Day 6 - Files, Modules and Stuff

Stuff

Few topics we missed in the classes but are important in programming.

break

The **break** keyword is used int the loops to break the execution of the loops and get out of it.

Commmon usecase is if we are checking a condition to match and once it matches we want to exit the loop.

```
count = 0

for data in range(1,11):
    if count > 5:
        break
    print(data**5)
    count = count + 1
```

continue

The **continue** keyword is again used in the loops to skip the remaining statments of the loop and start the next iteration.

```
```

In [11]:

```
count = 0

# print the powers of 5 from 0 to 5.
for data in range(1,11):
    if count > 5:
        break
    print(data**5)
    count = count + 1
```

```
1
32
243
1024
3125
7776
```

In [12]:

```
#assignment 1

count = 0

# print the powers of 5 from 0 to 5.
for data in range(1,11):
    if count >= 5 :
        continue
    print(data**5)
    count = count + 1
```

In [14]:

```
#strings - split()

my_string = "anshul-ashish-ashutosh"

my_string.split('-')
my_string.split()
```

Out[14]:

```
['anshul-ashish-ashutosh']
```

In [16]:

```
filename = "c:\\users\\arpit\\Dekstop\\my_file.txt"

filename.split("\\")[-1]
```

Out[16]:

```
'my_file.txt'
```

In [17]:

```
filename.split("\\")[-1]  ## my_file.txt
filename.split("\\")[-1].split('.')
```

Out[17]:

```
['my_file', 'txt']
```

Files

Working with Files is one of the most important tasks we do in data science and machine learning as it's impossible to generate data or ask it from the user everytime.

The files contain data for our use and we can load and manipulate that data for our special purposes. Now we will learn how to read and write files.

There are basically 3 steps while working with files.

1. Open the file
2. Read the data from the file
3. Close the file

For python each of these tasks have special functions and we will use them to work with

```
open() - to open the file and give us a file object

read() - to read the contents of the file and give us a string

close() - to close the file object
```

In [20]:

```
f = open('./datasets/wine.csv') # open the file and make the file object as f
```

In [18]:

```
f = open('./datasets/wine2.csv') # open the file and make the file object as f
```

```
---------------------------------------------------------------------
---
FileNotFoundError                         Traceback (most recent call la
st)
<ipython-input-18-6c537f8df1a2> in <module>()
----> 1 f = open('./datasets/wine2.csv') # open the file and make th
e file object as f

FileNotFoundError: [Errno 2] No such file or directory: './datasets/
wine2.csv'
```

In [21]:

```
data = f.read() # read all the contents of the file as a string
print(data[0:400])  # print only first 400 characters
```

```
class_label,class_name,alcohol,malic_acid,ash,alcalinity_of_ash,magn
esium,total_phenols,flavanoids,nonflavanoid_phenols,proanthocyanins,
color_intensity,hue,od280,proline
1,Barolo,14.23,1.71,2.43,15.6,127,2.8,3.06,0.28,2.29,5.64,1.04,3.92,
1065
1,Barolo,13.2,1.78,2.14,11.2,100,2.65,2.76,0.26,1.28,4.38,1.05,3.4,1
050
1,Barolo,13.16,2.36,2.67,18.6,101,2.8,3.24,0.3,2.81,5.68,1.03,3.17,1
185
1,Barolo,14.3
```

In [22]:

```
print(f.read())
```

Now once the data has been read there is nothing to read more. So if you use the `read()` function again, you will get empty string. To read again, close and open the file again.

In [3]:

```
f.close()  #close the file object
```

What to do when you want to go back again at the top. For this file object has function

`seek(position)` --> go to the position at that character in the file.

Though it's a functionality give it's rarely used for our specific purposes.

In [23]:

```
f = open('./datasets/wine.csv')  # open the file
data = f.read()  # read the full file
print(data[:100])
f.seek(12)                       # set the cursor to 12th character
data = f.read()                  # read the file from 12th character to last lin
e
print(data[:100])                # print 100 characters
f.close()                        # close the file.
```

```
class_label,class_name,alcohol,malic_acid,ash,alcalinity_of_ash,magn
esium,total_phenols,flavanoids,n
class_name,alcohol,malic_acid,ash,alcalinity_of_ash,magnesium,total_
phenols,flavanoids,nonflavanoid_
```

By default, python opens the file for reading only. What if we want to write to the file. We have a separate argument in the `open()` function which takes the mode in which we open the file.

eg.

```
f = open(filename, ['r']['w']['a']['b']['+'])
```

Info about each mode

mode word	details
r	read only mode
w	write only mode. Overwrites the previous content in the file
a	append mode. Adds the content at the end of the file.
b	opens the file in binary mode for images, non text files
+	used with any of above mode for both reading and writing

