JSON Text

> The json module's load function reads the entire JSON contents of its file object argument and converts the JSON into a Python object.

> This is known as ***deserializing*** the data.

```
with open("accounts.json", "r") as json_file:
    accounts = json.load(json_file)
```

```
accounts
```

```
{'accounts': [{'iban': 'TR1', 'fullname': 'kate austen', 'balance': 100000.0},
  {'iban': 'TR2', 'fullname': 'jack shephard', 'balance': 200000.0},
  {'iban': 'TR3', 'fullname': 'ben linus', 'balance': 300000.0}]}
```

Displaying the JSON Text

- The json module's **dumps function** (dumps is short for "dump string") returns a Python string representation of an object in JSON format.
- Using dumps with load, you can read the JSON from the file and display it in a nicely indented format—sometimes called "pretty printing" the JSON.
- When the **`dumps`** function call includes the **`indent`** keyword argument, the string contains newline characters and indentation for pretty printing

```
with open("accounts.json", "r") as json_file:
    print(json.dumps(json.load(json_file),indent=4))
```

```
{
    "accounts": [
        {
            "iban": "TR1",
            "fullname": "kate austen",
            "balance": 100000.0
        },
        {
            "iban": "TR2",
            "fullname": "jack shephard",
            "balance": 200000.0
        },
        {
            "iban": "TR3",
            "fullname": "ben linus",
            "balance": 300000.0
        }
    ]
}
```

HANDLING EXCEPTIONS

File IO Exceptions

> Various types of exceptions can occur when you work with files

- **`FileNotFoundError`**
  - It occurs if you attempt to open a non-existent file for reading with the `'r'` or `'r+'` modes.
- **`PermissionsError`**
  - It occurs if you attempt an operation for which you do not have permission.
  - This might occur if you try to open a file that your account is not allowed to access or create a file in a folder where your account does not have permission to write, such as where your computer's operating system is stored.

File IO Exceptions

> Various types of exceptions can occur when you work with files
  - **`ValueError`**
    - The error message is **`'I/O operation on closed file.'`**
    - It occurs when you attempt to write to a file that has already been closed.

Division by Zero and Invalid Input

```
42 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-87-52cebea8b64f> in <module>
----> 1 42 / 0

ZeroDivisionError: division by zero
```

```
number = int(input('Enter an integer: '))
```

```
Enter an integer: forty two

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-88-aa06ae0a3438> in <module>
----> 1 number = int(input('Enter an integer: '))

ValueError: invalid literal for int() with base 10: 'forty two'
```

`try` Statements

> You need to handle these exceptions so that you can enable code to continue processing.

```
while True:
    try:
        number1 = int(input('Enter numerator: '))
        number2 = int(input('Enter denominator: '))
        result = number1 / number2
    except ValueError: # tried to convert non-numeric value to int
        print('You must enter two integers\n')
    except ZeroDivisionError: # denominator was 0
        print('Attempted to divide by zero\n')
    else: # executes only if no exceptions occur
        print(f'{number1:.3f} / {number2:.3f} = {result:.3f}')
        break # terminate the loop
```

```
while True:
    try:
        number1 = int(input('Enter numerator: '))
        number2 = int(input('Enter denominator: '))
        result = number1 / number2
    except ValueError: # tried to convert non-numeric value to int
        print('You must enter two integers\n')
    except ZeroDivisionError: # denominator was 0
        print('Attempted to divide by zero\n')
    else: # executes only if no exceptions occur
        print(f'{number1:.3f} / {number2:.3f} = {result:.3f}')
        break # terminate the loop
```

```
Enter numerator: 2
Enter denominator: 0
Attempted to divide by zero

Enter numerator: 2
Enter denominator: forty two
You must enter two integers

Enter numerator: 10
Enter denominator: 2
10.000 / 2.000 = 5.000
```

