1) Explain string concatenation and replication.

String concatenation in Python is the process of combining two or more strings into a single string. This can be done using the "+" operator.

Example:

```
str1 = "Hello"
str2 = "World"
concatenated_str = str1 + " " + str2
print(concatenated_str)
# Output: Hello World
```

String replication in Python is the process of creating multiple copies of a string. This can be done using the "*" operator.

Example:

```
str1 = "Hello "
replicated_str = str1 * 3
print(replicated_str)
 # Output: Hello Hello Hello
```

In both examples, we are using the "+" operator to concatenate two strings and the "*" operator to replicate a string.

---

2) Explain Augmented assignment operator.

The augmented assignment operator in Python is a shorthand way of performing an operation on a variable and then assigning the result back to the same variable. It combines an arithmetic or bitwise operator with the assignment operator.

Example:

```
x = 5
x += 3
# This is equivalent to x = x + 3
print(x)
 # Output: 8

y = "Hello"
y += " World"
# This is equivalent to y = y + " World"
print(y)
# Output: Hello World
```

In the example, x += 3 is equivalent to x = x + 3, and y += " World" is equivalent to y = y + " World". The augmented assignment operator provides a more concise way of updating the value of a variable by combining the operation and assignment into a single statement.

---

3) Differentiate between dictionary and list.

List	Dictionary
The list is a collection of index value pairs like that of the array in C++.	The dictionary is a hashed structure of **the key and value** pairs.
The list is created by placing elements in [ ] separated by commas ","	The dictionary is created by placing elements in { } as "key":"value", each key-value pair is separated by commas ", "
The indices of the list are integers starting from 0.	The keys of the dictionary can be of any data type.
The elements are accessed via indices.	The elements are accessed via key-value pairs.
The order of the elements entered is maintained.	There is no guarantee for maintaining order.
Lists are orders, mutable, and can contain duplicate values.	Dictionaries are unordered and mutable but they cannot contain duplicate keys.

thislist = ["apple", "banana", "cherry"]
print(thislist)
	thisdict = {
	  "brand": "Ford",
	  "model": "Mustang",
	  "year": 1964
	}

---

4) Explain any 7 string methods

- upper(): This method returns a copy of the string with all the characters converted to uppercase

```
    text = "hello world"
print(text.upper())  # Output: HELLO WORLD
```

- lower(): This method returns a copy of the string with all the characters converted to lowercase\.

```
text = "HELLO WORLD"
print(text.lower())  # Output: hello world
```

- capitalize(): This method returns a copy of the string with the first character converted to uppercase and the rest to lowercase\.

```
text = "hello world"
```

```
print(text.capitalize())  # Output: Hello world
```

- strip(): This method returns a copy of the string with leading and trailing whitespace removed.

```
text = "   hello world   "
print(text.strip())  # Output: hello world
```

- replace(): This method returns a copy of the string with all occurrences of a specified substring replaced with another substring.

```
text = "hello world"
print(text.replace("hello", "hi"))  # Output: hi world
```

- split()`: This method returns a list of substrings separated by a specified delimiter.

```
text = "hello,world"
print(text.split(","))  # Output: ['hello', 'world']
```

- join(): This method joins the elements of an iterable \(such as a list\) into a single string using the specified separator.

```
words= ["hello", "world"]
print(" "\.join\(words\)\)  \# Output: hello world
```

---

5) **Explain pattern matching with regular expressions**.

Pattern matching with regular expressions in Python involves using the re module to search for specific patterns within strings. Regular expressions, or regex, are a powerful tool for matching and manipulating text based on patterns.

To use regular expressions in Python, you first import the re module. Then, you can use various functions and methods provided by the module to work with regular expressions, such as re.search(), re.match(), re.findall(), and re.sub().
Here are some common uses of regular expressions in Python:

1. Searching for a pattern within a string:

```
import re
text = "The quick brown fox jumps over the lazy dog"
pattern = r"fox"
result = re.search(pattern, text)
if result:
    print("Pattern found")
```

2. Matching a pattern at the beginning of a string:

```
import re
text = "apple banana cherry"
pattern = r"apple"
result = re.match(pattern, text)
if result:
    print("Pattern found at the beginning")
```

3. Finding all occurrences of a pattern within a string:

```
import re
text = "The quick brown fox jumps over the lazy dog"
pattern = r"the"
result = re.findall(pattern, text, re.IGNORECASE)
print(result)
```

4. Substituting a pattern with a replacement string:

```
import re
text = "The quick brown fox jumps over the lazy dog"
pattern = r"fox"
replacement = "cat"
result = re.sub(pattern, replacement, text)
print(result)
```

Regular expressions provide a flexible and powerful way to work with patterns in text, allowing you to search, match, and manipulate strings based on specific criteria.

