Handling Files

- Opening files using ‘with’ statement
- Pickling

Opening files using “*with*” statement

- Python’s with statement allows us not to worry about closing any opened files, as it automatically closes the opened file.

```
f = open(“sample.txt", "w")

f.write("Hello World!")

f.close()
```

```
with open(“sample.txt”, “w”) as file:

        file.write("Hello, World!")
```

Working With Two Files at the Same Time

```
f1 = open(“sample1.txt", “r")
data = f1.read()
f2 = open(“sample2.txt", "w")
f2.write(data)
f1.close()
f2.close()
```

- When we want to open more than one file at the same time, we can open as following:

```
fileFrom = “MyFile.png”
fileTo = “MyFileCopy.png”
with open(fileFrom, “rb”) as reader, open(fileTo, “wb”) as writer:
    data = reader.read()
    writer.write(data)
```

Pickling

- Recall that when we write to a text file, we need to convert other types to a string.

```
file.write(str(15))
file.write(str([1, 2, 3]))
```

- But when we read them back, the output will show us just strings. We will lose the types of those data.

```
print(file.read())
15[1, 2, 3]
```

Pickling

- To save data such as int, float, list, dict, etc., we can use a module called Pickle.
- The Pickle module is used for pickling (serializing) and unpickling (de-serializing) any Python object.
- To pickle objects we need to open a file in ‘b’ (binary) mode
- To save data in a file we use the dump() function:

  dump(dataObject, fileObject)
- To restore data from a file we use the load() function:

  load(fileObject)

Pickle - Example

```
import pickle
file =open(“students.dat”,”ab”)
while True:
    sid = int(input("Student ID: "))
    name = input("Student Name : ")
    grade = float(input("Grade: "))
    record = [sid, name, grade]
    pickle.dump(record,file)
    stop = input(“Enter more records (y/n)? ")
    if stop.lower() == “n”:
        break
print(“Size of file is {} bytes:”.format(file.tell()))
file.close()
```

```
with open(“students.dat","rb") as file:
while True:
   data = pickle.load(bfile)
   print(data)
```

Handling Exceptions

Errors

- Syntax errors are common errors in the code that cannot be interpreted.
  - Spelling mistakes – prit(x) -> print(x)
  - Missing out quotes – print(“Hello world) ->print(“Hello world”)
  - Missing out a colon - while True -> while True:
- Exceptions (or Runtime errors) are errors that occur during the program execution. It changes the normal flow of the program. Stops the program.
  - Reading a file, but file does not exist
  - Dividing by zero
  - Index out of range

```
age = int(input(age))
if nums[age]..
```

Handling Exceptions

We can handle exceptions using the try and except statements as follows.

```
try:
    statements
    statement that fails
    statements that are not executed
except:
    print('Something went wrong')
```

```
while True:
    try:
        num = int(input("Enter a number: "))
        break
    except:
        print(“An error occurred. Try again!”)
```

```
while True:
    try:
        num = int(input("Enter a number: "))
        break
    except Exception as e:
        print(“An error occurred. Try again!”, e)
```

Some Built-in Exceptions

- *IndexError:* Raised when an index of a sequence does not exist
    - print(myList[3]) # you have only 3 elements in myList (myList [0], myList [1], myList [2])
- *NameError:* Raised when a variable does not exist
    - print(x) # what is x? you did not define x
- *TypeError:* Raised when two different types are combined
    - x = 5; print(“Five “+x) # string cannot be concatenated to in*t*
- *ValueError:* Raised when there is a wrong value in a specified data type
    - int(x) # x should be convertible to integer, cannot be ‘asaasda’
- *ZeroDivisionError:* Raised when the second operator in a division is zero
    - 123/0 # you cannot a number by 0

https://www.w3schools.com/python/python_ref_exceptions.asp

Handling Exceptions

- You can define more than one exception block.

```
try:
   print (int(x))
except ValueError:
   print ("Not a number")
except NameError:
   print ("Variable x is not defined")
except:
   print (“An error occurred")
```

Optional *Else* block

- You can use the else block to define statements that should be executed if no errors are raised

```
try:
    a = int(input(“Enter a: ”))
    b = int(input(“Enter b: “))
    result = a/b
except (ValueError, ZeroDivisionError) as er:
    print (“Something went wrong!“, er)
else:
    print(“The quotient of values = ”, result)
```

Optional *finally* block

- You can use finally block to write statements that will be executed in any case – no matter if an error occurs or not.

```
try:
    file = open("Test.txt", "r")
    print(int(x))
except FileNotFoundError:
    print ("The file does not exist")
else:
    print("The statements in the try block did not raise any exception")
finally:
    file.close()
    print("This string will be printed in any case!")
```

Exception Handling

- General way for handling exceptions

```
try:
    Your code
except:
    Execute if exception occurs
else:
    Execute if everything is correct
finally:
    Execute always
```

Exception Handling

- Sometimes we can handle possible exceptions without try/except

```
try:
    a =int(input(“Enter 1st integer: “))
    b =int(input(“Enter 2nd integer: “))
    num = a/b
except ZeroDivisionError:
    print (“You cannot divide by 0")
except ValueError:
    print(“Values should be convertible to an integer")
finally:
    print("This string will be printed in any case!")
```

```
a =int(input(“Enter 1st integer: “))
b =int(input(“Enter 2nd integer: “))
if a.isdigit() and b.isdigit():
    if b == 0:
        print(“You cannot divide by 0")
    else:
        num = a/b
else:
    print("Values should be convertible to an integer”)
print("This string will be printed in any case!")
```