In [24]:

```
# let's do some writing

f = open('sampleFile.txt','w')
f.write("This is a sample file and it is just created")
f.close()

# see the file in the current directory
```

In [25]:

```
#open and read the content of the file
f  = open('sampleFile.txt')
data = f.read()
print(data)
f.close()
```

```
This is a sample file and it is just created
```

Opening a file in write mode will automatically remove the previous content

In [26]:

```
# let's do some writing

f = open('sampleFile.txt','w')
f.write("This is a sample file again  and it is just created again")
f.close()

#open and read the content of the file
f  = open('sampleFile.txt')
data = f.read()
print(data)
f.close()
```

```
This is a sample file again  and it is just created again
```

In [27]:

```
### question if we move close before print.
# let's do some writing

f = open('sampleFile.txt','w')
f.write("This is a sample file again  and it is just created again")
f.close()

#open and read the content of the file
f  = open('sampleFile.txt')
data = f.read()
f.close()
print(data)
```

```
This is a sample file again  and it is just created again
```

We see that the previous line have been gone. To add the content to the previous content we use the append mode with

```
a
```

In [28]:

```
f = open('sampleFile.txt','a')
f.write("\n\nThis is a new line that we are writing and we should get all conten
t now.")
f.close()

#open and read the content of the file
f  = open('sampleFile.txt')
data = f.read()
print(data)
f.close()
```

```
This is a sample file again  and it is just created again

This is a new line that we are writing and we should get all content
now.
```

Awesome now we know how to open, read and write to files. There are a couple of more use cases

1. **Read a file line by line**

   Python has different methods to read the content of the file. For this task we have
   A. readline() # read a single line and then read again
   B. readlines() # read all the lines and returns a list
   C. writelines() # write lines from an iterable like a list
2. **What if I forgot to close the file?**

   Generally nothing happens but you might lose your data. It's better to close.

   To overcome this issue of keeping track of closing files, python has a special way to handle automatic closing of the files

   The **with** statement. Internally it's fairly complex and you can learn more about it here (http://effbot.org/zone/python-with-statement.htm).

   For us we will use **with** whenever we want to create some context like things and want to run the python code under that context. eg. We want to open an file and want to do operations on the that file and then it will automatically handle everything.

```
with open(filename, mode) as f:
    f.read()
    f.write()
    # don't need to do f.close() as it will be automatically done
 after the block end.
```

In [29]:

```
with open('sampleFile.txt') as f:
    data = f.read()
    print(data)
```

```
This is a sample file again  and it is just created again

This is a new line that we are writing and we should get all content
now.
```

BREAK

We will use the `with` statement many times later when working with Tensorflow but here we generally work with files.

This concludes our simple file read write section. Let's see some problems

1. I don't want the full data in string. Why?? Because I don't want to manually see each value
2. In wine data set, each row represents data and the first row is header. Is there some way we can get the values in tabular form or like in dictionary

Here comes the modules section

Modules

Modules are the files in python which provide use more functionality than the standard python. Once we learn how to program and understand the syntax of python we want to do much more stuff.

Modules in python are files which contains classes, function and variables which can be used by others and make our lives easier.

Since modules are external we need to bring them into our program (we call them **importing a module**

How to import a module

we have a **import** keyword which does this for us and provide it a module name like

```
import math  #imports the math module
```

To use an object from that module we use the syntax as

```
pie = math.pi    # use the module name and then use the **dot** to get the
 object as we do for methods in string
print(pie)
```

If we want to import only specific objects from a module we use the **from** keyword as

```
from math import pi
print(pi)
```

Sometimes modules names are long as we will see the in the course later and to use objeects we have to use the full name. To simplify this we use **as** keyword to rename a module.

```
import math as mm
print(mm.pi)
```

In [ ]:

```
from sklearn.preprocessing import StandardScaler as scaler
```

In [30]:

```
import math

pei = math.pi

print(pei)
```

```
3.141592653589793
```

In [31]:

```
from math import pi  #directly  use the object
print(pi)
```

```
3.141592653589793
```

In [32]:

```
import math as mm
print(mm.pi)
```

```
3.141592653589793
```

Reading special files (csv)

What is csv format?

CSV stands for **Comma Separated Values**. It's one of the most common formats for passing around data in the world other than the **JSON** format which we will learn later.

The values in this file are separated by comma like

> 1,Barolo,14.23,1.71,2.43,15.6,127,2.8,3.06,0.28,2.29,5.64,1.04,3.92,1065

Here each value is separated by comma. Now when reading this type of file if we use the simple file read and write functions of python we get a complete string.

