

**Python Standard Library**:

Python has a batteries included philosophy which means it comes with a comprehensive library of packages and modules that provide common features we need while building real world applications.

In this section we are going to explore python standard library and more specifically going to learn:

🡪 How to work with files and directories.

🡪 SQLite databases.

🡪 Date and Time objects

🡪 Generate random values

🡪 Send emails

And so on…

*Treat this section as a reference to these topics*.

**Working with Paths**:

In this lecture we will look at *Path* class which is the foundation of working with files and directories.

from pathlib import Path

//pathlib module and Path class

Now we can create a path object in a few different ways.

1. To Create an absolute path:

Path("C:\\Program Files\\Windows")    #Defining Path For Windows

Path("/usr/local/bin")              #Defining Path for mac

Note: When there is a long path, double \\ slashes look ugly, so we can simplify this using a raw string (using *r* before string). *In a raw string backslash is not considered an escape character*.

Path(r "C:\Program Files\Windows")    #Defining Path For Windows

1. Create a path object that represents current folder.

path = Path()

print(path) // .

It shows**.** Means current folder

1. To navigate to a relative path from current folder.

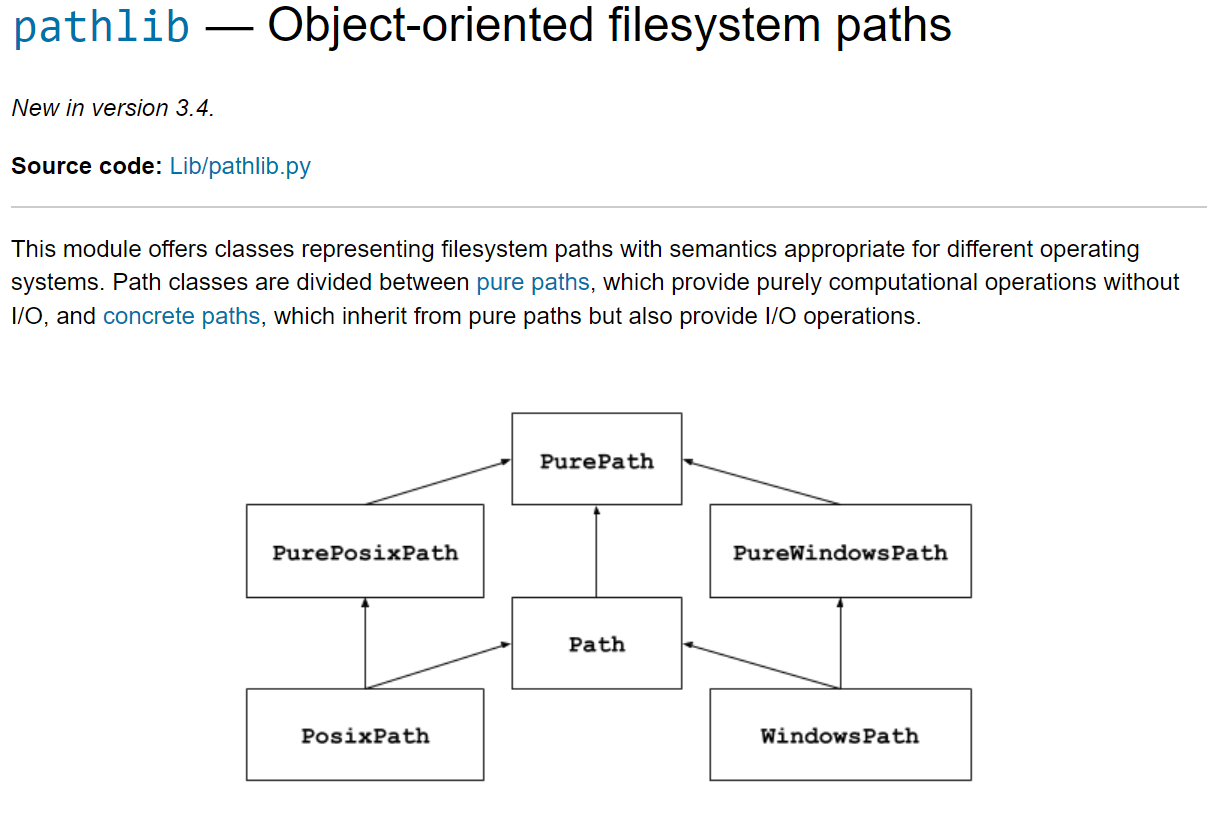
path = Path(r"ecommerce\\_\_init\_\_.py")

1. Combine path objects together.
2. Get the home directory of the current user.

path = Path()

print(path.home()) 🡪C:\Users\himanshu

Note: Refer to *python3 pathlib* documentation for exploring more methods and properties of *Path* class.