---

6) **What is matching multiple pattern with the group? Explain with example**.

Matching Objects:
Say you want to separate the area code from the rest of the phone number. Adding parentheses will create groups in the regex: (\d\d\d)-(\d\d\d-\d\d\d\d). Then you can use the group() match object method to grab the matching text from just one group.

```
import re
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
mo = phoneNumRegex.search('My number is 415-555-4242.')
print(mo.group(1))
areaCode, number = mo.groups()
print("area code:", areaCode)
print("number:", number)
```

Output:

```
415
area code: 415
number: 555-4242
```

Retrieve all the Groups at Once:

If you would like to retrieve all the groups at once, use the groups() method

```
import re
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
mo = phoneNumRegex.search('My number is 415-555-4242.')
print(mo.groups())
```

Output:

```
('415', '555-4242')
```

Match a parenthesis:

If you need to match a parenthesis in your text. For instance, maybe the phone numbers you are trying to match have the area code set in parentheses. In this case, you need to escape the (and) characters with a backslash.

```
import re
phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
print(mo.group(1))
```

Output:

```
(415)
```

The \(and \) escape characters in the raw string passed to re.compile() will match actual parenthesis characters.

Matching Multiple Groups with the Pipe:

The | character is called a pipe. We can use it anywhere we want to match one of many expressions.

```
import re
heroRegex = re.compile (r'Batman|Tina Fey')
mo1 = heroRegex.search('Batman and Tina Fey.')
print(mo1.group())
```

Output:

```
'Batman'
```

---

7) **Explain Optional matching with the question mark .**

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match whether or not that bit of text is there. The “?” character flags the group that precedes it as an optional part of the pattern.

```
import re
batRegex = re.compile(r'Bat(wo)?man')
mo1 = batRegex.search('The Adventures of Batman')
mo2 = batRegex.search('The Adventures of Batwoman')
print(mo1.group())
print(mo2.group())
```

Output:

```
Batman
Batwoman
```

Here, we will search for a pattern with a pattern ‘Batman’ or ‘Batwoman’. The **(wo)?** part of the regular expression means that the pattern wo is an optional group. The regex will match text that has zero instances or one instance of wo in it. This is why the regex matches both ‘Batwoman’ and ‘Batman’.

---

8) What is Pattern matching with zero or more and one or more? Explain.

The * (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again.

```
import re
batRegex = re.compile(r'Bat(wo)*man')
mo1 = batRegex.search('The Adventures of Batman')
mo1.group() #'Batman'

mo2 = batRegex.search('The Adventures of Batwoman')
mo2.group() #'Batwoman'

mo3 = batRegex.search('The Adventures of Batwowowowoman')
mo3.group() #'Batwowowowoman'
```

For 'Batman', the (wo)* part of the regex matches zero instances of wo in the string; for 'Batwoman', the (wo)* matches one instance of wo; and for 'Batwowowowoman', (wo)* matches four instances of wo.
If you need to match an actual star character, prefix the star in the regular expression with a backslash, *.

---

9) Explain pattern matching with Specific repetitions.

Pattern matching with specific repetitions in Python allows you to specify the exact number of times a pattern should be repeated within a string. This is useful when you want to match patterns with a specific number of occurrences, such as a certain number of digits in a phone number or a specific sequence of characters.

In Python, you can use curly braces {} to specify the exact number of repetitions for a pattern. For example, if you want to match a sequence of exactly 5 digits, you can use the pattern \d{5}. This will match any sequence of 5 consecutive digits in a string.

```
import re
haRegex = re.compile(r'(Ha){3}')
mo2 = haRegex.search('Ha')== None
print(mo2)
```

Output:

```
True
```

---

10) Explain Greedy and nongreedy matching.

Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

```
import re
greedyHaRegex = re.compile(r'(Ha){3,5}')
mo1 = greedyHaRegex.search('HaHaHaHaHa')
mo1.group() #'HaHaHaHaHa'

nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
mo2.group() #'HaHaHa
```

---

11) Differentiate between findall() and search().

Search (): search () will return a Match object of the first matched text in the searched string. It returns a Match object only on the first instance of matching text.

```
import re
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
mo.group() #'415-555-9999
```

findall (): Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.

```
import re
phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
 # ['415-555-9999', '212-555-0000']
```

---

12) **What are the different pattern matching character classes? Explain**.

\d: Any numeric digit from 0 to 9.
\D: Any character that is not a numeric digit from 0 to 9.
\w: Any letter, numeric digit, or the underscore character.
(Think of this as matching “word” characters.)
\W: Any character that is not a letter, numeric digit, or the underscore character.
\s: Any space, tab, or newline character. (Think of this as matching “space” characters.)
\S: Any character that is not a space, tab, or newline.

```
xmasRegex = re.compile(r'\d+\s\w+')
xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
#['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

The regular expression \d+\s\w+ will match text that has one or more numeric digits (\d+), followed by a whitespace character (\s), followed by one or more letter/digit/underscore characters (\w+). The findall() method returns all matching strings of the regex pattern in a list.

---

13) How do you make your own character set explain.

There are times when you want to match a set of characters but the short-hand character classes (\d, \w, \s, and so on) are too broad. You can define your own character class using square brackets. For example, the character class [aeiouAEIOU] will match any vowel, both lowercase and uppercase.

```
import re
vowelRegex = re.compile(r'[aeiouAEIOU]')
vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
#['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class [a-zA-Z0-9] will match all lowercase letters, uppercase letters, and numbers.

---

14) Explain caret (^) and dollar ($) with example.

The caret (^) is used to match the start of a string. For example, the pattern ^abc will only match if the string starts with "abc".

```
import re
prg1 = re.compile(r'^Hello')
prg1.search('Hello world!')
prg1.search('He said hello.') == None #True
```
The r'^Hello' regular expression string matches strings that begin with 'Hello'.

The dollar ($) is used to match the end of a string. For example, the pattern de$ will only match if the string ends with "de".

```
import re
p1 = re.compile(r'\d$')
p1.search('Your number is 42')
p1.search('Your number is forty two.') == None #True
```
The r'\d$' regular expression string matches strings that end with a numeric character from 0 to 9.

And you can use the ^ and $ together to indicate that the entire string must match the regex—that is, it’s not enough for a match to be made on some subset of the string.

```
import re
p1 = re.compile(r'^\d+$')
p1.search('1234567890')
p1.search('12345xyz67890') == None #True
p1.search('12 34567890') == None #True
```
The r'^\d+$' regular expression string matches strings that both begin and end with one or more numeric characters.

---

15) Explain wildcard character with example.

The . (or dot) character in a regular expression is called a wildcard and will match any character except for a newline.

```
import re
atRegex = re.compile(r'.at')
atRegex.findall('The cat in the hat sat on the flat mat.') #['cat', 'hat', 'sat', 'lat', 'mat']
```

Remember that the dot character will match just one character, which is why the match for the text flat in the previous example matched only lat. To match an actual dot, escape the dot with a backslash: \..

---

16) List all the regex symbols with exmples.

- The ? matches zero or one of the preceding group.

• The * matches zero or more of the preceding group.

• The + matches one or more of the preceding group.
• The {n} matches exactly n of the preceding group.
• The {n,} matches n or more of the preceding group.
• The {,m} matches 0 to m of the preceding group.
• The {n,m} matches at least n and at most m of the preceding group.
• {n,m}? or *? or +? performs a nongreedy match of the preceding group.
• ^spam means the string must begin with spam.
• spam$ means the string must end with spam.
• The . matches any character, except newline characters.
• \d, \w, and \s match a digit, word, or space character, respectively.
• \D, \W, and \S match anything except a digit, word, or space character, respectively.
• [abc] matches any character between the brackets (such as a, b, or c).
• [^abc] matches any character that isn’t between the bracket

---

17) Explain case-insensitive matching.