A suite of statements that might raise exceptions.

```
while True:
    try:
        number1 = int(input('Enter numerator: '))
        number2 = int(input('Enter denominator: '))
        result = number1 / number2
    except ValueError: # tried to convert non-numeric value to int
        print('You must enter two integers\n')
    except ZeroDivisionError: # denominator was 0
        print('Attempted to divide by zero\n')
    else: # executes only if no exceptions occur
        print(f'{number1:.3f} / {number2:.3f} = {result:.3f}')
        break # terminate the loop
```

```
while True:
    try:
        number1 = int(input('Enter numerator: '))
        number2 = int(input('Enter denominator: '))
        result = number1 / number2
    except ValueError: # tried to convert non-numeric value to int
        print('You must enter two integers\n')
    except ZeroDivisionError: # denominator was 0
        print('Attempted to divide by zero\n')
    else: # executes only if no exceptions occur
        print(f'{number1:.3f} / {number2:.3f} = {result:.3f}')
        break # terminate the loop
```

Exception handlers

The Catch-All Exception Handler

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except:
    print('Some other error occurred.')
```

This "except" statement is "bare": it does not refer to a specific exception.

This code provides a catch-all exception handler.

The Catch-All Exception Handler

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except:
    print('Some other error occurred.')
```

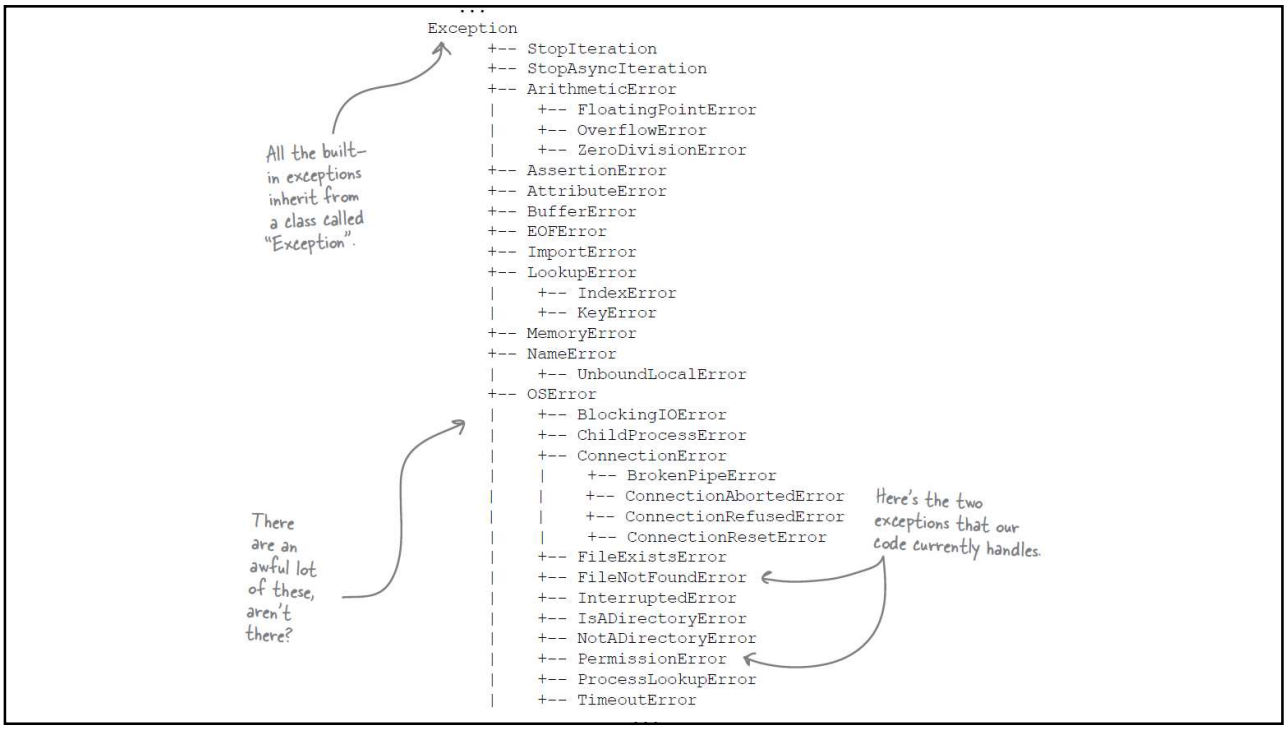
This code works, but doesn't really tell you much when some unexpected exception occurs.

