10 File Handling

File handling in Python involves interacting with external files, allowing the reading and writing of data. This is crucial for tasks such as data storage, retrieval, and manipulation.

10.1 Reading and Writing Text Files:

- **Opening a File:**
    - Use the `open()` function to open a file, specifying the file path and mode (e.g., 'r' for reading, 'w' for writing).

```
file_path = "example.txt"
# Reading a file
with open(file_path, 'r') as file:
   content = file.read()
# Writing to a file
with open(file_path, 'w') as file:
   file.write("Hello, World!")
```

Note: Please make sure that a text file with the given name exist on your computer in same folder as your python script, or in the path folder you mention.

- Parameters:
  - `filename`: Name of the file you want to open.
  - `mode`: Describes the purpose of opening the file. Common modes include:

The open() function, the first argument, is the file details. If the file path is different, you need to mention the actual path. For example, write 'C:\\PythonFiles\\myfile.txt' (with double-backslash). The second argument is the mode, which specifies how the file will be used.

- `'r'`: Read (means reading, default mode, raises an error if the file doesn't exist).
- `'w'`: Write (means writing, creates a new file. If the file name doesn't exist, it will be created. If it exists, any existing data will be erased).
- `'a'`: Append (means append, creates a new file if it doesn't exist, otherwise appends to the file).

- **Reading Files:** provides several methods to read from files.

  - read(): Reads the entire file as a string.
  `content = file.read()`
  - readline(): Reads a single line from the file.
  `line = file.readline()`
  - readlines(): Reads all lines in a file and returns them as a list of strings.
  `lines = file.readlines()`

Example:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

- **Writing to Files:** allows writing to files using various methods.

  - write(): Writes a string to a file.
  `file.write("Hello, World!")`

- writelines(): Writes a list of strings to a file.
```
file.writelines(["Hello, ", "World!"])
```

Example:

```
with open("output.txt", "w") as file:
    file.write("This is a sample text.")
```

- **Appending to Files:** When you want to add content to an existing file without deleting its previous content, you use the append mode (`'a'`).

Example:

```
with open("output.txt", "a") as file:
    file.write("\nThis is an appended line.")
```

- **Closing Files:** After performing file operations, it's important to close the file to free up system resources. This can be done using the `close()` method.

Example:

```
file = open("example.txt", "r")
 Perform file operations
file.close()
```

Using a for loop to read text files

```
# Example 1
f=open('myfile.txt','r')
for line in f:
    print(line,end='') #remember this two single quotes, not a single
double quote
f.close()
```

Following is an example of how to read a file, access lines, save it, split, strip, indexing access, do some operation on the float numbers. Here, I am using a text file where some ids are written and some values are written with tab separation.

```
# Example 2
import io
fhandle = io.open("ids2.txt", "r")
counter = 0
eval_sum = 0.0
for line in fhandle:
    line_str = line.strip()
    print(line_str)
    line_list = line_str.split("\t")
    print(line_list)
    val= line_list[10]
    print(val)
    eval_sum = eval_sum + float(val)
    counter = counter + 1
fhandle.close()
mean = eval_sum/counter
print("Mean e-value is: ", str(mean))
```

# text.txt contains AGCTGTCGTGTAGCTGTAGTCATG

```
# Example 3
import io
fhandle = io.open("text.txt", "r")
f = fhandle.readline()
fhandle.close()
print(f)
for line in fhandle:
    line_str = line.strip()
    line_list = line_str.split("\n")
    print(line_list)
    val= line_list[10]
    print(val)
    eval_sum = eval_sum + float(val)
    counter = counter + 1
fhandle.close()
mean = eval_sum/counter
print("Mean e-value is: ", str(mean))
```

```
# Example 4
a = 1
b = 2
import io
fhandle = io.open("file1.txt", "w")
fhandle.write(str(a) + "\n")
fhandle.write(str(b) + "\n")
fhandle.close()
```

10.2 Working with CSV files:

When dealing with Excel and CSV files in Python, different libraries and approaches are used compared to plain text or binary files. Python provides powerful libraries like `pandas`, `csv`, and `openpyxl` to handle these types of files efficiently.

Reading CSV (Comma-Separated Values) Files:

CSV (Comma-Separated Values) files are plain text files that store tabular data, with each line representing a row and columns separated by commas. The `csv` module provides functionality for reading from and writing to CSV files.

Example:

```
import csv
csv_file_path = "data.csv"
# Reading from a CSV file
with open(csv_file_path, 'r') as csv_file:
    reader = csv.reader(csv_file)
    for row in reader:
        print(row)
```

- Using the `pandas` library:
 `pandas` provides a more convenient way to handle CSV files, especially for large datasets. It reads the file into a `DataFrame`, which is a powerful data structure for data manipulation.

Example:

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.head())
```

Writing to CSV Files

- Using the `csv` module:

Example:

```
import csv
with open('output.csv', mode='w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['Name', 'Age', 'City'])
    csv_writer.writerow(['Alice', '23', 'New York'])
```

- Using the `pandas` library:

```
import pandas as pd
data = {'Name': ['Alice', 'Bob'], 'Age': [23, 25], 'City': ['New York',
'Los Angeles']}
df = pd.DataFrame(data)
df.to_csv('output.csv', index=False)
```

10.3 Working with Excel Files:

Excel files (typically `.xls` or `.xlsx`) are more complex than CSV files, as they support multiple sheets, formulas, cell formatting, and more. Python offers several libraries to work with Excel files.

Reading Excel Files

- Using `pandas`: `pandas` can read Excel files into a `DataFrame`, similar to how it reads CSV files.

Example:

```
import pandas as pd
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print(df.head())
```

- Using `openpyxl`: `openpyxl` is a library specifically designed for reading and writing Excel 2010 xlsx/xlsm/xltx/xltm files.

Example:

```python
from openpyxl import load_workbook
workbook = load_workbook(filename='data.xlsx')
sheet = workbook.active
for row in sheet.iter_rows(values_only=True):
    print(row)
```

Writing to Excel Files

- Using `pandas`: `pandas` can also write `DataFrame` objects to Excel files.

Example1:

```python
import pandas as pd
data = {'Name': ['Alice', 'Bob'], 'Age': [23, 25], 'City': ['New York',
'Los Angeles']}
df = pd.DataFrame(data)
df.to_excel('output.xlsx', index=False)
```

Example2:

```python
import pandas as pd
my_dict = {
  'NAME':['Alex','Ron','ravi'],
  'ID':[1,2,3],
  'MATH':[45,34,50]
}
df = pd.DataFrame(data=my_dict)
df2=df.copy()
#print(df)
#df.to_excel('df_file.xlsx',header=False)
#df.to_excel('df_file.xlsx',index = False)
#df.to_excel('df_file.xlsx',index = False,columns=['NAME','ID'])
#df.to_excel('df_file.xlsx',index = False,columns=['NAME','ID','MATH'])
#df.to_excel('df_file.xlsx',index = False,sheet_name='mySheet1')
with pd.ExcelWriter('df_file.xlsx',engine='openpyxl') as my_obj:
   df.to_excel(my_obj,sheet_name='mySheet1')
   df2.to_excel(my_obj,sheet_name='mySheet2')

df1 = pd.read_excel (r'in_file.xlsx', sheet_name='mySheet1')
print (df1)
df2 = pd.read_excel (r'in_file.xlsx', sheet_name='mySheet2')
print(df2)
with pd.ExcelWriter('out_file.xlsx',engine='openpyxl') as my_obj:
   df1.to_excel(my_obj,sheet_name='mySheet1')
   df2.to_excel(my_obj,sheet_name='mySheet2')
```

- Using `openpyxl`: `openpyxl` allows writing data to Excel files, including more complex tasks like creating formulas or applying styles.

Example:

```
from openpyxl import Workbook
workbook = Workbook()
sheet = workbook.active
sheet['A1'] = 'Name'
sheet['B1'] = 'Age'
sheet['C1'] = 'City'
sheet.append(['Alice', 23, 'New York'])
workbook.save('output.xlsx')
```

-Using ‘xlwt’

Example:

```
a = "ACDEFGHIKLMNPQRSTVWY"
# Writing to an excelsheet
import xlwt
from xlwt import Workbook
# Workbook is created
wb = xlwt.Workbook()
# add_sheet is used to create sheet.
sheet1 = wb.add_sheet('Sheet 1')
for i in range(len(a)):
    sheet1.write(0,i,a[i])
wb.save('xlwt.xls')
```

-Using xlswriter

Example:

```
import xlsxwriter
#create file (workbook) and add worksheet
outWorkbook = xlsxwriter.Workbook("out.xlsx")
outSheet = outWorkbook.add_worksheet()
#declare data
names = ["Kyle", "Bob", "Mary"]
values = [70, 82, 71]
#write headers
outSheet.write("A1", "Names")
outSheet.write("B1", "Scores")
#write data to file
for item in range(len(names)):
    outSheet.write(item+1, 0, names[item])
    outSheet.write(item+1, 1, values[item])
outSheet.write("D1", "Total")
outSheet.write_formula("D2", "=SUM(B2:B4)")
outSheet.write(1,0, names[0])
outSheet.write(2,0, names[1])
outSheet.write(3,0, names[2])
outSheet.write(1,1, values[0])
outSheet.write(2,1, values[1])
outSheet.write(3,1, values[2])
outWorkbook.close()
```

10.4 Deleting and renaming files

