6. Python File Operation

6.1. Reading config files in python

6.2. Writing log files in python

6.3. Understanding read functions, read(), readline() and readlines()

6.4. Understanding write

6.5. functions write() and writelines()

6.6. Manipulating file pointer using seek

6.7. Programming using file operations

6.1. Reading config files in python

Python Reading .ini Configuration Files

This article aims to read configuration files written in the common **.ini** configuration file format.

The **configparser** module can be used to read configuration files.

Code #1: Configuration File

abc.ini

```
; Sample configuration file
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local
# Setting related to debug configuration
[debug]
pid-file = /tmp/spam.pid
show_warnings = False
log_errors = true
[server]
nworkers: 32
port: 8080
root = /www/root
signature:
```

Code #2: Reading the file and extracting values. (config_read.py)

```
from configparser import ConfigParser

configur = ConfigParser()
print (configur.read('abc.ini'))

print ("Sections : ", configur.sections())
```

```
print ("Installation Library : ", configur.get('installation','library'))
print ("Log Errors debugged ? : ", configur.getboolean('debug','log_errors'))
print ("Port Server : ", configur.getint('server','port'))
print ("Worker Server : ", configur.getint('server','nworkers'))
```

Output:

```
['abc.ini']
Sections:  ['installation', 'debug', 'server']
Installation Library:  /usr/local/lib
Log Errors debugged?:  True
Port Server:  8080
Worker Server:  32
```

Code #3: Modifying (writing) the config file and reading it. (config_write.py)

```
from configparser import ConfigParser

configur = ConfigParser()
print (configur.read('abc.ini'))

print ("Sections : ", configur.sections())
print ("Installation Library : ", configur.get('installation','library'))
print ("Log Errors debugged ? : ", configur.getboolean('debug','log_errors'))
print ("Port Server : ", configur.getint('server','port'))
print ("Worker Server : ", configur.getint('server','nworkers'))
configur.set('server','port','9000')
configur.set('debug','log_errors','False')

import sys
configur.write(sys.stdout)
```

Output:

['abc.ini']

Sections : ['installation', 'debug', 'server']

Installation Library : /usr/local/lib

Log Errors debugged ? : True

Port Server : 8080

Worker Server : 32

[installation]

library = %(prefix)s/lib

include = %(prefix)s/include

bin = %(prefix)s/bin

prefix = /usr/local

[debug]

pid-file = /tmp/spam.pid

show_warnings = False

log_errors = False

[server]

nworkers = 32

port = 9000

root = /www/root

signature =

6.2. Writing log files in python

Logging in Python

Logging is a means of tracking events that happen when some software runs. Logging is important for software developing, debugging and running. If you don't have any logging record and your program crashes, there are very little chances that you detect the cause of the problem. And if you detect the cause, it will consume a lot of time. With logging, you can

leave a trail of breadcrumbs so that if something goes wrong, we can determine the cause of the problem.

There are a number of situations like if you are expecting an integer, you have been given a float and you can a cloud API, the service is down for maintenance and much more. Such problems are out of control and are hard to determine.

Why Printing is not a good option?

Some developers use the concept of printing the statements to validate if the statements are executed correctly or some error has occurred. But printing is not a good idea. It may solve your issues for simple scripts but for complex scripts, printing approach will fail.

Python has a built-in module **logging** which allows writing status messages to a file or any other output streams. The file can contain the information on which part of the code is executed and what problems have been arisen.

Levels of Log Message

There are five built-in levels of the log message.

- **Debug:** These are used to give detailed information, typically of interest only when diagnosing problems.
- **Info:** These are used to Confirm that things are working as expected
- **Warning:** These are used an indication that something unexpected happened, or indicative of some problem in the near future
- **Error:** This tells that due to a more serious problem, the software has not been able to perform some function
- **Critical:** This tells serious error, indicating that the program itself may be unable to continue running

If required, developers have the option to create more levels but these are sufficient enough to handle every possible situation. Each built-in level has been assigned its numeric value.

Level	Numeric Value
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Logging module is packed with several features. It has several constants, classes, and methods. The items with all **caps are constant**, the **capitalize items are classes** and the items which start with **lowercase letters** are **methods.**

There are several logger objects offered by the module itself.

