CSD 2101

Python Programming

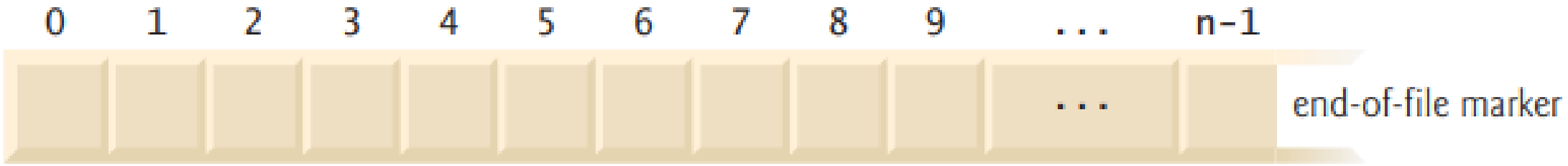
MODULE V FILE HANDLING AND EXCEPTION HANDLING 9

Introduction to Files – Opening and Closing Files – Reading and Writing Files – File Position – Exception: Errors and Exceptions, Exception Handling, Multiple Exceptions.

- Variables, lists, tuples, dictionaries, sets, arrays, pandas Series and pandas Data Frames offer only temporary data storage.
- Files are named locations on disk to store related information. They are used to permanently store data in non-volatile memory (e.g. hard disk).
- Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.
- When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

- 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.

The diagram below shows a conceptual view of a file:



We consider text files in several popular formats - plain text, JSON (JavaScript Object Notation) and CSV (comma-separated values).

Python, a file operation takes place in the following order:

1.Open a file

2.Read or write

3.Close the file

Opening Files in Python

- Python has a built-in open() function to open a file.
- This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

r - Opens a file for reading.

w - Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists

x - Opens a file for exclusive creation. If the file already exists, the operation fails.

a - Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.

t - Opens in text mode

b - Opens in binary mode.

+ - Opens a file for updating (reading and writing)

x - Create - will create a file, returns an error if the file exist

```
f=open("hai.txt","x")
```

r - Opens a file for reading

```
f=open("sampl.txt","r")
print(f.read())
```

Open a file on a different location:

```
f=open("G:\\hello.txt","r")
print(f.read())
hello welcome
```

read the whole file, line by line:

```
f=open("sampl.txt","r")
for x in f:
    print(x)
```

```
hello welcome

hello welcome
```

Close Files

```
f=open("sampl.txt","r")
print(f.readline())
f.close()
```

a - Opens a file for appending at the end of the file without truncating it.

```
f=open("sampl.txt",“a")
    f.write("hai")
    f=open("sampl.txt","r")
    print(f.read())
    hello welcomehello welcomehai
```

w - Opens a file for writing

```
    f=open("sampl.txt",“w")
    f.write("hi")
    f=open("sampl.txt","r")
    print(f.read())
```

hi

Delete a File:

- To delete a file, you must import the OS module, and run its os.remove() function:

```
import os
os.remove("hai.txt")
```

Delete Folder

- To delete an entire folder, use the os.rmdir() method:

```
import os
os.rmdir("new")
```

Exception :

An exception is **an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions**.

Sr.No.	Exception Name & Description
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.

6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.

11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** Base class for all lookup errors.
15	**IndexError** Raised when an index is not found in a sequence.

16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**IOError** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.

21	**IOError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.

26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Errors:

Errors or mistakes in a program are often referred to as bugs. The process of finding and eliminating errors is called debugging. Errors can be classified into three major groups:

- Syntax errors
- Runtime errors
- Logical errors

Syntax Errors

Syntax errors, also known as parsing errors

```
if(a>b)
SyntaxError: expected ':'
```

Common Python syntax errors include:

- leaving out a keyword
- putting a keyword in the wrong place
- leaving out a symbol, such as a colon, comma or brackets
- misspelling a keyword
- incorrect indentation
- empty block

Runtime errors

If a program is syntactically correct – that is, free of syntax errors – it will be run by the Python interpreter.

However, the program may exit unexpectedly during execution if it encounters a *runtime error* – a problem which was not detected when the program was parsed, but is only revealed when a particular line is executed. When a program comes to a halt because of a runtime error, we say that it has crashed.

Some examples of Python runtime errors:

- division by zero
- performing an operation on incompatible types
- using an identifier which has not been defined
- accessing a list element, dictionary value or object attribute which doesn’t exist
- trying to access a file which doesn’t exist

Logical errors:

Logical errors are the most difficult to fix. They occur when the program runs without crashing, but produces an incorrect result. The error is caused by a mistake in the program's logic.

- using the wrong variable name
- indenting a block to the wrong level
- using integer division instead of floating point division
- getting operator precedence wrong
- making a mistake in a boolean expression
- off-by-one, and other numerical errors

Exceptions:

Exceptions are raised when the program is syntactically correct, but the code resulted in an error.

```
a=5
b=a/0
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    b=a/0
ZeroDivisionError: division by zero
```

```
2+b*4
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    2+b*4
NameError: name 'b' is not defined

5+'5'
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    5+'5'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- The try block - test a block of code for errors.
- The except block - handle the error.
- The else block - execute code when there is no error.
- The finally block - execute code, regardless of the result of the try- and except blocks.

The try block will generate an exception, because x is not defined

```
>>>try:
        print(z)
>>>except:
        print("an exception error")
…
>>>an exception error
```

Many Exceptions

Print one message if the try block raises a NameError and another for other errors:

```
>>>try:
        print(z)
>>>except NameError:
        print("Variable z is not defined")
>>>except:
        print("something went to wrong")


>>>Variable z is not defined
```

Else

The else keyword to define a block of code to be executed if no errors were raised:

```
try:
    print("Hello Welcome")
except:
    print("something went to wrong")
else:
    print("nothing went to wrong")
```

```
Hello Welcome
nothing went to wrong
```

```
x=10
try:
    print(y)
except:
    print("something went to wrong")
else:
    print("Nothing went to wrong")
```

```
something went to wrong
```

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
    print(y)
except:
    print("something went to wrong")
finally:
    print("The try except is not finished")
```

```
something went to wrong
The try except is not finished
```

```
y=10
try:
    print(y)
except:
    print("something went to wrong")
finally:
    print("The try except is not finished")
```

```
10
The try except is not finished
```

```
try:
    f=open("ha.txt")
    try:
        f.write("Hello Welcome")
    except:
        print("Something went to wrong write a file")
    finally:
        f.close()
except:
    print("Something went to wrong to open a file")
```

Something went to wrong to open a file

```
try:
    f=open("hai.txt")
    try:
        f.write("Hello Welcome")
    except:
        print("Something went to wrong to write a file")
    finally:
        f.close()
except:
    print("Something went to wrong to open a file")
```

Something went to wrong to write a file

Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception.

```
def fun(var):
    try:
        return int(var)
    except (ValueError, Argument):
        print("The argument doesn't contains number", Argument)


fun(10)
```

10

```
def fun(var):
    try:
        return int(var)
    except (ValueError, Argument):
        print("The argument doesn't contains number", Argument)


fun("x")
```

The argument does not contain numbers invalid literal for int() with base 10: 'x'

Raise an exception

Raise an error and stop the program if x is lower than 0:

```
x=-1
if x<0:
    raise Exception("No number below zero")
```

```
Traceback (most recent call last):
  File "<pyshell#93>", line 2, in <module>
    raise Exception("No number below zero")
Exception: No number below zero
```

Raise a TypeError if x is not an integer:

```
msg="hello welcome"
if not type(msg) is int:
    raise TypeError("only integers are allowed")
```

```
Traceback (most recent call last):
  File "<pyshell#97>", line 2, in <module>
    raise TypeError("only integers are allowed")
TypeError: only integers are allowed
```

User-defined Exceptions

- Programs may name their own exceptions by creating a new exception class.
- Exceptions should typically be derived from the Exception class, either directly or indirectly.

```
class Usererror(RuntimeError):
    def __init__(self,arg):
        self.args=arg

try:
    raise Usererror("usererror")
except Usererror as e:
    print(e.args)
```

```
('u', 's', 'e', 'r', 'e', 'r', 'r', 'o', 'r')
```

help(Exception)

```
Help on class Exception in module builtins:

class Exception(BaseException)
 |  Common base class for all non-exit exceptions.
 |
 |  Method resolution order:
 |      Exception
 |      BaseException
 |      object
 |
 |  Built-in subclasses:
 |      ArithmeticError
 |      AssertionError
 |      AttributeError
 |      BufferError
 |      ... and 15 other subclasses
 |
```

```
 |  Methods defined here:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for
accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from BaseException:
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __reduce__(...)
 |      Helper for pickle.
```

```
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __setattr__(self, name, value, /)
|      Implement setattr(self, name, value).
|
|  __setstate__(...)
|
|  __str__(self, /)
|      Return str(self).
|
|  with_traceback(...)
|      Exception.with_traceback(tb) --
|      set self.__traceback__ to tb and return self.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from BaseException:
|
|  __cause__
|      exception cause
|
|  __context__
|      exception context
|
|  __dict__
|
|  __suppress_context__
|
|  __traceback__
|
|  args
```

```
def divided(a,b):
    try:
        result=a/b
    except ZeroDivisionError:
        print("division by Zero")
    else:
        print("result is",result)
    finally:
        print("executing finally clause")
```

```
divided(2,1)
result is 2.0
executing finally clause
divided(2,0)
division by Zero
executing finally clause
divided(0,2)
result is 0.0
executing finally clause
```