NextGen Academy
Towards fulfilling a million dreams

Python Programming

Module III: Dictionaries , Functions and Modules

June 20, 2023

Contents

This unit covers

- Introduction to Dictionaries, Functions, and Modules.
- File Operations: Opening and closing files, different file modes, reading and writing to files, and manipulating directories.
- Exception Handling: Understanding exceptions, keywords like try, catch, except, else, finally, and how to raise exceptions.
- Exploring Regular Expressions: Concepts, types of regular expressions, and using the match function.

1 Opening and closing file

1.1 File Handling

File handling is an important part of programming, and any programmer needs to know how to open, read, and write files in Python. A computer file is a collection of information that can be accessed, read, and modified with the help of a computer program. Python's in-built functions and methods make it easy to manage files effectively. This article will cover the basics of working with Python files, including opening, reading, and writing in a variety of modes.

1.2 Opening a File in Python

Python's built-in function open() is used to open files. The file name (including the path to the file) and the mode in which the file should be opened are the first and second arguments provided to the open() function, respectively.

1.2.1 Syntax of the open() function

file = open("filename", "mode")

The second argument, mode, is optional, and if not specified, it defaults to "r" (read-only mode).

1.2.2 Opening modes in Python

- **"r" mode for reading:** This mode is used when we want to read a file. It is the default mode if no mode is specified. If the file does not exist, an error will occur.
- **"w" mode for writing:** This mode is used when we want to write to a file. If the file does not exist, it will be created. If it exists, the contents of the file will be truncated (deleted) and replaced with the new data.
- **"a" mode for appending:** This mode is used when we want to append data to an existing file. If the file does not exist, it will be created.
- **"x" mode for exclusive creation:** This mode is used when we want to create a new file, but it should not exist already. If the file exists, an error will occur.
- **"+" mode for updating:** This mode is used when we want to read and write to a file. It allows us to update the existing data in a file.

1.2.3 Examples of opening files in Python

To open a file named "example.txt" in read-only mode:

file = open("example.txt", "r")

To open a file named "newfile.txt" in write mode:

file = open("newfile.txt", "w")

1.3 Closing a File in Python

It is crucial to use the close() method to close a file after you have finished reading from or writing to it. This ensures that all data is written to the file and releases any system resources that the file was utilizing.

A. Using the close() method:

file = open("example.txt", "r") content = file.read() file.close()

In this example, we opened a file in read mode, read its contents, and then closed the file using the close() method.

B. Best practices for closing files:

When you are finished working with a file, always close it.

When you are finished working with a file, use the with statement to have it automatically close.

When working with multiple files, avoid leaving files open for extended periods of time.

C. Examples of closing files in Python:

*# Example **using** with statement with open("example.txt", "r") as file: content = file.read() # Example of manually closing a file file = open("example.txt", "r") content = file.read() file.close()*

2 Various types of file modes

2.1 Opening Files in Python

The **open()** Python method is the primary file handling function. The basic syntax is:

file_object = open('file_name', 'mode')

The **open()** function takes two elementary parameters for file handling:

1. The **file_name** includes the file extension and assumes the file is in the current working directory. If the file location is elsewhere, provide the absolute or relative path.

2. The **mode** is an optional parameter that defines the file opening method. The table below outlines the different possible options:

Mode	Description
'r'	Reads from a file and returns an error if the file does not exist (**default**).
'w'	Writes to a file and creates the file if it does not exist or overwrites an existing file.
'x'	Exclusive creation that fails if the file already exists.

Mode	Description
'a'	Appends to a file and creates the file if it does not exist or overwrites an existing file.
'b'	Binary mode. Use this mode for non-textual files, such as images.
't'	Text mode. Use only for textual files (**default**).
'+'	Activates read and write methods.

The mode must have exactly one create(**x**)/read(**r**)/write(**w**)/append(**a**) method, at most one +. Omitting the mode defaults to **'rt'** for reading text files.

Below is a table describing how each of the modes behave when invoked.

Behavior	Modes
Read	**r**, **r+**, **w+**, **a+**, **x+**
Write	**r+**, **w**, **w+**, **a**, **a+**, **x+**
Create	**w**, **w+**, **a**, **a+**, **x**, **x+**
Pointer Position Start	**r**, **r+**, **w**, **w+**, **x**, **x+**
Pointer Position End	**a**, **a+**
Truncate (clear contents)	**w**, **w+**
Must Exist	**r**, **r+**
Must Not Exist	**x**, **x+**

2.1.1 Read Mode

The read mode in Python opens an existing file for reading, positioning the pointer at the file's start.