The Catch-All Exception Handler

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except Exception as err:
    print('Some other error occurred:', str(err))
```

Unlike the "bare" except catch-all shown above, this one arranges for the exception object to be assigned to the "err" variable.

The value of "err" is then used as part of the friendly message (as it's always a good idea to report all exceptions).



Learning About Exceptions from “sys”

```
>>>
================================ RESTART: Shell ================================
>>>
>>> import sys
>>>
>>> try:
        1/0
except:
        err = sys.exc_info()
        for e in err:
            print(e)

<class 'ZeroDivisionError'>
division by zero
<traceback object at 0x105b22188>
>>>
```

Be sure to import the “sys” module.

Dividing by zero is *never* a good idea...and when your code divides by zero an exception occurs

Let’s extract and display the data associated with the currently occurring exception.

Here’s the data associated with the exception, which confirms that we have an issue with divide-by-zero.

Catching Multiple Exceptions in One except Clause

```
while True:
    try:
        number1 = int(input('Enter numerator: '))
        number2 = int(input('Enter denominator: '))
        result = number1 / number2
    except (ValueError, ZeroDivisionError):
        print('Try again!\n')
    else: # executes only if no exceptions occur
        print(f'{number1:.3f} / {number2:.3f} = {result:.3f}')
        break # terminate the loop
```

finally Clause

- Operating systems typically can prevent more than one program from manipulating a file at once.
- When a program finishes processing a file, the program should close it to **release** the resource.
  - This enables other programs to use the file
- Closing the file helps prevent a ***resource leak*** in which the file resource is not available to other programs because a program using the file never closes it.

finally Clause

- A try statement may have a `finally` clause as its last clause after any except clauses or else clause.
- The `finally` clause is guaranteed to execute, regardless of whether its try suite executes successfully, or an exception occurs

finally Clause

```
try:
    print('try suite with no exceptions raised')
except:
    print('this will not execute')
else:
    print('else executes because no exceptions in the try suite')
finally:
    print('finally always executes')
```

```
try suite with no exceptions raised
else executes because no exceptions in the try suite
finally always executes
```

finally Clause

```
try:
    print('try suite raises an exception')
    int("forty two")
    print('this will not execute')
except ValueError:
    print('a ValueError occurred')
else:
    print('else will not execute because an exception occurred')
finally:
    print('finally always executes')
```

```
try suite raises an exception
a ValueError occurred
finally always executes
```

finally Clause

```
def fun():
    try:
        return 42
    except:
        return 100
    finally:
        return 108
```

```
x = fun()
```

What is ***x***?

finally Clause

```
def fun():
    try:
        return 42
    except:
        return 100
    finally:
        return 108
```

```
x = fun()
```

```
x
```

```
108
```

Explicitly Raising an Exception

> The **`raise`** statement explicitly raises an exception.

> The simplest form of the raise statement is

`raise` *ExceptionClassName*

> The **`raise`** statement creates an object of the specified exception class.

Explicitly Raising an Exception

```
x = int(input('Enter an integer between 1-00'))
if not  ( 0 < x <= 100):
    raise ValueError(f"{x} is not between 1 and 100.")
```

```
Enter an integer between 1-000

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-107-278600a38605> in <module>
      1 x = int(input('Enter an integer between 1-00'))
      2 if not  ( 0 < x <= 100):
----> 3     raise ValueError(f"{x} is not between 1 and 100.")

ValueError: 0 is not between 1 and 100.
```