Some useful methods are:

1. To see if a file or directory exists (*exists*):

path = Path(r"ecommerce\\_\_init\_\_.py")

path2 = Path(r"ecommerce\\_\_nofile\_\_.nofile")

print(path.exists()) True

print(path2.exists()) False

1. To check if this path represents a file or a directory,

*is\_file* and *is\_dir* methods

path = Path(r"ecommerce\\_\_init\_\_.py") 🡪 a file

path2 = Path(r"ecommerce") 🡪 a directory

print(path.is\_file()) True

print(path2.is\_dir()) True

1. To print the *end part of the path* (*name*)

path = Path(r"ecommerce\\_\_init\_\_.py")

path2 = Path(r"ecommerce")

print(path.name) \_\_init\_\_.py

print(path2.name) ecommerce

1. Print the file name without the extension (*stem*).

path = Path(r"ecommerce\\_\_init\_\_.py")

print(path.stem) \_\_init\_\_

1. To get the extension of the file (*suffix*).

path = Path(r"ecommerce\\_\_init\_\_.py")

print(path.suffix) .py

1. To get the *first part of path* (*parent*)

path = Path(r"ecommerce\\_\_init\_\_.py")

print(path.parent) ecommerce

*Some real project Path methods*:

🡪 To create a new path based on the existing path but only change name and extension of file (*with\_name*).

path = Path(r"ecommerce\\_\_init\_\_.py")

new\_path = path.with\_name("File.txt")

print(new\_path) ecommerce\File.txt

🡪 To get the absolute value of the path (*absolute*):

path = Path(r"ecommerce\\_\_init\_\_.py")

new\_path = path.with\_name("File.txt")

print(new\_path.absolute())

O/P:

C:\Users\himanshu\Desktop\python-mosh\ecommerce\File.txt

Note: This file is not present yet, it’s only representing the path.

🡪 We have a similar method to above method, but we use it to change the extension of the file.

path = Path(r"ecommerce\\_\_init\_\_.py")

new\_path = path.with\_name("File.txt")

print(new\_path.with\_suffix(".py")) ecommerce\File.py

**Working with Directories**:

So here we have a path object that represents a directory,

from pathlib import Path

path = Path("ecommerce")

print(path.is\_dir()) True

*A few useful methods that we need to be aware of*:

print(path.exists())# To see if this path exists

path.mkdir()        # To create a directory

path.rmdir()        # To remove a directory

path.rename("ecommerce2")     # To rename it

These methods we will see in detail,

1. Method *to iterate over a directory*, *iterdir*():

With this method we can get the list of files and directories in this path.

from pathlib import Path

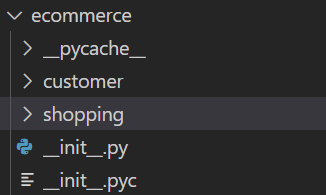
path = Path("ecommerce")

print(path.iterdir()) <generator object Path.iterdir at 0x00000209AD98ACE0>

We get a generator object. *A generator object as the name implies returns a new value every time we iterate*.

So when we are *working with a large list of items*, instead of storing all those items into memory, we use a generator object (*iterate it and get new value every time*). Because it might be possible that our directory might consist of millions of files.

Warning: Some files have been added into our directory, so now our ecommerce directory looks like this:

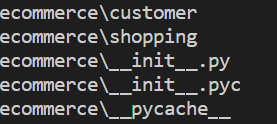


*Here is how we iterate over it*:

path = Path("ecommerce")

for p in path.iterdir():

    print(p)

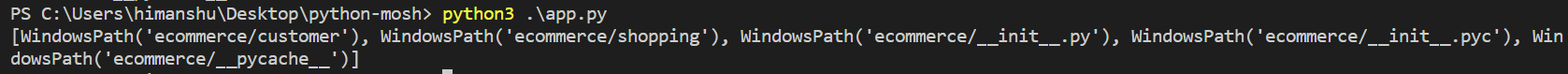


I can see two files and 3 directories in ecommerce directory. We can say this method returns both the files and directories.