Normally, regular expressions match text with the exact casing you specify. But sometimes you care only about matching the letters without worrying whether they’re uppercase or lowercase. To make your regex case-insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile().

```
import re
robocop = re.compile(r'robocop', re.I)
robocop.search('RoboCop is part man, part machine, all cop.').group()
#'RoboCop'

robocop.search('ROBOCOP protects the innocent.').group()
# 'ROBOCOP'
```

---

18) what is the regex string method used for substituting string? Explain.

Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The sub() method for Regex objects is passed two arguments. The first argument is a string to replace any matches.The second is the string for the regular expression. The sub() method returns a string with the substitutions applied.

```
import re
namesRegex = re.compile(r'Agent \w+')
namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
#'CENSORED gave the secret documents to CENSORED.'
```

---

19) **Explain complex regexes with example.**

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the re.compile() function to ignore whitespace and comments inside the regular expression string. This "verbose mode" can be enabled by passing the variable re.VERBOSE as the second argument to re.compile().Now instead of a hard-to-read regular expression like this:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}
(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

you can spread the regular expression over multiple lines with comments like this:

```
phoneRegex = re.compile(r'''(
(\d{3}|\(\d{3}\))? # area code
(\s|-|\.)? # separator
\d{3} # first 3 digits
(\s|-|\.) # separator
\d{4} # last 4 digits
(\s*(ext|x|ext.)\s*\d{2,5})? # extension
)''', re.VERBOSE)
```

Note how the previous example uses the triple-quote syntax (''') to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.The comment rules inside the regular expression string are the same as regular Python code: The # symbol and everything after it to the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part of the text pattern to be matched.This lets you organize the regular expression so it's easier to read.

---

20) **What is re.IGNORECASE, re.DOTALL, and re.VERBOSE? Explain.**

- **re.IGNORECASE:** The re.IGNORECASE allows the regular expression to become case-insensitive.
- **re.DOTALL**: By using re.DOTALL flag, you can modify the behavior of dot (.) character to match the newline character apart from other characters.
- **re.VERBOSE:** It allows representing a regular expression in a more readable way. The Verbose flag treats # character as a comment character and also ignores all the whitespace characters including the line break.

```
import re
p1 = re.compile('Foo', re.IGNORECASE | re.DOTALL)
print(p1)
p1= someRegexValue = re.compile('Foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

```
  print(p1)
```
You can get around this limitation by combining the re.IGNORECASE, re.DOTALL, and re.VERBOSE variables using the pipe character (|), which in this context is known as the bitwise or operator.

---

21) Explain the process of copying, moving and renaming files using shutil module.

Copying Files: To copy a file from one location to another, you can use the shutil.copy() function.

```
import shutil
# Copy a file from source to destination
shutil.copy("source_file.txt", "destination_folder")
```

In this example, the shutil.copy() function takes two arguments: the source file path and the destination folder path. It then copies the file from the source to the destination.

Moving Files: To move a file from one location to another, you can use the shutil.move() function.

```
import shutil
# Move a file from source to destination
shutil.move("source_file.txt", "destination_folder")
```

Similar to copying, the shutil.move() function takes two arguments: the source file path and the destination folder path. It then moves the file from the source to the destination.

Renaming Files: To rename a file, you can use the shutil.move() function and provide the new name as the destination path.

```
import shutil
# Rename a file by moving it to the same directory with a new name
shutil.move("old_name.txt", "new_name.txt")
```

In this example, the shutil.move() function is used to rename the file "old_name.txt" to "new_name.txt" by moving it to the same directory with the new name.
In all these examples, the shutil module simplifies the process of copying, moving, and renaming files in Python by handling low-level details of file operations. It provides a high-level interface for these operations, making it easier to work with files and directories in Python.

---

22) **Explain reading,creating,adding and extracting zip files with example.**

Reading Zip Files: To read the contents of a zip file in Python, you can use the zipfile module. Here's an example of how to read the contents of a zip file:

```
import zipfile
# Open the zip file
with zipfile.ZipFile('example.zip', 'r') as zip_ref:
 # Print the list of files in the zip file
    print(zip_ref.namelist())
 # Extract a specific file from the zip file
    zip_ref.extract('file_to_extract.txt', 'destination_folder')
```

Creating Zip Files: To create a new zip file in Python, you can use the zipfile module to add files to the zip archive. Here's an example of how to create a new zip file and add files to it:

