Session- 7

10 July 2024 11:11 PM

By Alok Ranjan
Linkedin- www.linkedin.com/in/alok-ranjandigu-coder
GitHub- https://github.com/alokyadav2020

Agenda

- List & dictionary comprehensions
- File Handling
- Exception Handling

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
Example:
Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
Without list comprehension you will have to write a `for` statement with a conditional test inside:

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = []


for x in fruits:

  if "a" in x:

    newlist.append(x)


print(newlist)
```

With list comprehension you can do all that with only one line of code:

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]


newlist = [x for x in fruits if "a" in x]


print(newlist)
```

The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Condition

The *condition* is like a filter that only accepts the items that valuate to `True`.

Example

Only accept items that are not "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

The condition `if x != "apple"` will return `True` for all elements other than "apple", making the new list contain all fruits except "apple".
The *condition* is optional and can be omitted:

Example

With no `if` statement:

```
newlist = [x for x in fruits]
```

Iterable

The *iterable* can be any iterable object, like a list, tuple, set etc.

Example

You can use the `range()` function to create an iterable:

```
newlist = [x for x in range(10)]
```

Same example, but with a condition:

Example

Accept only numbers lower than 5:

```
newlist = [x for x in range(10) if x < 5]
```

Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Example

Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

You can set the outcome to whatever you like:

Example

Set all values in the new list to 'hello':

```
newlist = ['hello' for x in fruits]
```

The *expression* can also contain conditions, not like a filter, but as a way to manipulate the outcome:

Example

Return "orange" instead of "banana":

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

File Handling

The key function for working with files in Python is the `open()` function.
The `open()` function takes two parameters; *filename*, and *mode*.
There are four different methods (modes) for opening a file:
`"r"` - Read - Default value. Opens a file for reading, error if the file does not exist
`"a"` - Append - Opens a file for appending, creates the file if it does not exist
`"w"` - Write - Opens a file for writing, creates the file if it does not exist
`"x"` - Create - Creates the specified file, returns an error if the file exists
In addition you can specify if the file should be handled as binary or text mode
`"t"` - Text - Default value. Text mode
`"b"` - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because `"r"` for read, and `"t"` for text are the default values, you do not need to specify them.
Note: Make sure the file exists, or else you will get an error.

Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:
`"a"` - Append - will append to the end of the file
`"w"` - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")

f.write("Now the file has more content!")

f.close()


#open and read the file after the appending:

f = open("demofile2.txt", "r")

print(f.read())
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")

f.write("Woops! I have deleted the content!")

f.close()


#open and read the file after the overwriting:

f = open("demofile3.txt", "r")

print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:
`"x"` - Create - will create a file, returns an error if the file exist
`"a"` - Append - will create a file if the specified file does not exist
`"w"` - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

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

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Open a File on the Server

Assume we have the following file, located in the same folder as Python:
demofile.txt

Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!

To open the file, use the built-in `open()` function.
The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

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

If the file is located in a different location, you will have to specify the file path, like this:

Example

Open a file on a different location:

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

Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the `readline()` method:

Example

Read one line of the file:

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

By calling `readline()` two times, you can read the two first lines:

Example

Read two lines of the file:

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

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

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

Close Files

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

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

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the 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")
```

Note: You can only remove *empty* folders.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
These exceptions can be handled using the `try` statement:

Example

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

```
try:
  print(x)
except:
  print("An exception occurred")
```

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

Since the try block raises an error, the except block will be executed.
Without the try block, the program will crash and raise an error:

Example

This statement will raise an error, because `x` is not defined:

```
print(x)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

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

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
```

```
  print("Something else went wrong")
```

Else

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

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

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

The program can continue, without leaving the file object open.

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.
To throw (or raise) an exception, use the `raise` keyword.

Example

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

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

The `raise` keyword is used to raise an exception.
You can define what kind of error to raise, and the text to print to the user.

Example

Raise a TypeError if x is not an integer:

```
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```