- **Logger.info(msg) :** This will log a message with level INFO on this logger.
- **Logger.warning(msg) :** This will log a message with level WARNING on this logger.
- **Logger.error(msg) :** This will log a message with level ERROR on this logger.
- **Logger.critical(msg) :** This will log a message with level CRITICAL on this logger.
- **Logger.log(lvl,msg) :** This will Logs a message with integer level lvl on this logger.
- **Logger.exception(msg) :** This will log a message with level ERROR on this logger.
- **Logger.setLevel(lvl) :** This function sets the threshold of this logger to lvl. This means that all the messages below this level will be ignored.
- **Logger.addFilter(filt) :** This adds a specific filter filt to the to this logger.
- **Logger.removeFilter(filt) :** This removes a specific filter filt to the to this logger.
- **Logger.filter(record) :** This method applies the logger's filter to the record provided and returns True if record is to be processed. Else, it will return False.
- **Logger.addHandler(hdlr) :** This adds a specific handler hdlr to the to this logger.
- **Logger.removeHandler(hdlr) :** This removes a specific handler hdlr to the to this logger.
- **Logger.hasHandlers() :** This checks if the logger has any handler configured or not.

The Basics

Basics of using the logging module to record the events in a file are very simple.

For that, simply import the module from the library.

1. Create and configure the logger. It can have several parameters. But importantly, pass the **name of the file** in which you want to **record the events.**
2. Here the format of the logger can also be set. By default, the file works in **append** mode but we can change that to **write mode** if required.
3. Also, the level of the logger can be set which acts as the threshold for tracking based on the numeric values assigned to each level.

   There are several attributes which can be passed as parameters.

4. The list of all those parameters is given in Python Library. The user can choose the required attribute according to the requirement.
   After that, create an object and use the various methods as shown in the example.

Example:

```
#importing module
import logging

#Create and configure logger
logging.basicConfig(filename="newfile.log",
                    format='%(asctime)s %(message)s',
                    filemode='w')

#Creating an object
logger=logging.getLogger()
#Setting the threshold of logger to DEBUG
logger.setLevel(logging.DEBUG)
#Test messages
logger.debug("Harmless debug Message")
logger.info("Just an information")
logger.warning("Its a Warning")
logger.error("Did you try to divide by zero")
logger.critical("Internet is down")
```

Output: Newfile.log content

```
2021-08-03 20:53:22,387 Harmless debug Message
2021-08-03 20:53:22,387 Just an information
2021-08-03 20:53:22,387 Its a Warning
2021-08-03 20:53:22,387 Did you try to divide by zero
2021-08-03 20:53:22,387 Internet is down
```

Example:

Let's take a simple example and see how you can plug in the logging debug function in place of the print function.

```
import logging
import os

logging.basicConfig(filename="newfile1.log", format='%(asctime)s :: %(levelname)s ::
%(funcName)s :: %(lineno)d \
:: %(message)s', level = logging.INFO)

def addition(x, y):
    add = x + y
    return add

def subtract(x, y):
    sub = x - y
    return sub

def multiply(x, y):
    mul = x * y
    return mul

def divide(x, y):
    div = x / y
    return div

def exponent(x, y):
    exp = x ** y
    return exp
```

```
num1 = 20
num2 = 2

def main():
    add_result = addition(num1, num2)
    logging.info('Add: {} + {} = {}'.format(num1, num2, add_result))
    sub_result = subtract(num1, num2)
    logging.info('Sub: {} - {} = {}'.format(num1, num2, sub_result))
    mul_result = multiply(num1, num2)
    logging.info('Mul: {} * {} = {}'.format(num1, num2, mul_result))
    div_result = divide(num1, num2)
    logging.info('Div: {} / {} = {}'.format(num1, num2, div_result))
    exp_result = exponent(num1, num2)
    logging.info('Exp: {} ** {} = {}'.format(num1, num2, exp_result))

main()
os.startfile("newfile1.log")
```

Output: Newfile1.log content

```
2021-08-08 16:26:34,785 :: INFO :: main :: 36 :: Add: 20 + 2 = 22
2021-08-08 16:26:34,785 :: INFO :: main :: 39 :: Sub: 20 - 2 = 18
2021-08-08 16:26:34,785 :: INFO :: main :: 42 :: Mul: 20 * 2 = 40
2021-08-08 16:26:34,785 :: INFO :: main :: 45 :: Div: 20 / 2 = 10.0
2021-08-08 16:26:34,785 :: INFO :: main :: 48 :: Exp: 20 ** 2 = 400
```

Python File I/O - Read and Write Files

In Python, the IO module provides methods of three types of IO operations; raw binary files, buffered binary files, and text files. The canonical way to create a file object is by using the open() function.

Any file operations can be performed in the following three steps:

1. Open the file to get the file object using the built-in open() function. There are different access modes, which you can specify while opening a file using the open() function.
2. Perform read, write, append operations using the file object retrieved from the open() function.
3. Close and dispose the file object.

Here is a list of the different access modes of opening a file −

Sr. No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file

	does not exist, creates a new file for writing.
6	**wb** Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	**w+** Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	**wb+** Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a** Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**ab** Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	**a+** Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	**ab+** Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

Reading File

File object includes the following methods to read data from the file.

- read(chars): reads the specified number of characters starting from the current position.
- readline(): reads the characters starting from the current reading position up to a newline character.
- readlines(): reads all lines until the end of file and returns a list object.

The following C:\myfile.txt file will be used in all the examples of reading and writing files.

C:\myfile.txt

This is the first line.

This is the second line.

This is the third line.

The following example performs the read operation using the read(chars) method.

Example: Reading a File

```
>>> f = open('C:\myfile.txt') # opening a file
>>> lines = f.read() # reading a file
>>> lines
'This is the first line. \nThis is the second line.\nThis is the third line.'
>>> f.close() # closing file object
```

Above, f = open('C:\myfile.txt') opens the myfile.txt in the default read mode from the current directory and returns a file object. f.read() function reads all the content until EOF as a string. If you specify the char size argument in the read(chars) method, then it will read that many chars only. f.close() will flush and close the stream.

Reading a Line

The following example demonstrates reading a line from the file.

Example: Reading Lines

```
>>> f = open('C:\myfile.txt') # opening a file
>>> line1 = f.readline() # reading a line
>>> line1
'This is the first line. \n'
>>> line2 = f.readline() # reading a line
```

```
>>> line2
'This is the second line.\n'
>>> line3 = f.readline() # reading a line
>>> line3
'This is the third line.'
>>> line4 = f.readline() # reading a line
>>> line4
''
>>> f.close() # closing file object
```

As you can see, we have to open the file in 'r' mode. The readline() method will return the first line, and then will point to the second line in the file.

Reading All Lines

The following reads all lines using the readlines() function.

Example: Reading a File

Copy

```
>>> f = open('C:\myfile.txt') # opening a file
>>> lines = f.readlines() # reading all lines
>>> lines
'This is the first line. \nThis is the second line.\nThis is the third line.'
>>> f.close() # closing file object
```

The file object has an inbuilt iterator. The following program reads the given file line by line until StopIteration is raised, i.e., the EOF is reached.

Example: File Iterator

```
f=open('C:\myfile.txt')
while True:
    try:
        line=next(f)
        print(line)
    except StopIteration:
        break
```

```
f.close()
```

Use the for loop to read a file easily.

Example: Read File using the For Loop

```
f=open('C:\myfile.txt')
for line in f:
    print(line)
f.close()
```

Output

```
This is the first line.
This is the second line.
This is the third line.
```

Reading Binary File

Use the 'rb' mode in the open() function to read a binary files, as shown below.

Example: Reading a File

```
>>> f = open('C:\myimg.png', 'rb') # opening a binary file
>>> content = f.read() # reading all lines
>>> content
b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x00\x08\x00\x00\x00\x08\x08\x06
\x00\x00\x00\xc4\x0f\xbe\x8b\x00\x00\x00\x19tEXtSoftware\x00Adobe ImageReadyq
\xc9e\x00\x00\x00\x8dIDATx\xdab\xfc\xff\xff?\x03\x0c0/zP\n\xa4b\x818\xeco\x9c
\xc2\r\x90\x18\x13\x03*8\t\xc4b\xbc\x01\xa8X\x07$\xc0\xc8\xb4\xf0>\\\x11P\xd7?
\xa0\x84\r\x90\xb9\t\x88?\x00q H\xc1C\x16\xc9\x94_\xcc\x025\xfd2\x88\xb1\x04
\x88\x85\x90\x14\xfc\x05\xe2( \x16\x00\xe2\xc3\x8c\xc8\x8e\x84:\xb4\x04H5\x03
\xf1\\ .bD\xf3E\x01\x90\xea\x07\xe2\xd9\xaeB`\x82'
>>> f.close() # closing file object
```

Writing to a File

The file object provides the following methods to write to a file.

- write(s): Write the string s to the stream and return the number of characters written.
- writelines(lines): Write a list of lines to the stream. Each line must have a separator at the end of it.