```
import os
# os.remove('myfile.txt')
# os.rename('myfile.txt','newfile.txt')
```

11 Functions

Functions in Python are blocks of organized, reusable code designed to perform a specific task. They enhance code modularity, readability, and reusability.

11.1 Defining and Calling Functions:

Defining a Function: Functions are defined using the def keyword, followed by the function name and parameters.

Here are simple rules to define a function in Python.

- Function blocks begin with the keyword *def* followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

```
# Example1
def hello(name):
  "Print a Hello with the name provided by the user."
  print("Hello, " + name + "!")
# Example2
def checkIfPrime(x):
  "check if a number is prime."
  for y in range(2, x):
    if (x%y == 0):
      return False
    return True
# Example3
def pow(x, y):
  "print x raised to the power y"
  result = x**y
  print(x,"raised to the power",y,"is",result)
```

Calling a Function: Functions are called by using the function name followed by parentheses and providing any required arguments.

```
hello("Sam")
print(checkIfPrime(13))
pow(8,2)
```

```
Output:
Hello, Sam!
True
8 raised to the power 2 is 64
```

11.2 Parameters and Return Values:

Parameters: Functions can accept parameters (input values) to make them more flexible and versatile.

```
def add(x, y):
   return x + y
```

• **Return Values:** The `return` statement is used to send a value back to the caller.

```
result = add(3, 5)
def check1(x):
    if x>0:
        result="Positive"
    elif x<0:
        result="Negative"
    else:
        result="Zero"
    return(result)
print(check1(1))
print(check1(-10))
print(check1(0))
```

```
Output:
Positive
Negative
Zero
```

Functions without a `return` statement implicitly eturn `None`.

```
def check2(x):
  if x>0:
    result="Positive"
  elif x<0:
    result="Negative"
  else:
    result="Zero"
  result
print(check2(1))
print(check2(-10))
print(check2(0))
```

```
Output:
None
None
None
```

Using return in more efficient way:

```
def check3(x):
  if x>0:
    return("Positive")
  elif x<0:
    return("Negative")
  else:
    return("Zero")
print(check3(1))
print(check3(-10))
print(check3(0))
```

```
Output:
Positive
Negative
Zero
```

An example with multiple returns

```
def multi_return():
  my_list = ["red", 20, "round"]
  return my_list
print(multi_return())
```

```
Output:
```

```
['red', 20, 'round']
```

11.3 Function Arguments

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

11.3.1 Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that is passed into a function.

From a function's perspective:

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.
- You can call a function by using the following types of formal arguments −Required arguments, Keyword arguments, Default arguments, Variable-length arguments

11.3.2 Required arguments

Required arguments are the arguments passed to a function in the correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. Let me clarify the difference further with a more detailed explanation.

Named Arguments:

Named arguments are parameters that you define in a function signature. These can either be required or have default values.

Example of Named Arguments:

```
def greet(name, age=30):  # `name` and `age` are named arguments
    print(f"Hello, {name}. You are {age} years old.")
```

In the above function:
- `name` is a named argument that is required because no default value is provided.
- `age` is a named argument that has a default value of `30`. If you don't pass a value for `age`, it will default to 30.

Keyword Arguments:

Keyword arguments refer to the way you pass values to the named arguments during the function call by explicitly mentioning the parameter name.

Example of Keyword Arguments:

```
greet(name="Alice", age=25)  # Passing arguments by name (keyword
arguments)
```

Here, during the function call:
- `name="Alice"` is a keyword argument because you are passing a value by naming the parameter.
- `age=25` is also a keyword argument.

Example Combining Positional and Keyword Arguments:

```
# Function definition with named arguments
def greet(name, age=30):
    print(f"Hello, {name}. You are {age} years old.")

# Calling using positional and keyword arguments
greet("Alice")  # Positional argument for `name`, uses default `age=30`
greet("Bob", age=40)  # Positional for `name`, keyword argument for
`age`
```

```
Output:
Hello, Alice. You are 30 years old.
Hello, Bob. You are 40 years old.
```

- In `greet("Alice")`, "Alice" is passed as a positional argument for `name`, and `age` defaults to 30.
- In `greet("Bob", age=40)`, "Bob" is passed as a positional argument, and `age=40` is passed as a keyword argument.

11.3.3 Default arguments OR Default Parameter Value

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed.

Default Arguments: Parameters can have default values, allowing the function to be called without providing those values.

```
def hello(name, greeting="Hello"):
  print(greeting + ", " + name + "!")
def info1( name, age = 35 ):
   print("Name: ", name)
   print("Age ", age)
   return;
```

Now lets call the functions

```
hello("Sam")
```

```
Output:
Hello, Sam!
```

```
info1( age=50, name="miki" )
```

```
Output:
Name: miki
Age 50
```

```
info1(name="miki" )
```

```
Output:
Name: miki
Age 35
```

Keyword Arguments: Arguments can be passed by explicitly mentioning the parameter name. Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

```
hello(name="Alice", greeting="Hi")
def show( str ):
   print(str)
   return;
show( str = "My string")
```

```
Output:
Hi, Alice!
My string
```

The following example gives clearer picture. Note that the order of parameters does not matter.

```
def info2( name, age ):
   print("Name: ", name)
   print("Age ", age)
   return;
info2(age=50, name="miki")
```

```
Output:
```

```
Name: miki
Age 50
```

11.3.4 Variable-Length Argument Lists/Arbitrary Arguments, *args/Arbitrary Keyword Arguments, **kwargs

Python allows you to have functions with varying numbers of arguments. Functions can accept a variable number of arguments using `*args` and `**kwargs`. You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Using *args as a function parameter enables you to pass an arbitrary number of arguments to that function. The arguments are then accessible as the tuple args in the body of the function.

Syntax for a function with non-keyword variable arguments is this

```
def functionname([formal_args,] *var_args_tuple ):
      function_code
      return(expression)
```

An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Example

```
def var_arg(named_arg, *args):
  print(named_arg)
  print(args)
var_arg(1,2,3,4,5)
```

```
Output:
1
(2, 3, 4, 5)
```

*Note: The parameter *args must come after the named parameters to a function. The name args is just a convention; you can choose to use another.*

Example

```
def addNum(*num):
  sum=0
  for i in num:
    sum = sum+i
  print(sum)

addNum(1,2,3,4,5)

def UserAge(**age):
  for i,j in age.items():
    print("Name=%s, Age=%s"%(i,j))
UserAge(Peter=5,John=7,Sam=10)
```

```
Output:
```

```
15
Name=Peter, Age=5
Name=John, Age=7
Name=Sam, Age=10
```

**kwargs (standing for keyword arguments) allows you to handle named arguments that you have not defined in advance. The keyword arguments return a dictionary in which the keys are the argument names, and the values are the argument values.

```
def my_func(x, y=7, *args, **kwargs):
  print(kwargs)
my_func(2,3,4,5,6, a=7, b=8)
```

```
Output:
{'a': 7, 'b': 8}
```

a and b are the names of the arguments that we passed to the function call. The arguments returned by **kwargs are not included in *args. This is because Python has a special function called *args which processes multiple arguments in a function.

Example

```
def show_users(a, b, *args):
    print(a, b, *args) #We can pass in as many arguments to our
function
show_users("Alex", "Peter", "Violet", "Julie")
```

```
Output:
Alex Peter Violet Julie
```

One can assign default value also

Example

```
def pow2(x, y=2):
    result = x**y
    print(x,"raised to the power",y,"is",result)
pow2(3) #here y is optional and will take the value 2 when not provided
```

```
Output:
3 raised to the power 2 is 9
```

```
pow2(3,4) #so this will also work
```

```
Output:
3 raised to the power 4 is 81
```

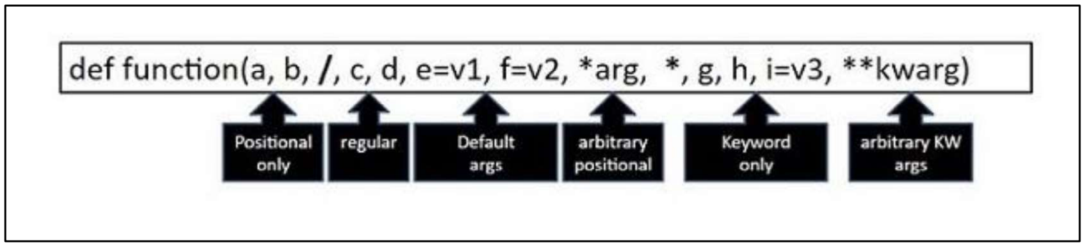
11.3.5 Order of Arguments

A function can have arguments of any of the types defined above. However, the arguments must be declared in the following order −

- The argument list begins with the positional-only args, followed by the slash (/) symbol.
- It is followed by regular positional args that may or may not be called as keyword arguments.

- Then there may be one or more args with default values.
- Next, arbitrary positional arguments represented by a variable prefixed with single asterisk that is treated as tuple. It is the next.
- If the function has any keyword-only arguments, put an asterisk before their names start. Some of the keyword-only arguments may have a default value.
- Last in the bracket is argument with two asterisks ** to accept arbitrary number of keyword arguments.

The following diagram shows the order of formal arguments −



(Image source: https://www.tutorialspoint.com/python/python_functions.htm; Reference 3)

Examples followed by their output