If we do not have a million files in our directory then instead we can convert this generator to a list using list comprehension.

print([p for p in path.iterdir()])

O/P:



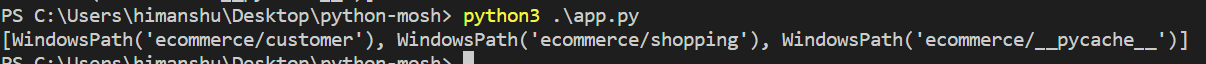
We get an array of WindowsPath objects. The Path class we imported on the top is the base class for two different classes’ viz. *PosixPath* and *WindowsPath*.

*Posix* is the standard use in UNIX like operating systems

Next, if we want to see only directories we can apply a filter in our list comprehension using *is\_dir* method.

print([p for p in path.iterdir() if p.is\_dir()])

O/P:



We only see three directories.

So *iterdir* method is very useful to get the list of files and directories in path but it has two *limitations*.

🡪 We cannot search by a pattern.

🡪 It does not search recursively.

So for these scenarios we use a different method called *glob*.

For example, we can search for all the files with *.py* extension using a pattern in this method,

pyfiles = [p for p in path.glob("\*.py")]

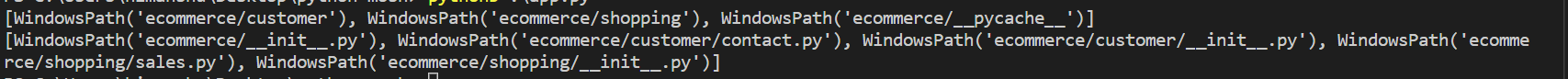
print(pyfiles) // [WindowsPath('ecommerce/\_\_init\_\_.py')]

To search for .py files recursively, use *rglob* method,

pyfiles = [p for p in path.rglob("\*.py")]

print(pyfiles)

O/P:



We get all .py files in ecommerce directory and its children.

**Working with Files**:

In this lecture we will learn about the useful methods to work with files.

from pathlib import Path

path = Path("ecommerce/\_\_init\_\_.py")

So here we have a path object that references a file and it has a couple of useful methods.

# Check whether this file exists

print(path.exists())  # True

# To rename the file

path.rename("init.txt")

# To delete the file

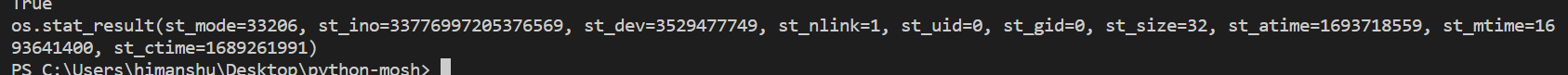
path.unlink()

We have another important method here called stat.

This *stat* method *returns information about this file*.

print(path.stat())

We get a stat result object with some useful attributes.



For example *st\_size* returns size of this file in bytes, *st\_atime* gives last access time, *st\_mtime* returns last modified time and *st\_ctime* gives time of creation.

Note: The time values are in second after epic which is starting time of computer (which is platform dependent). For example on UNIX systems the epic time is January 1st 1970.

So to convert this into human readable time. We need to import *ctime* function from *time* module.

from time import ctime

print(ctime(path.stat().st\_ctime))

# Thu Jul 13 20:56:31 2023

We also have some *methods to read data from the file*.

🡪 *read\_bytes* method

It returns content of the file as a bytes object for representing binary data.

print(path.read\_bytes())

# b'print("ecommerce initialized")\r\n'

🡪 *read\_text* method

Returns the content of the file as a string.

print(path.read\_text())

# print("ecommerce initialized")

Note: Using *read\_text* function is easier then built in open function we used earlier.

Since with open function we have to specify the *file name*, *mode* and use *with* statement to handle opening and closing of the file.

with open("\_\_init\_\_.py", "r") as file: # tedious as compared to read\_text()

    ...

Instead when we use read\_text method, all the magic happens in the background and we no longer have to worry about opening and closing the file.

Similar to these methods we also have

🡪 *write\_text*

🡪 *write\_bytes*

So with this path object, we can do all kinds of operation on a file. However when it comes to copying a file, this path object is not the ideal way.

For example,

source = Path("ecommerce/\_\_init\_\_.py")

target = Path()/ "\_\_init\_\_.py"

We want to copy the file from ecommerce directory (*source*) to file in the current directory (*target*).