Create a new File and Write

The following creates a new file if it does not exist or overwrites to an existing file.

Example: Create or Overwrite to Existing File

Copy

```
>>> f = open('C:\myfile.txt','w')
>>> f.write("Hello") # writing to file
5
>>> f.close()

# reading file
>>> f = open('C:\myfile.txt','r')
>>> f.read()
'Hello'
>>> f.close()
```

In the above example, the `f=open("myfile.txt","w")` statement opens `myfile.txt` in write mode, the `open()` method returns the file object and assigns it to a variable `f`. `'w'` specifies that the file should be writable. Next, `f.write("Hello")` overwrites an existing content of the `myfile.txt` file. It returns the number of characters written to a file, which is 5 in the above example. In the end, `f.close()` closes the file object.

Appending to an Existing File

The following appends the content at the end of the existing file by passing `'a'` or `'a+'` mode in the `open()` method.

Example: Append to Existing File

```
>>> f = open('C:\myfile.txt','a')
>>> f.write(" World!")
7
>>> f.close()

# reading file
>>> f = open('C:\myfile.txt','r')
```

```
>>> f.read()
'Hello World!'
>>> f.close()
```

Write Multiple Lines

Python provides the `writelines()` method to save the contents of a list object in a file. Since the newline character is not automatically written to the file, it must be provided as a part of the string.

Example: Write Lines to File

```
>>> lines=["Hello world.\n", "Welcome to TutorialsTeacher.\n"]
>>> f=open("D:\myfile.txt", "w")
>>> f.writelines(lines)
>>> f.close()
```

Opening a file with "w" mode or "a" mode can only be written into and cannot be read from. Similarly "r" mode allows reading only and not writing. In order to perform simultaneous read/append operations, use "a+" mode.

Writing to a Binary File

The `open()` function opens a file in text format by default. To open a file in binary format, add `'b'` to the mode parameter. Hence the `"rb"` mode opens the file in binary format for reading, while the `"wb"` mode opens the file in binary format for writing. Unlike text files, binary files are not human-readable. When opened using any text editor, the data is unrecognizable.

The following code stores a list of numbers in a binary file. The list is first converted in a byte array before writing. The built-in function bytearray() returns a byte representation of the object.

Example: Write to a Binary File

```
f=open("binfile.bin","wb")
num=[5, 10, 15, 20, 25]
arr=bytearray(num)
```

```
f.write(arr)
f.close()
```

Python Delete File

Delete a File

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

Example

Remove the file "demofile.txt":

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

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

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

Example

Remove the folder "myfolder":

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

6.6. Manipulating file pointer using seek()

Python seek() function

The concept of file handling is used to preserve the data or information generated after running the program. Like other programming languages like C, C++, Java, Python also support file handling.

seek() method

In Python, **seek()** function is used to **change the position of the File Handle** to a given specific position. File handle is like a cursor, which defines from where the data has to be read or written in the file.

Syntax: f.seek(offset, from_what), where f is file pointer

Parameters:

Offset: Number of positions to move forward

from_what: It defines point of reference.

Returns: Does not return any value

The reference point is selected by the **from_what** argument. It accepts three values:

- **0:** sets the reference point at the beginning of the file
- **1:** sets the reference point at the current file position
- **2:** sets the reference point at the end of the file

By default from_what argument is set to 0.

Note: Reference point at current position / end of file cannot be set in text mode except when offset is equal to 0.

Example 1: Let's suppose we have to read a file named "seek.txt" which contains the following text:

"Code is like humor. When you have to explain it, it is bad."

```
# Opening "seek.txt" text file
f = open("seek.txt", "r")
```

Second parameter is by default 0, sets Reference point to 20th index position from the beginning.

```
f.seek(20)
# prints current position
print(f.tell())
print(f.readline())
f.close()
```

Output:

20

When you have to explain it, it’s bad.

Example 2: Seek() function with negative offset only works when file is opened in binary mode. Let’s suppose the binary file contains the following text.

b'Code is like humor. When you have to explain it, it is bad.'

```
#Opening "seek.txt" text file in binary mode
f = open("seek.txt", "rb")


# sets Reference point to 10th position to the left from end.
f.seek(-10, 2)
# prints current position
print(f.tell())
# Converting binary to string and printing.
print(f.readline().decode('utf-8'))
f.close()
```

Output:

47

it is bad.

6.7. Programming using file operations