> **Note:** If the file does not exist, Python throws an error.

To read a text file in Python, load the file by using the **open()** function:

f = open("<file name>")

The mode defaults to read text (**'rt'**). Therefore, the following method is equivalent to the default:

f = open("<file name>", "rt")

To read files in binary mode, use:

f = open("<file name>", "rb")

Add + to open a file in read and write mode:

f = open("<file name>", "r+") # Textual read and writef = open("<file name>", "rt+") # Same as abovef = open("<file name>", "rb+") # Binary read and write

In all cases, the function returns a file object and the characteristics depend on the chosen mode.

2.1.2 Write Mode

Write mode creates a file for writing content and places the pointer at the start. If the file exists, write truncates (clears) any existing information.

> **Warning:** Write mode deletes existing content ***immediately***. Check if a file exists before overwriting information by accident.

To open a file for writing information, use:

f = open("<file name>", "w")

The default mode is text, so the following line is equivalent to the default:

f = open("<file name>", "wt")

To write in binary mode, open the file with:

f = open("<file name>", "wb")

Add + to allow reading the file:

f = open("<file name>", "w+") # Textual write and readf = open("<file name>", "wt+") # Same as abovef = open("<file name>", "wb+") # Binary write and read

The **open()** function returns a file object whose details depend on the chosen modes.

2.1.3 Append Mode

Append mode adds information to an existing file, placing the pointer at the end. If a file does not exist, append mode creates the file.

> **Note:** The key difference between write and append modes is that append does not clear a file's contents.

Use one of the following lines to open a file in append mode:

f = open("<file name>", "a") # Text appendf = open("<file name>", "at") # Same as abovef = open("<file name>", "ab") # Binary append

Add the + sign to include the read functionality.

> **Note:** Learn how to append a string in Python.

2.1.4 Create Mode

Create mode (also known as exclusive create) creates a file only if it doesn't exist, positioning the pointer at the start of the file.

> **Note:** If the file exists, Python throws an error. Use this mode to avoid overwriting existing files.

Use one of the following lines to open a file in create mode:

f = open("<file name>", "x") # Text createf = open("<file name>", "xt") # Same as abovef = open("<file name>", "xb") # Binary create

Add the + sign to the mode include reading functionality to any of the above lines.

3 Reading and writing to files

Python offers various methods to read and write to files where each functions behaves differently. One important thing to note is the file operations mode. To read a file, you need to open the file in the read or write mode. While to write to a file in Python, you need the file to be open in write mode.

Here are some of the functions in Python that allow you to read and write to files:

- **read() :** This function reads the entire file and returns a string
- **readline() :** This function reads lines from that file and returns as a string. It fetch the line n, if it is been called nth time.
- **readlines() :** This function returns a list where each element is single line of that file.
- **readlines() :** This function returns a list where each element is single line of that file.
- **write() :** This function writes a fixed sequence of characters to a file.
- **writelines() :** This function writes a list of string.
- **append() :** This function append string to the file instead of overwriting the file.

Let's take an example file "abc.txt", and read individual lines from the file with a for loop:

#open the file

text_file = open('/Users/pankaj/abc.txt','r')

#get the list of line

line_list = text_file.readlines();

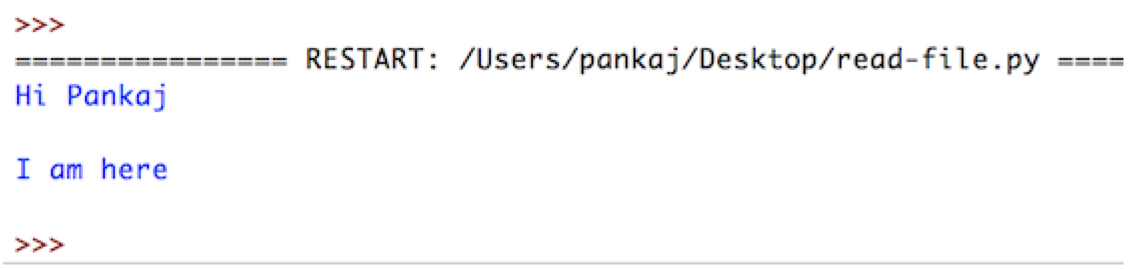
#for each line from the list, print the line

for line in line_list:

print(line)

text_file.close() #don't forget to close the file

Output:



```
>>>
================ RESTART: /Users/pankaj/Desktop/read-file.py ====
Hi Pankaj

 I am here

>>>
```

Now, that we know how to read a file in Python, let's move ahead and perform a write operation here with the writelines() function.

#open the file

text_file = open('/Users/pankaj/file.txt','w')

#initialize an empty list

word_list= []

#iterate 4 times

for i in range (1, 5):

print("Please enter data: ")

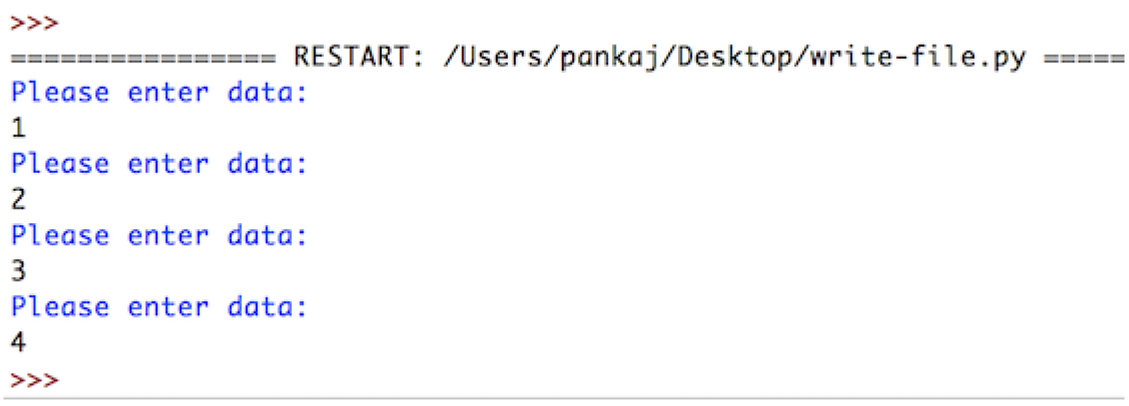
line = input() #take input

word_list.append(line) #append to the list

text_file.writelines(word_list) #write 4 words to the file

text_file.close() #don't forget to close the file

Output



```
>>>
================ RESTART: /Users/pankaj/Desktop/write-file.py =====
Please enter data:
1
Please enter data:
2
Please enter data:
3
Please enter data:
4
>>>
```

3.1 Writing Files in Python

Before we can write to a file in Python, it must first be opened in a different file opening mode. We can do this by supplying the ***open()*** method with a special argument.

In Python, write to file using the **open()** method. You'll need to pass both a filename and a special character that tells Python we intend to write to the file.

Add the following code to write.py. We'll tell Python to look for a file named "sample.txt" and overwrite its contents with a new message.

```
# open the file in write mode
myfile = open("sample.txt",'w')

myfile.write("Hello from Python!")
```

Passing 'w' to the **open()** method tells Python to open the file in write mode. In this mode, any data already in the file is lost when the new data is written.

If the file doesn't exist, Python will create a new file. In this case, a new file named "sample.txt" will be created when the program runs.

Run the program using the Command Prompt:

```
>python write.py
```

Python can also write multiple lines to a file. The easiest way to do this is with the *writelines()* method.

```
# open the file in write mode
myfile = open("sample.txt",'w')

myfile.writelines("Hello World!","We're learning Python!")

# close the file
myfile.close()
```

We can also write multiple lines to a file using special characters:

```
# open the file in write mode
myfile = open("poem.txt", 'w')

line1 = "Roses are red.\n"
line2 = "Violets are blue.\n"
line3 = "Python is great.\n"
line4 = "And so are you.\n"
```

```
myfile.write(line1 + line2 + line3 + line4)
```

Using string concatenation makes it possible for Python to save text data in a variety of ways.

However, if we wanted to avoid overwriting the data in a file, and instead append it or change it, we'd have to open the file using another file opening mode.

4 Manipulating directories

A directory is a collection of files and subdirectories. A directory inside a directory is known as a subdirectory.

Python has the os module that provides us with many useful methods to work with directories (and files as well).

4.1 Get Current Directory in Python

We can get the present working directory using the getcwd() method of the os module.

This method returns the current working directory in the form of a string. **For example:**

```
import os

print(os.getcwd())

# Output: C:\Program Files\PyScripter
```

Here, getcwd() returns the current directory in the form of a string.