*Using path object*,

target.write\_text(source.read\_text())

First read the file from source then write it into target (seems tedious)

*Using shutil module*: (*better approach*)

import shutil

source = Path("ecommerce/\_\_init\_\_.py")

target = Path() / "\_\_init\_\_.py"

shutil.copy(source, target) # cleaner and easier approach then path object

**Working with Zip Files**:

**Working with CSV Files**:

In CSV files, CSV stands for comma separated value and it looks like a simplified spread sheet stored as a plain text file.

For working with CSV files import *csv* module and to create a file use *open* function by using with statement (*better approach*)

import csv

with open("data.csv","w") as file: # use “w” to write into a file

Our *csv* module has a method for creating *csv writer* to which we pass our file object.

with open("data.csv", "w") as file:

    writer = csv.writer(file)

Note: value passed to *writer* method has to be a file object not a path to file, so cannot use Path module here.

Now we have a writer and we can simply use it to write tabular data to our csv file.

with open("data.csv", "w") as file:

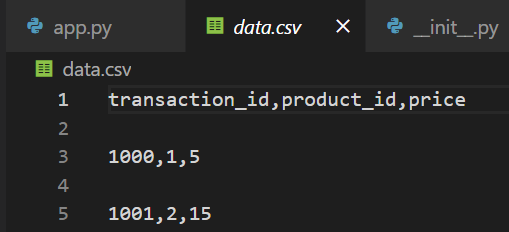
    writer = csv.writer(file)

    writer.writerow(["transaction\_id", "product\_id", "price"])

    writer.writerow([1000, 1, 5])

    writer.writerow([1001, 2, 15])

After running this program, we have a data.csv file in our project folder.



As we can see, it is a simplified spreadsheet where we have a table of data and each line represents a row and our cells are separated using a comma (*very simple way to store data and transfer it from one machine to another*).

*How to read csv file*?

🡪 Same data.csv file, we will open in read mode (*do not supply second argument to open function*).

🡪 Instead of csv writer we will use csv reader.

🡪 Convert this reader to a list object using list function.

with open("data.csv") as file:

    reader = csv.reader(file)

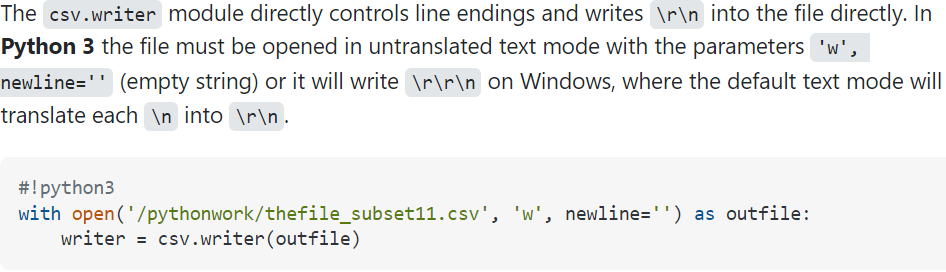
    print(list(reader))

O/P:

I am getting empty list after every item, also there is space between each row of csv file (it might be due to that, but **why**??).

[['transaction\_id', 'product\_id', 'price'], [], ['1000', '1', '5'], [], ['1001', '2', '15'], []]

Answer:



So just use *newline=””* argument in the writer function

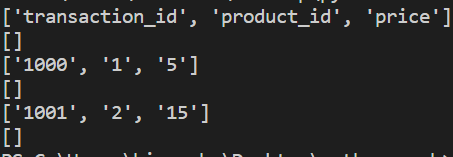
We can iterate over all rows,

with open("data.csv") as file:

    reader = csv.reader(file)

    for row in reader:

        print(row)



Some practice problems we can perform with csv module:

1. Iterate over all the csv files in a directory and remove the first row from each and replace with new values.
2. Combine multiple csv files into one single csv file.
3. Summarize the value from csv files. Suppose we have 10 csv files representing 10 different products. Combine the sales data from all these files and return sum of all sales.

**Working with JSON files**:

JSON short of JavaScript Object Notation is a very popular way to format data in a human readable way.

**Working with SQLite Database**:

JSON short of JavaScript Object Notation is a very popular way to format data in a human readable way.

**Working with Timestamps**:

Basically we have two modules for working with date and time