To apply any read or update operation to a file, we need a pointer or reference to the file. A file reference can be obtained by opening a file using the built-in open function. This function returns a reference to the `file` object, which is also known as a file handle in some literature. The minimum requirement with the open function is the name of the file with an absolute or relative path. One optional parameter is the access mode to indicate in which mode a file is to be opened. The access mode can be `read`, `write`, `append`, or others. A full list of access mode options is as follows:

- r: This option is for opening a file in read-only mode. This is a default option if the access mode option is not provided:

```
f = open ('abc.txt')
```

- a: This option is for opening a file to append a new line at the end of the file:

```
f = open ('abc.txt', 'a')
```

- w: This option is for opening a file for writing. If a file does not exist, it will create a new file. If the file exists, this option will override it and any existing contents in that file will be destroyed:

```
f = open ('abc.txt', 'w')
```

- x: This option is for opening a file for exclusive writing. If the file already exists, it will throw an error:

```
f = open ('abc.txt', 'x')
```

- t: This option is for opening a file in text mode. This is the default option.
- b: This option is for opening a file in binary mode.
- +: This option is for opening a file for reading and writing:

```
f = open ('abc.txt', 'r+'
```

The mode options can be combined to get multiple options. In addition to the filename and the access mode options, we can also pass the encoding type, especially for text files. Here is an example of opening a file with `utf-8`:

```
f = open("abc.txt", mode='r', encoding='utf-8')
```

• 'rb+' → reading & writing. Set file pointer at begining of binary file.

• 'ab' → appending. Creates if not exists.

• 'ab+' → appending & reading .

When we con … the

resources for other processes to use the file. A file can be closed by using the `close` method on the file instance or the file handle. Here is a code snippet showing the use of the `close` method:

```
file = open("abc.txt", 'r+w')
#operations on file

file.close()
```

Once a file is closed, the resources associated with the file instance and locks (if any) will be released by the operating system, which is a best practice in any programming language.

Reading and writing files

A file can be read by opening the file in access mode `r` and then using one of the read methods. Next, we summarize different methods available for read operations:

- `read(n)`: This method reads n characters from a file.
- `readline()`: This method returns one line from a file.
- `readlines()`: This method returns all lines from a file as a list.

Similarly, we can append or write to a file once it is opened in an appropriate access mode. The methods that are relevant to appending a file are as follows:

- `write (x)`: This method writes a string or a sequence of bytes to a file and returns the number of characters added to the file.
- `writelines (lines)`: This method writes a list of lines to a file.

In the next code example, we will create a new file, add a few text lines to it, and then read the text data using the read operations discussed previously:

```
#writereadfile.py: write to a file and then read from it
f1 = open("myfile.txt",'w')
f1.write("This is a sample file\n")
lines =["This is a test data\n", "in two lines\n"]
f1.writelines(lines)
f1.close()

f2 = open("myfile.txt",'r')
```

```
print(f2.read(4))
```

2023/05/15 10:47

```
print(f2.readline())
print(f2.readline())

f2.seek(0)
for line in f2.readlines():

    print(line)
f2.close()
```

In this code example, we write three lines to a file first. In the read operations, first, we read four characters, followed by reading two lines using the `readline` method. In the end, we move the pointer back to the top of the file using the `seek` method and access all lines in the file using the `readlines` method.

In the next section, we will see how the use of a context manager makes file handling convenient.

Using a context manager

Correct and fair usage of resources is critical in any programming language. A file handler and a database connection are a couple of many examples where it is a common practice to not release the resources on time after working with objects. If the resources are not released at all, then it will end up in a situation called **memory leakage** and may impact the system performance, and ultimately may result in the system crashing

Binary file

- To open use :— 'b'
- For reading :- 'rb'
- For writing :- 'wb'
- built in f^n bytearray() is used to convert normal data into binary before writing.
- 'wb+' → writing & reading. Creates a new binary file for writing.

2023/05/15 10:55

option. Sample code to transfer contents from one file to another file is as follows:

```
#multifilesread2.py

with open("1.txt",'r') as file1, open("3.txt",'w') as file2:
    for line in file1.readlines():
      file2.write(line)
```

Python also has a more elegant solution to operate on multiple files using the `fileinput` module. This module's input function can take a list of multiple files and then treat all such files as a single input. Sample code with two input files, `1.txt` and `2.txt`, and using the `fileinput` module is presented next:

```
#multifilesread1.py
import fileinput
with fileinput.input(files = ("1.txt",'2.txt')) as f:
    for line in f:

        print(f.filename())
        print(line)
```

With this approach, we get one file handle that operates on multiple files sequentially. Next, we will discuss error and exception handling in Python.