```
def my_func(x, y=7, *args, **kwargs):
  print(kwargs)
my_func(2,3,4,5,6, a=7, b=8)
```

```
Output:
{'a': 7, 'b': 8}
```

```
def my_func(x, y=7, *args, v=8,**kwargs):
    print(x,y,v)
    print(args)
    print(kwargs)
my_func(1,2,3,4,5,v=6,a=5,b=6)
```

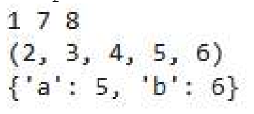
```
Output:
1 2 6
(3, 4, 5)
{'a': 5, 'b': 6}
```

```
def my_func2(x, y=7, *args, v=8,**kwargs):
    print(x,y,v)
    print(args)
    print(kwargs)
my_func2(1,2,3,4,5,6,a=5,b=6)
```

```
Output:
 1 2 8
 (3, 4, 5, 6)
 {'a': 5, 'b': 6}
```

```
def my_func3(x, *args, y=7, v=8,**kwargs):
    print(x,y,v)
    print(args)
    print(kwargs)
my_func3(1,2,3,4,5,6,a=5,b=6)
```

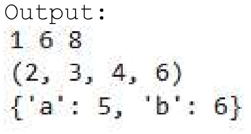
Output:



```
1 7 8
(2, 3, 4, 5, 6)
{'a': 5, 'b': 6}
```

```
def my_func4(x, *args, y, v=8,**kwargs):
    print(x,y,v)
    print(args)
    print(kwargs)
my_func4(1,2,3,4,6,a=5,b=6, y=6)
```

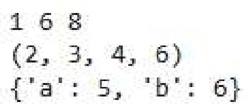
Output:



```
1 6 8
(2, 3, 4, 6)
{'a': 5, 'b': 6}
```

```
my_func4(1,2,3,4,6,y=6,a=5,b=6)
```

Output:



```
1 6 8
(2, 3, 4, 6)
{'a': 5, 'b': 6}
```

Named arguments: In the Case of named arguments, the order of actual arguments doesn't matter.

Example

```
pow(x=8,y=2)
pow(y=2,x=8)
```

One can also use named and unnamed arguments in a single call

```
pow(8, y=2)
```

However, the following will not work

```
pow(x=8, 2)
```

Because, positional arguments must appear before a keyword argument in Python. This is because Python interprets positional arguments in the order in which they appear. Then, it interprets the keyword arguments that have been specified.

Example

```
def add_numbers(a, b):
    return a + b
print(add_numbers(2, 3)) #we have two positional arguments
print(add_numbers(a=2, b=2)) #We can also specify these arguments as
keyword arguments
print(add_numbers(2, b=2))
print(add_numbers(a=3, 2))
```

However, we cannot specify a keyword argument first and then switch to the positional one.

11.4 Lambdas OR The Anonymous Functions

Creating a function normally (using def) assigns it to a variable with its name automatically. Python allows us to create functions on-the-fly, provided that they are created using lambda syntax. These functions are called anonymous, which means that the function is without a name because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions. Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions. An anonymous function cannot be a direct call to print because lambda requires an expression. Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

The syntax of lambda functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

This approach is most commonly used when passing a simple function as an argument to another function. The syntax is shown in the next example and consists of the lambda keyword followed by a list of arguments, a colon, and the expression to evaluate and return.

```
str1 = 'Python lessons'
upper = lambda string: string.upper()
print(upper(str1))
```

Let's see the following standard function generation:

```
def polynomial(x):
  return x**2 + 5*x +4
print(polynomial(-4))
```

Now the same function can be given as lambda function as follows:

```
print((lambda x: x**2 + 5*x + 4)(-4))
```

In the code above, we created an anonymous function on the fly and called it with an argument.