4.2 Changing Directory in Python

In Python, we can change the current working directory by using the chdir() method.

The new path that we want to change into must be supplied as a string to this method. And we can use both the forward-slash / or the backward-slash \ to separate the path elements.

Let's see an example:

```
import os

# change directory
os.chdir('C:\\Python33')

print(os.getcwd())

Output: C:\Python33
```

Here, we have used the chdir() method to change the current working directory and passed a new path as a string to chdir().

4.3 List Directories and Files in Python

All files and sub-directories inside a directory can be retrieved using the listdir() method.

This method takes in a path and returns a list of subdirectories and files in that path.

If no path is specified, it returns the list of subdirectories and files from the current working directory.

```
import os

print(os.getcwd())
C:\Python33

# list all sub-directories
os.listdir()
['DLLs',
'Doc',
'include',
'Lib',
'libs',
'LICENSE.txt',
'NEWS.txt',
'python.exe',
'pythonw.exe',
'README.txt',
'Scripts',
'tcl',
'Tools']

os.listdir('G:\\')
['$RECYCLE.BIN',
'Movies',
'Music',
'Photos',
'Series',
'System Volume Information']
```

4.4 Making a New Directory in Python

In Python, we can make a new directory using the mkdir() method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

```
os.mkdir('test')


os.listdir()
['test']
```

4.5 Renaming a Directory or a File

The rename() method can rename a directory or a file.

For renaming any directory or file, rename() takes in two basic arguments:

- the old name as the first argument
- the new name as the second argument.

Let's see an example:

```
import os


os.listdir()
['test']


# rename a directory
os.rename('test','new_one')


os.listdir()
['new_one']
```

Here, 'test' directory is renamed to 'new_one' using the rename() method.

4.6 Removing Directory or File in Python

In Python, we can use the remove() method or the rmdir() method to remove a file or directory.

First let's use remove() to delete a file,

```
import os
```

# delete "myfile.txt" file

os.remove("myfile.txt")

Here, we have used the remove() method to remove the "myfile.txt" file.

Now let's use rmdir() to delete an empty directory,

import os

# delete the empty directory "mydir"

os.rmdir("mydir")

In order to remove a non-empty directory, we can use the rmtree() method inside the shutil module.

For example

import shutil

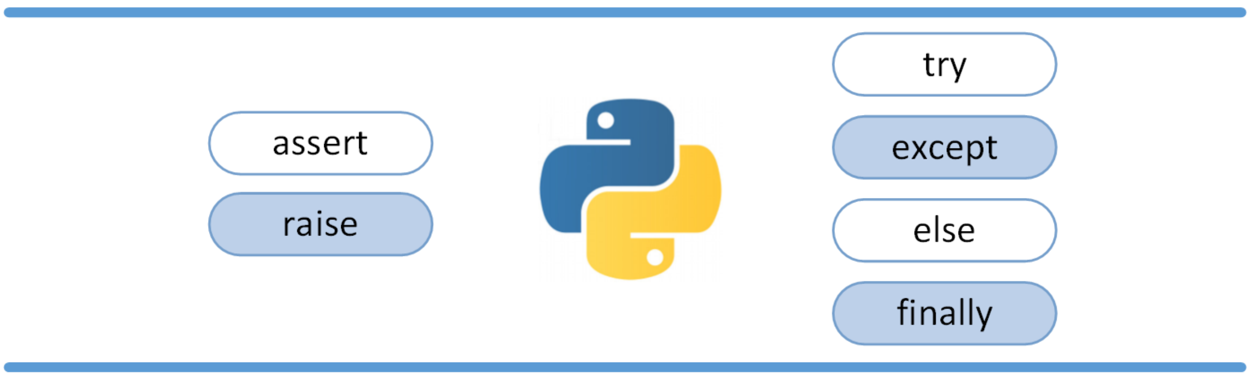
# delete "mydir" directory and all of its contents

shutil.rmtree("mydir")

It's important to note that these functions permanently delete the files or directories, so we need to careful when using them.

5 Exception Handling

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception. In this article, you will see what an exception is and how it differs from a syntax error. After that, you will learn about raising exceptions and making assertions. Then, you'll finish with a demonstration of the try and except block.



6 Various keywords to handle exception such try, catch, except, else, finally, raise

In Python, an exception is an event that occurs during the execution of a program and disrupts the normal flow of the program. It represents an error or an exception condition that the program encounters and cannot handle by itself.

When an exception occurs, it is "raised" or "thrown" by the Python interpreter. The exception then propagates up the call stack, searching for an exception handler that can catch and handle the exception. If no suitable exception handler is found, the program terminates, and an error message is displayed.

Here's an example of a *ZeroDivisionError* exception being raised and handled using a *try-except* block:

```
try:
result = 10 / 0 # Raises ZeroDivisionError
except ZeroDivisionError:
print("Error: Division by zero!")
```

In this example, the code within the *try* block raises a *ZeroDivisionError* exception when attempting to divide by zero. The exception is caught by the *except* block, and the specified error message is printed, allowing the program to continue execution instead of abruptly terminating.

6.1 Nested Try-Except Blocks

Nested try-except blocks provide a way to handle specific exceptions at different levels of code execution. This technique allows you to catch and handle exceptions more precisely based on the context in which they occur.

Consider the following example:

```
try:
# Outer try block
try:
# Inner try block
file = open("nonexistent_file.txt", "r")
content = file.read()
file.close()
print("File content:", content)
except FileNotFoundError:
print("Error: File not found!")
except:
print("Error: Outer exception occurred!")
```

Output:

```
Error: File not found!
```

In this example, the inner try block attempts to open a file "nonexistent_file.txt" in read mode, which doesn't exist and raises a *FileNotFoundError*. The exception is caught by the inner except block, which prints the error message "Error: File not found!".

Since the exception is handled within the inner except block, the outer except block is not executed. However, if the inner except block was not executed, the exception would propagate to the outer except block, and the code within the outer except block would be executed.

6.2 Catching and Re-Raising Exceptions

Catching and re-raising exceptions is a useful technique when you need to handle an exception at a specific level of code execution, perform certain actions, and then allow the exception to reproduce to higher levels for further handling. Let's explore the example further and discuss its significance.

In the provided code snippet, the *validate_age* function takes an *age* parameter and checks if it is negative. If the age is negative, a *ValueError* is raised using the *raise* keyword. The exception is then caught by the except block that specifies *ValueError* as the exception type.

```
def validate_age(age):
try:
if age < 0:
raise ValueError("Age cannot be negative!")
except ValueError as ve:
print("Error:", ve)
raise # Re-raise the exception
try:
validate_age(-5)
except ValueError:
print("Caught the re-raised exception!")
```

In this case, if the age provided to *validate_age* is -5, the condition *if age < 0* is satisfied, and a *ValueError* is raised with the message "Age cannot be negative!".

The except block then catches the *ValueError* and prints the error message using *print("Error:", ve)*. This step allows you to perform specific actions, such as logging the error or displaying a user-friendly error message.

After printing the error message, the *raise* statement is used to re-raise the caught exception. This re-raised exception propagates to a higher level of code execution, allowing it to be caught by an outer exception handler if present.

The output of this code snippet is:

Error: Age cannot be negative!

Caught the re-raised exception!

This example demonstrates the importance of catching and re-raising exceptions. By catching an exception, performing necessary actions, and re-raising it, you have more control over how the exception is handled at different levels of your code.

7 Regular Expressions

In 1951, mathematician Stephen Cole Kleene described the concept of a regular language, a language that is recognizable by a finite automaton and formally expressible using regular expressions. In the mid-1960s, computer science pioneer Ken Thompson, one of the original designers of Unix, implemented pattern matching in the QED text editor using Kleene's notation.

Since then, regexes have appeared in many programming languages, editors, and other tools as a means of determining whether a string matches a specified pattern. Python, Java, and Perl all support regex functionality, as do most Unix tools and many text editors.

7.1 The re Module

Regex functionality in Python resides in a module named re. The re module contains many useful functions and methods, most of which you'll learn about in the next tutorial in this series.

For now, you'll focus predominantly on one function, re.search().

re.search(<regex>, <string>)

re.search(<regex>, <string>) scans <string> looking for the first location where the pattern <regex> matches. If a match is found, then re.search() returns a **match object**. Otherwise, it returns None.

re.search() takes an optional third <flags> argument that you'll learn about at the end of this tutorial.

7.2 How to Import re.search()

Because search() resides in the re module, you need to import it before you can use it. One way to do this is to import the entire module and then use the module name as a prefix when calling the function:

import re

re.search(...)

Alternatively, you can import the function from the module by name and then refer to it without the module name prefix:

```
from re import search

search(...)
```

The examples in the remainder of this tutorial will assume the first approach shown—importing the re module and then referring to the function with the module name prefix: re.search(). For the sake of brevity, the import re statement will usually be omitted, but remember that it's always necessary.

7.3 First Pattern-Matching Example

Now that you know how to gain access to re.search(), you can give it a try:

```
1»> s = 'foo123bar'

2

3»> # One last reminder to import!

4»> import re

5

6»> re.search('123', s)

7<_sre.SRE_Match object; span=(3, 6), match='123'>
```

Here, the search pattern <regex> is 123 and <string> is s. The returned match object appears on **line 7**. Match objects contain a wealth of useful information that you'll explore soon.

For the moment, the important point is that re.search() did in fact return a match object rather than None. That tells you that it found a match. In other words, the specified <regex> pattern 123 is present in s.

A match object is **truthy**, so you can use it in a Boolean context like a conditional statement:

```
»> if re.search('123', s):

... print('Found a match.')

... else:

... print('No match.')

...

Found a match.
```

The interpreter displays the match object as <_sre.SRE_Match object; span=(3, 6), match='123'>. This contains some useful information.

span=(3, 6) indicates the portion of <string> in which the match was found. This means the same thing as it would in slice notation:

```
»> s[3:6]

'123'
```

In this example, the match starts at character position 3 and extends up to but not including position 6.

match='123' indicates which characters from <string> matched.

This is a good start. But in this case, the <regex> pattern is just the plain string '123'. The pattern matching here is still just character-by-character comparison, pretty much the same as the in operator and .find() examples shown earlier. The match object helpfully tells you that the matching characters were '123', but that's not much of a revelation since those were exactly the characters you searched for.

7.4 Python Regex Metacharacters

The real power of regex matching in Python emerges when <regex> contains special characters called **metacharacters**. These have a unique meaning to the regex matching engine and vastly enhance the capability of the search.

Consider again the problem of how to determine whether a string contains any three consecutive decimal digit characters.

In a regex, a set of characters specified in square brackets ([]) makes up a **character class**. This metacharacter sequence matches any single character that is in the class, as demonstrated in the following example:

```
»> s = 'foo123bar'

»> re.search('[0-9][0-9][0-9]', s)

<_sre.SRE_Match object; span=(3, 6), match='123'>
```

[0-9] matches any single decimal digit character—any character between '0' and '9', inclusive. The full expression [0-9][0-9][0-9] matches any sequence of three decimal digit characters. In this case, s matches because it contains three consecutive decimal digit characters, '123'.

These strings also match:

```
»> re.search('[0-9][0-9][0-9]', 'foo456bar')

<_sre.SRE_Match object; span=(3, 6), match='456'>

»> re.search('[0-9][0-9][0-9]', '234baz')

<_sre.SRE_Match object; span=(0, 3), match='234'>

»> re.search('[0-9][0-9][0-9]', 'qux678')

<_sre.SRE_Match object; span=(3, 6), match='678'>
```

On the other hand, a string that doesn't contain three consecutive digits won't match:

```
»> print(re.search('[0-9][0-9][0-9]', '12foo34'))
```

```
None
```

With regexes in Python, you can identify patterns in a string that you wouldn't be able to find with the in operator or with string methods.

Take a look at another regex metacharacter. The dot (.) metacharacter matches any character except a newline, so it functions like a wildcard:

```
»> s = 'foo123bar'

»> re.search('1.3', s)

<_sre.SRE_Match object; span=(3, 6), match='123'>

»> s = 'foo13bar'

»> print(re.search('1.3', s))

None
```

In the first example, the regex 1.3 matches '123' because the '1' and '3' match literally, and the . matches the '2'. Here, you're essentially asking, "Does s contain a '1', then any character (except a newline), then a '3'?" The answer is yes for 'foo123bar' but no for 'foo13bar'.

These examples provide a quick illustration of the power of regex metacharacters. Character class and dot are but two of the metacharacters supported by the re module. There are many more. Next, you'll explore them fully.

7.5 using match function

Also, whenever we found a match to the regex pattern, Python returns us the Match object. Later we can use the following methods of a re.Match object to extract the matched values and positions.

Method	Meaning
group()	Return the string matched by the regex pattern. See capturing groups.
groups()	Returns a tuple containing the strings for all matched subgroups.
start()	Return the start position of the match.
end()	Return the end position of the match.
span()	Return a tuple containing the (start, end) positions of the match.