Handling errors and exceptions

There are many types of errors possible in Python. The most common one is related to the syntax of the program and is typically known as a **syntax error**. On many occasions, errors are reported during the execution of a program. Such errors are called **runtime errors**. The runtime errors that can be handled within our program are called **exceptions**. This section will focus on how to handle runtime errors or exceptions. Before jumping on to error handling, we will briefly introduce the most common runtime errors as follows:

- `IndexError`: This error occurs when a program tries to access an item at an invalid index (location in the memory).
- `ModuleNotFoundError`: This error will be thrown when a specified module is not found at the system path.
- `ZeroDivisionError`: This error is thrown when a program tries to divide a number by zero.

- `KeyError`: This error occurs when a program tries to fetch a value from a dictionary using an invalid key.
- `StopIteration`: This error is thrown when the `__next__` method does not find any further items in a container.
- `TypeError`: This error occurs when a program tries to apply an operation on an object of an inappropriate type.

A complete list of errors is available in the official documentation of Python. In the following subsections, we will discuss how to handle errors, sometimes also called exceptions, using appropriate constructs in Python.

Working with exceptions in Python

When runtime errors arise, the program can terminate abruptly and can cause damage to system resources such as corrupting files and database tables. This is why error or exception handling is one of the key ingredients of writing robust programs in any language. The idea is to anticipate that runtime errors can occur and if such an error occurs, what the behavior of our program would be as a response to that particular error.

Like many other languages, Python uses the `try` and `except` keywords. The two keywords are followed by separate blocks of code to be executed. The `try` block is a regular set of statements for which we anticipate that an error may occur. The `except` block will be executed only if there is an error in a `try` block. Next is the syntax of writing Python code with `try` and `except` blocks:

```
try:
    #a series of statements
except:
    #statements to be executed if there is an error in \

     try block
```

If we anticipate a particular error type or multiple error types, we can define an `except` block with the error name and can add as many `except` blocks as we need. Such named `except` blocks are executed only if the named exception is raised in the `try` block. With the `except` block statement, we can also add an `as` statement to store the exception object as a variable that is raised during the `try` block. The `try` block in the next code example has many possible runtime errors and that is why it has multiple `except` blocks:

```
#exception1.py
try:
    print (x)
```

```
    x = 5
    y = 0
    z = x /y
    print('x'+ y)
except NameError as e:
    print(e)

except ZeroDivisionError:
    print("Division by 0 is not allowed")
except Exception as e:
    print("An error occured")
    print(e)
```

To illustrate a better use of an `except` block(s), we added multiple except blocks that are explained next:

- **The NameError block:** This block will be executed when a statement in the `try` block tries to access an undefined variable. In our code example, this block will be executed when the interpreter tries to execute the `print(x)` statement. Additionally, we named the exception object as `e` and used it with the `print` statement to get the official error detail associated with this error type.
- **The ZeroDivisionError block:** This block will be executed when we try to execute `z = x/y` and y = 0. For this block to be executed, we need to fix the `NameError` block first.
- **The default except block:** This is a catch-all `except` block, which means if no match is found with the previous two `except` blocks, this block will be executed. The last statement `print('x'+ y)` will also raise an error of type `TypeError` and will be handled by this block. Since we are not receiving any one particular type of exception in this block, we can use the `Exception` keyword to store the exception object in a variable.

Note that as soon an error occurs in any statement in the `try` block, the rest of the statements are ignored, and the control goes to one of the `except` blocks. In our code example, we need to fix the `NameError` error first to see the next level of exception and so on. We added three different types of errors in our example to demonstrate how to define multiple `except` blocks for the same `try` block. The order of the `except` blocks is important because more specific `except` blocks with error names have to be defined first and an `except` block without specifying an error name has to always be at the end.

The following figure shows all the exception handling blocks:

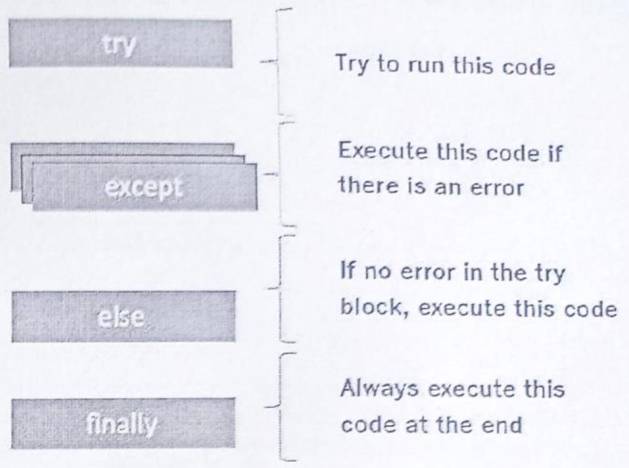
try
Try to run this code
except
Execute this code if there is an error
else
If no error in the try block, execute this code
finally
Always execute this code at the end

Figure 4.1 – Different exception handling blocks in Python

As shown in the preceding diagram, in addition to `try` and `except` blocks, Python also supports `else` and `finally` blocks to enhance the error handling functionality. The `else` block is executed if no errors were raised during the `try` block. The code in this block will be executed as normal and no exception will be thrown if any error occurs within this block. Nested `try` and `except` blocks can be added within the `else` block if needed. Note that this block is optional.