To separate the values we need to use the `split()` function of **string** to get the values in a list and then we need to get the values individually.

In [33]:

```
with open('./datasets/wine.csv') as f:
    line = f.readline()
    print(line)
    header_list = line.split(',')
    print(header_list)
    print(header_list[3])

    for line in f:
        data = line.split(",")
        print(data[3])
        break
```

```
class_label,class_name,alcohol,malic_acid,ash,alcalinity_of_ash,magn
esium,total_phenols,flavanoids,nonflavanoid_phenols,proanthocyanins,
color_intensity,hue,od280,proline

['class_label', 'class_name', 'alcohol', 'malic_acid', 'ash', 'alcal
inity_of_ash', 'magnesium', 'total_phenols', 'flavanoids', 'nonflava
noid_phenols', 'proanthocyanins', 'color_intensity', 'hue', 'od280',
'proline\n']
malic_acid
1.71
```

This becomes tedious after some time and it has some issues like what is some column has multiple values separated by comma. Then this approach will fail.

What can do ? Use a module

csv module

csv modules do the boring work for us and gives the data already separated as we want. We are lazy.

Importing csv module

```
import csv
```

csv modules has few objects which make things easier for us

- **reader** - reads the data from file based on separator
- **writer** - writes the data in **csv** format

Let' see how to use them

In [38]:

```
import csv
with open('./datasets/wine.csv') as f:
    pointer = csv.reader(f, delimiter = ',')
    row = next(pointer)
    for row in pointer:
        print(row)
        break
```

```
['1', 'Barolo', '14.23', '1.71', '2.43', '15.6', '127', '2.8', '3.0
6', '0.28', '2.29', '5.64', '1.04', '3.92', '1065']
```

In [39]:

```
import csv
with open('./datasets/wine.csv') as f:
    reader = csv.reader(f, delimiter = ',')
#     row = next(reader) commented next
    for row in reader:
        print(row)
        break
```

```
['class_label', 'class_name', 'alcohol', 'malic_acid', 'ash', 'alcal
inity_of_ash', 'magnesium', 'total_phenols', 'flavanoids', 'nonflava
noid_phenols', 'proanthocyanins', 'color_intensity', 'hue', 'od280',
'proline']
```

Here we see some new things.

next() - next function gets us the value at the current position in the iterator and the moves the cursor to the next item in iterator. In this case the next row of data in the **csv** file.

We use **next()** function to skip some lines which we do not want some times, eg. skipping the header row or skipping some rows at the start.

For writing the csv files we use the **writer()** function to write to csv. The writer() function takes following important arguments.

1. **file object** - pointer to the file to write
2. **delimiter** - the character used to separate the value.

In [41]:

```
my_data = ["spam","ham","poha","parantha"]

with open("csvwrite.csv","w") as f_out:
    writer = csv.writer(f_out, delimiter = ",")
    writer.writerow(my_data)
```

Working with Dictionaries in csv files

Though plain csv **reader** and **writer** objects give us the ability to read the data properly but what we are getting is still a list for each row and we have to manually remmeber the index of each column to know about what data column we are working with.

It would be much better that if we read each row as a dictionary and the keys of the dictionary tell use the column name and values are the value in that column for the row. This will make it easier to understand the data.

Fortunately, **csv** module provides us with the 2 great classes for this task

1. **DictReader** - Read the data from the file row by row as a dictionary
2. **DictWriter** - Write the data to a csv file from dictionaries.

Both the classes work similary as the **reader()** and **writer()** function we have seen above

Let's see the examples and code snippets.

In [45]:

```
from pprint import pprint
with open('./datasets/wine.csv') as f:
    dict_reader = csv.DictReader(f)
    for row in dict_reader:
        pprint(row)
        break
```

```
{'alcalinity_of_ash': '15.6',
 'alcohol': '14.23',
 'ash': '2.43',
 'class_label': '1',
 'class_name': 'Barolo',
 'color_intensity': '5.64',
 'flavanoids': '3.06',
 'hue': '1.04',
 'magnesium': '127',
 'malic_acid': '1.71',
 'nonflavanoid_phenols': '0.28',
 'od280': '3.92',
 'proanthocyanins': '2.29',
 'proline': '1065',
 'total_phenols': '2.8'}
```

Everything is same as the **reader** only the different class is used.

One extra line used is `pprint()` instead of `print()`. This is done to show the dictionary output beautifully having each key value pair in each line instead of in a single line.

Similarly for the Writing a dictionary

In [46]:

```
dict_to_write = {'age': '20', 'height': '62', 'id': '1', 'weight': '120.6', 'nam
e': 'Alice'}

with open('DictWriterSample.csv','w') as f:
    writer = csv.DictWriter(f,delimiter="|", fieldnames =  dict_to_write.keys())
    writer.writerow(dict_to_write)
```

This concludes our section on working with csv files. For more information in working with csv files, read the documentation (https://docs.python.org/3.1/library/csv.html#csv.DictReader).

DateTime module

During data science we work a lot with the date and time data Since everything come as a string and date and time are a series data so we need something which can show the linearity in time

for that we have `datetime` module which helps us convert the strings to date objects we use this to do many tasks like

1. Convert string to datetime objects
2. Convert date to strings of our choice - US uses **month/day/year** we use **day/month/year** format
3. Do operations on date eg. add days, month, years
4. Do operations like select only data for one month or a week

Insid the `datetime` module we have `datetime` object which we use. So to import it we use

```
import datetime.datetime
```

or

```
from datetime import datetime #commonly used
```

This `datetime` object has a function `strptime()`

which we read as **(str)ing (p)arse (time)**

strptime() --> converts string to datetime object

It has 2 arguments

1. **datestring** --> the string which contains the date eg. *25/05/2018*
2. **format** --> the format in which the string is preset **eg.** *%D/%M/%Y*

The format string uses special characters to represent each part in the datestring. The complete list can be found at documentation. (https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior)

Some coomon are

special character	what it represents
%m	Month in numeric form
%d	Day in numeric form
%Y	Year in YYYY format **eg** *2018*

The seperators will be used as it is. In our example we have **/** as a separator so it will be used as it is.

Example to convert string to date

```
from datetime import datetime

datestring = "22/06/2018"
date_object = datetime.strptime(datestring, "%D/%M/%Y")

print(date_object)
print(type(date_object))
```

Let's see them in code now.

In [47]:

```
from datetime import datetime

my_date = "9/1/2015 6:09"

date_obj = datetime.strptime(my_date,'%m/%d/%Y %H:%M')
print(date_obj)
print(type(date_obj))
```

```
2015-09-01 06:09:00
<class 'datetime.datetime'>
```

How to do operationso on the datetime object?

To do operations such as adding days, months to our object we use the **timedelta** from the **datetime** module as

```
from datetime import timedelta
```

Then we use the `timedelta()` function to add the days or whatever we want as

```
date_obj = date_obj + timedelta(days=1)
print(date_obj
```

Common arguments of `timedelta()` are

- weeks
- days
- hours
- seconds
- minutes

See more at the documentation. (https://docs.python.org/3.2/library/datetime.html#datetime.timedelta)

In [51]:

```
from datetime import timedelta

date_obj = date_obj + timedelta(weeks=6)
print(date_obj)
```

```
2015-10-13 06:09:00
```

In [52]:

```
indian_date = "18/6/2018"
training_start_date = datetime.strptime(indian_date, "%d/%m/%Y")
print(training_start_date)
training_end_date = training_start_date + timedelta(weeks=6)
print(training_end_date)
```

```
2018-06-18 00:00:00
2018-07-30 00:00:00
```

In [53]:

```
diff = training_end_date - training_start_date
print(diff.days)
print(diff.total_seconds())
42*24*60*60
```

```
42
3628800.0
```

Out[53]:

```
3628800
```

Converting datetime objects back to string

We learned how to convert string to datetime

How to do the reverse? Do we need to do the reverse?

Yes. Sometimes dates are in 1 format and we need need to convert to other format.

Eg. US and Rest of the world format

Also sometimes we need to extract some information which is important to us and want to leave rest of infomration.

Eg. We just need to extract the month or week from the full date.

Here comes the new function

strftime() - This works on the datetime object. So the date_obj that we have created. It takes the argument as

formatString - The format in which we want to get the date. This format will be made up of same special characters as see while converting string to datetime object.

Example

```
from datetime import datetime

my_date = "9/1/2015 6:09"

# convert string to date
date_obj = datetime.strptime(my_date,'%m/%d/%Y %H:%M')
print(date_obj)

#convert back to string in indian format and remove the hour and minute da
ta
normal_date = date_obj.strftime('%d/%m/%Y')
print(normal_date)
print(type_normal_data)
```

Use different format strings to get what you want. No one can stop you.

In [54]:

```
my_date = "9/1/2015 6:09"

# convert string to date
date_obj = datetime.strptime(my_date,'%m/%d/%Y %H:%M')
print(date_obj)

#convert back to string in indian format and remove the hour and minute data
normal_date = date_obj.strftime('%d/%m/%Y')
print(normal_date)        #changed to our format
print(type(normal_date))  #it's of type string
```

```
2015-09-01 06:09:00
01/09/2015
<class 'str'>
```

END CLASS

In [ ]:

```

```