```
import zipfile
# Create a new zip file
with zipfile.ZipFile('new_zip_file.zip', 'w') as zip_ref:
    # Add files to the zip file
    zip_ref.write('file1.txt')
    zip_ref.write('file2.txt')
```

Adding Files to Zip Files: To add files to an existing zip file in Python, you can use the zipfile module to append new files to the zip archive. Here's an example of how to add files to an existing zip file:

```
import zipfile
# Open the existing zip file in append mode
with zipfile.ZipFile('existing_zip_file.zip', 'a') as zip_ref:
    # Add new files to the zip file
    zip_ref.write('new_file1.txt')
    zip_ref.write('new_file2.txt')
```

Extracting Zip Files: To extract files from a zip archive in Python, you can use the zipfile module to extract all or specific files from the zip file. Example of how to extract files from a zip file:

```
import zipfile
# Open the zip file
with zipfile.ZipFile('example.zip', 'r') as zip_ref:
 # Extract all files from the zip file
        zip_ref.extractall('destination_folder')
```

In all these examples, the zipfile module provides a convenient way to work with zip files in Python, allowing you to read, create, add, and extract files from zip archives.

---

23) What are the different modules used to scrape webpages in python? Explain.

1. **Requests:** The requests module is used to send HTTP requests to webpages and retrieve their content. It allows you to make GET and POST requests, handle response data, and work with cookies and headers. Here's an example of how to use the requests module to scrape a webpage:

```
import requests
url = 'https://example.com'
response = requests.get(url)
if response.status_code == 200:
    # Print the content of the webpage
    print(response.content)
else:
    print('Failed to retrieve webpage')
```

2. **Beautiful Soup:** The Beautiful Soup library is used for parsing HTML and XML documents. It provides a way to navigate and search the parsed tree, extract data from webpages, and modify the parsed tree. Here's an example of how to use Beautiful Soup to scrape a webpage:

```
from bs4 import BeautifulSoup
import requests
url = 'https://example.com'
response = requests.get(url)
if response.status_code == 200:
    soup = BeautifulSoup(response.content, 'html.parser')
    # Find all the links on the webpage
    links = soup.find_all('a')
    for link in links:
        print(link.get('href'))
else:
    print('Failed to retrieve webpage')
```

3. **Scrapy:** Scrapy is a powerful web crawling and scraping framework that provides a set of tools for extracting data from websites. It allows you to define the structure of the data you want to extract and provides features for handling pagination, following links, and dealing with JavaScript-rendered content. Here's an example of how to use Scrapy to scrape a webpage:

```
import scrapy
class MySpider(scrapy.Spider):
 name = 'myspider'
 start_urls = ['https://example.com']
       def parse(self, response):
              # Extract data from the webpage
              # ...
      # Run the spider
```

4. **Selenium:** Selenium is a web testing tool that can also be used for web scraping. It allows you to automate web browsers and interact with webpages in a way that simulates human behavior, making it useful for scraping dynamic or JavaScript-rendered content. Here's an example of how to use Selenium to scrape a webpage:

```
from selenium import webdriver
url = 'https://example.com'
driver = webdriver.Chrome()
driver.get(url)
# Find elements on the webpage and extract data
# ...
driver.quit()
```

---

24) How do you Parse HTML with BeautifSoup Mpdule? Explain.

To parse HTML with the Beautiful Soup module in Python, you can follow these steps:

Step 1: Install Beautiful Soup

If you haven't already installed Beautiful Soup, you can do so using pip:

```
pip install beautifulsoup4
```

Step 2: Import Beautiful Soup and Requests

Next, you'll need to import the Beautiful Soup and requests modules in your Python script:

Step 3: Make a GET Request

Send a GET request to the webpage you want to scrape using the requests module

Step 4: Create a Beautiful Soup Object

Create a Beautiful Soup object to parse the HTML content of the webpage

Step 5: Extract Data

You can then use the Beautiful Soup object to navigate and search the parsed HTML tree to extract data from the webpage.

Here's the example of how to parse HTML with Beautiful Soup in Python:

```
from bs4 import BeautifulSoup
import requests
url = 'https://example.com'
response = requests.get(url)
```

```
if response.status_code == 200:
    soup = BeautifulSoup(response.content, 'html.parser')
    links = soup.find_all('a')
    for link in links:
        print(link.get('href'))
else:
    print('Failed to retrieve webpage')
```

In this example, we send a GET request to the webpage, create a Beautiful Soup object from the response content, and then find and print all the links on the webpage.

---

25) Explain controlling the browser with the Selenium module.

Selenium is a powerful tool for controlling web browsers through programs and performing actions such as clicking, typing, and navigating through web pages. In Python, the Selenium module can be used to automate browser actions.
Here's an example of how to use the Selenium module in Python to control a web browser:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
# Create a new instance of the Chrome browser
driver = webdriver.Chrome()
# Navigate to a website
driver.get("https://www.example.com")
# Find an element on the page and interact with it
search_box = driver.find_element_by_name("q")
search_box.send_keys("Selenium")
search_box.send_keys(Keys.RETURN)
# Wait for the page to load
driver.implicitly_wait(10)
# Close the browser
driver.quit()
```

In this example, we first import the necessary modules from Selenium. We then create a new instance of the Chrome browser using webdriver.Chrome(). We navigate to a website using driver.get() and then find an element on the page using driver.find_element_by_name() and interact with it by sending keys and pressing return. Finally, we wait for the page to load and close the browser using driver.quit().
Selenium can be used for more complex tasks such as logging in, filling out forms, and scraping data from websites. It is a powerful tool for automating browser actions and can be very useful for web testing and web scraping.

---

26) Explain the process of reading Excel documents with example.

To read Excel documents in Python, we can use the pandas library, which provides a convenient way to work with tabular data. Here's an example of how to read an Excel file using pandas:

```
import pandas as pd
# Load the Excel file into a pandas DataFrame
df = pd.read_excel('example.xlsx')
# Display the contents of the DataFrame
print(df)
```

In this example, we first import the pandas library using import pandas as pd. We then use the read_excel() function to load the contents of the Excel file 'example.xlsx' into a pandas DataFrame. Finally, we print the contents of the DataFrame using print(df).
The read_excel() function has many optional parameters that allow us to customize how the Excel file is read, such as specifying which sheet to read, which columns to use as the index, and how to handle missing values.
Once the Excel file is loaded into a DataFrame, we can perform various operations on the data, such as filtering, sorting, and aggregating, using the powerful capabilities of pandas.
Overall, reading Excel documents in Python using pandas is a straightforward process and provides a flexible and efficient way to work with tabular data from Excel files.

---

27) How do you extract text from pdf and word documents using python.

To extract text from PDF and Word documents in Python, we can use the PyPDF2 library for PDF files and the python-docx library for Word files.

Extracting text from a PDF document using PyPDF2:

```
import PyPDF2
# Open the PDF file
pdf_file = open('example.pdf', 'rb')
# Create a PDF reader object
pdf_reader = PyPDF2.PdfReader(pdf_file)
# Extract text from each page of the PDF
text = ''
for page_num in range(len(pdf_reader.pages)):
   page = pdf_reader.pages[page_num]
   text += page.extract_text()
# Close the PDF file
pdf_file.close()
# Display the extracted text
print(text)
```

Extracting text from a Word document using python-docx:

```
from docx import Document
# Open the Word document
doc = Document('example.docx')
# Extract text from the Word document
text = ''
for paragraph in doc.paragraphs:
    text += paragraph.text + '\n'
# Display the extracted text
print(text)
```

In these examples, we first open the PDF or Word document using the appropriate library. We then use the provided functions to extract text from the document. For PDF files, we loop through each page and extract the text, while for Word files, we loop through each paragraph and extract the text.
Once we have extracted the text from the documents, we can perform further processing or analysis on the text as needed.
Overall, extracting text from PDF and Word documents in Python is straightforward using the PyPDF2 and python-docx libraries, and provides a convenient way to work with textual content from these types of files.

---

28) What are the procedures for Connecting to an SMTP Server?

To connect to an SMTP (Simple Mail Transfer Protocol) server in Python, you can use the built-in smtplib library. Here are the general procedures for connecting to an SMTP server in Python:

1. Import the smtplib library:

```
import smtplib
```

2. Create an SMTP object and establish a connection to the server:

```
    # Replace 'smtp.example.com' with the address of your SMTP server
  smtp_server = smtplib.SMTP('smtp.example.com', 587)  # Use port 587 for TLS encryption or port 465 for SSL encryption
  smtp_server.ehlo()  # Send the extended HELO command to identify ourselves to the server
  smtp_server.starttls()  # Upgrade the connection to use TLS encryption
```

3. Log in to the SMTP server:

```
    # Replace 'your_email@example.com' and 'your_password' with your email credentials
  smtp_server.login('your_email@example.com', 'your_password')
```

4. Send an email:

```
    # Replace the email addresses with the sender and recipient addresses
```

```
from_address = 'your_email@example.com'
to_address = 'recipient_email@example.com'
subject = 'Subject of the email'
body = 'Body of the email'
# Construct the email message
message = f'Subject: {subject}\n\n{body}'
# Send the email
smtp_server.sendmail(from_address, to_address, message)
```

5. Close the connection to the SMTP server:

```
smtp_server.quit()
```

By following these procedures, you can connect to an SMTP server in Python and send emails using the smtplib library. Remember to replace the placeholder values with your actual SMTP server address, email credentials, and email content.

---

29) Explain the process of sending an email using in python

To send an email using Python, you can use the smtplib library to connect to an SMTP server and send the email. Here's a step-by-step explanation of the process:

1. Import the smtplib library: First, you need to import the smtplib library in your Python script using the following import statement:

```
import smtplib
```

2. Create an SMTP object and establish a connection to the server: Next, you need to create an SMTP object and establish a connection to the SMTP server. You will need to specify the server address and port number. For example, to connect to an SMTP server at 'smtp.example.com' using port 587 for TLS encryption, you can use the following code:

```
  smtp_server = smtplib.SMTP('smtp.example.com', 587)
smtp_server.ehlo()
smtp_server.starttls()
```

3**. Log in to the SMTP server:** After establishing a connection, you need to log in to the SMTP server using your email credentials (i.e., your email address and password). This is done using the login method of the SMTP object:

```
smtp_server.login('your_email@example.com', 'your_password')
```

4. Send an email: Once you are logged in, you can construct the email message with the sender and recipient addresses, subject, and body of the email. Then, you can use the sendmail method of the SMTP object to send the email:

```
  from_address = 'your_email@example.com'
to_address = 'recipient_email@example.com'
```

```
 subject = 'Subject of the email'
  body = 'Body of the email'
   message = f'Subject: {subject}\n\n{body}'
   smtp_server.sendmail(from_address, to_address, message)
```

5. Close the connection to the SMTP server: Finally, after sending the email, you should close the connection to the SMTP server using the quit method:

```
smtp_server.quit()
```

By following these steps, you can connect to an SMTP server in Python and send emails using the smtplib library. Remember to replace the placeholder values with your actual SMTP server address, email credentials, and email content.

---

30) How do you manipulate Images with Pillow.

To manipulate images with Pillow in Python, you can use the Pillow library, which is a fork of the Python Imaging Library (PIL). Here's a step-by-step explanation of how to manipulate images using Pillow:

1. Install Pillow: If you haven't already installed Pillow, you can do so using pip:

```
pip install pillow
```

2. Import the Image module from Pillow: In your Python script, you need to import the Image module from Pillow:

```
from PIL import Image
```

3. Open an image file: You can open an image file using the open function of the Image module. For example, to open an image file named "example.jpg":

```
image = Image.open('example.jpg')
```

4. Perform image manipulation operations: Once the image is opened, you can perform various image manipulation operations using the methods and functions provided by the Image module. Some common image manipulation operations include:

```
  - Resizing an image:
       resized_image = image.resize((width, height))
  - Rotating an image:
       rotated_image = image.rotate(angle)
  - Adding a watermark:
       watermark = Image.open('watermark.png')
    image.paste(watermark, (x, y), watermark)
  - Applying filters and effects:
       from PIL import ImageFilter
blurred_image = image.filter(ImageFilter.BLUR)
```

5. Save the manipulated image: After performing the desired image manipulation operations, you can save the manipulated image using the save method of the Image object:

```
manipulated_image.save('manipulated_image.jpg')
```

By following these steps, you can manipulate images using Pillow in Python. Remember to replace the placeholder values with your actual image file names and manipulation parameters.

<u>10 marks:</u>

1. **Write a program Magic 8 ball with list.**

```
import random

responses = ["It is certain",
"Yes, definitely",
"Most likely",
"Outlook good",
  "Yes",
 "Ask again later",
    "Better not tell you now",
    "Cannot predict now",
    "Concentrate and ask again",
    "Don't count on it",
    "My reply is no",
    "My sources say no",
    "Outlook not so good",
    "Very doubtful"
]

def magic_8_ball():
    print("Welcome to the Magic 8 Ball! Ask me a yes/no question.")
    question = input("Your question: ")
    if question:
        print(random.choice(responses))
    else:
        print("Please ask a question.")

magic_8_ball()
```

---

2. **Write a password locker program.**

```
import pyperclip

passwords = {
    "email": "password1",
    "social_media": "password2",
    "banking": "password3"
}

def password_locker():
    print("Welcome to the Password Locker!")
    while True:
        print("Enter 'exit' to quit.")
        service = input("Enter the service name (e.g. email, social_media, banking): ").lower()
        if service == 'exit':
            break
        elif service in passwords:
            pyperclip.copy(passwords[service])
            print(f"Password for {service} has been copied to clipboard.")
        else:
            print("Service not found in the password locker.")

password_locker()
```

---

3. **Write a program to extract phone number and email address using Regex.**

```
import re
def extract_contact_info(text):
    phone_numbers = re.findall(r'\b(?:\d{3}[-.\s]|\(\d{3}\))?\d{3}[-.\s]?\d{4}\b', text)
    email_addresses = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-
z]{2,}\b', text)
    return phone_numbers, email_addresses
text = "Contact us at 123-456-7890 or email us at example@email.com"
phone_numbers, email_addresses = extract_contact_info(text)
print("Phone Numbers:", phone_numbers)
print("Email Addresses:", email_addresses)
```

---

4. **Program to generate random quiz file of guessing state capitals.**

```
import random

capitals = {
    'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
    'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
    # ... and so on for all 50 states
}
states = list(capitals.keys())
random.shuffle(states)
quiz_file = open('state_capitals_quiz.txt', 'w')
for i in range(len(states)):
    correct_answer = capitals[states[i]]
    wrong_answers = list(capitals.values())
    del wrong_answers[wrong_answers.index(correct_answer)]
    wrong_answers = random.sample(wrong_answers, 3)
    answer_options = wrong_answers + [correct_answer]
    random.shuffle(answer_options)

    quiz_file.write(f"{i + 1}. What is the capital of {states[i]}?\n")
    for j in range(4):
        quiz_file.write(f"  {'ABCD'[j]}. {answer_options[j]}\n")
    quiz_file.write('\n')

quiz_file.close()
```

---

5. **Write a program to download all XKCD Comics to a drive**.

```
import requests
import os
from bs4 import BeautifulSoup

# Create a folder to store the comics
folder_name = 'XKCD_Comics'
if not os.path.exists(folder_name):
    os.makedirs(folder_name)

# Get the total number of comics
url = 'https://xkcd.com/archive/'
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')
comic_links = soup.find_all('a')
```

```
    total_comics = len(comic_links)

    # Download each comic
    for i in range(1, total_comics):
        comic_url = f'https://xkcd.com/{i}/info.0.json'
        response = requests.get(comic_url)
        comic_data = response.json()
        img_url = comic_data['img']
        img_response = requests.get(img_url)
        img_name = img_url.split('/')[-1]

        with open(f'{folder_name}/{img_name}', 'wb') as f:
            f.write(img_response.content)
            print(f'Downloaded {img_name}')

    print('All comics downloaded successfully!')
```

---

6. **Write a program to read an image, Colouring and cropping image using Pillow module.**

```
from PIL import Image, ImageOps

# Read the image
image_path = 'example.jpg'
img = Image.open(image_path)

# Colouring the image
coloured_img = ImageOps.colorize(img, "#ff0000", "#0000ff")

# Cropping the image
width, height = img.size
left = width / 4
top = height / 4
right = 3 * width / 4
bottom = 3 * height / 4
cropped_img = img.crop((left, top, right, bottom))

# Save the coloured and cropped images
coloured_img.save('coloured_image.jpg')
cropped_img.save('cropped_image.jpg')

print('Images coloured and cropped successfully!')
```