The `finally` block is executed regardless of whether there is an error in the `try` block or not. The code inside the `finally` block is executed without any exception handling mechanism. This block is mainly used to free up the resources by closing the connections or opened files. Although it is an optional block, it is highly recommended to implement this block.

Next, we will look at the use of these blocks with a code example. In this example, we will open a new file for writing in the `try` block. If an error occurs in opening the file, an exception will be thrown, and we will send the error details to the console using the `print` statement in the `except` block. If no error occurs, we will execute the code in the `else` block that is writing some text to the file. In both cases (error or no error), we will close the file in the `finally` block. The complete sample code is as follows:

```
#exception2.py
try:
```

```
    f = open("abc.txt", "w")
```

```
except Exception as e:
    print("Error:" + e)
else:
    f.write("Hello World")
    f.write("End")

finally:
    f.close()
```

We have covered extensively how to handle an exception in Python. Next, we will discuss how to raise an exception from Python code.

Raising exceptions

Exceptions or errors are raised by the Python interpreter at runtime when an error occurs. We can also raise errors or exceptions ourselves if a condition occurs that may give us bad output or crash the program if we proceed further. Raising an error or exception will provide a graceful exit of the program.

An exception (object) can be thrown to the caller by using the `raise` keyword. An exception can be of one of the following types:

- A built-in exception
- A custom exception
- A generic `Exception` object

In the next code example, we will be calling a simple function to calculate a square root and will implement it to throw an exception if the input parameter is not a valid positive number:

```
#exception3.py
import math
def sqrt(num):

    if not isinstance(num, (int, float)) :
        raise TypeError("only numbers are allowed")
    if num < 0:
        raise Exception ("Negative number not supported")

```

```
    return math.sqrt(num)
```

```

if __name__ == "__main__":
    try:
        print(sqrt(9))
        print(sqrt('a'))

        print (sqrt(-9))
    except Exception as e:
        print(e)
```

In this code example, we raised a built-in exception by creating a new instance of the `TypeError` class when the number passed to the `sqrt` function is not a number. We also raised a generic exception when the number passed is lower than `0`. In both cases, we passed our custom text to its constructor. In the next section, we will study how to define our own custom exception and then throw it to the caller.

Defining custom exceptions

In Python, we can define our own custom exceptions by creating a new class that has to be derived from the built-in `Exception` class or its subclass. To illustrate the concept, we will revise our previous example by defining two custom exception classes to replace the built-in `TypeError` and the `Exception` error types. The new custom exception classes will be derived from the `TypeError` and the `Exception` classes. Here is sample code for reference with custom exceptions:

```
#exception4.py
import math

class NumTypeError(TypeError):
    pass

class NegativeNumError(Exception):
    def __init__(self):
        super().__init__("Negative number not supported")

def sqrt(num):

    if not isinstance(num, (int, float)) :
        raise NumTypeError("only numbers are allowed")
```

```
    if num < 0:
        raise NegativeNumError

    return math.sqrt(num)

if __name__ == "__main__":

    try:
        print(sqrt(9))
        print(sqrt('a'))
        print (sqrt(-9))
    except NumTypeError as e:
        print(e)
    except NegativeNumError as e:
        print(e)
```

In this code example, the `NumTypeError` class is derived from the `TypeError` class and we have not added anything in this class. The `NegativeNumError` class is inherited from the `Exception` class and we override its constructor and add a custom message for this exception as part of the constructor. When we raise these custom exceptions in the `sqrt()` function, we do not pass any text with the `NegativeNumError` exception class. When we used the main program, we get the message with the `print (e)` statement as we have set it as part of the class definition.

In this section, we covered how to handle built-in error types using `try` and `except` blocks, how to define custom exceptions, and how to raise an exception declaratively. In the next section, we will cover logging in Python.

Using the Python logging module

Logging is a fundamental requirement for any reasonably sized application. Logging not only helps in debugging and troubleshooting but also provides insight into details of an application's internal issues. A few advantages of logging are as follows: