THEORY QUESTIONS ASSIGNMENT

Python based theory

To be completed at student's own pace and submitted before given deadline

NO	TASK	POINTS
PYTHON		
1	Theory questions	30
2	String methods	29
3	List methods	11
4	Dictionary methods	11
5	Tuple methods	2
6	Set methods	12
7	File methods	5
	TOTAL	**100**

Grade: 100%

An incredibly sharp piece of admission - every single answer was one of the most in-depth response I had seen to any particular part of Python; this is truly a invaluable document for any programmer given how well it describes stuff & acts as a programming resource. This was a pleasure for me to mark, brilliant work!

1. Python theory questions	**30 points**

1. **What is Python and what are its main features?**

Python is a dynamic, high level, free open source and interpreted programming language. It supports object-oriented programming as well as procedural oriented programming. ✓

Features in Python:

- Easy to code: Python is a high-level programming language. Python is very easy to learn the language as compared to other programming languages.
- Free and Open Source: Python language is freely available at the official website. Open Source means that source code is also available to the public, so you can download it as, use it as well as share it.
- Object-Oriented Language: Python supports object-oriented language and concepts of classes.
- Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory. ✓
- Large Standard Library which provides a rich set of module and functions so you do not have to write your own code for every single thing. Can reuse and share code.
- Dynamically Typed Language: simple, easy to learn syntax.

2. Discuss the difference between Python 2 and Python 3?

Programming languages constantly evolve as developers extend the functionality of the language and iron out quirks that cause problems for developers. Python 3 was introduced in 2008 with the aim of making Python easier to use and change the way it handles strings to match the demands placed on the language today. Programmers who first learned to program in Python 2 sometimes find the new

changes difficult to adjust to, but newcomers often find that the new version of the language makes more sense. ✓

Python 3.0 is fundamentally different to previous Python releases because it is the first Python release that is not compatible with older versions. Programmers usually don't need to worry about minor updates (e.g. from 2.6 to 2.7) as they usually only change the internal workings of Python and don't require programmers to change their syntax. The change between Python 2.7 (the final version of Python 2) and Python 3.0 is much more significant — code that worked in Python 2.7 may need to be written in a different way to work in Python 3.0.

The most problematic difference is Python 2 and Python 3 have different, sometimes incompatible, libraries, which can make an upgrade from 2 to 3 very problematic. The opposite is also true. Since Python 3 is the future, many of today's developers are creating libraries strictly for use with Python 3. Older libraries built for Python 2 are not forwards-compatible. ✓

Other key differences:

1. The two versions have different print statement syntaxes
   In Python 3, the print statement has been replaced with a print () function. For example, in Python 2 it is print "hello" but in Python 3 it is print ("hello"). ✓

2. There is better Unicode support in Python 3
   In Python 3, text strings are Unicode by default. In Python 2, strings are stored as ASCII by default–you have to add a "u" if you want to store strings as Unicode in Python 2.x. ✓

3. Python 3 has improved integer division
   In Python 2, if you write a number without any digits after the decimal point, it rounds your calculation down to the nearest whole number. In Python 3, the expression 5 / 2 will return the expected result of 2.5 without having to worry about adding those extra zeroes. This is an example of how Python 3 syntax can be more intuitive, making it easier for newcomers to learn Python programming. ✓

3. **What is PEP 8?**

PEP 8, sometimes spelled PEP8 or PEP-8, is a document that provides guidelines and best practices on how to write Python code. It was written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan. The primary focus of PEP 8 is to improve the readability and consistency of Python code. ✓

PEP stands for Python Enhancement Proposal, and there are several of them. A PEP is a document that describes new features proposed for Python and documents aspects of Python, like design and style, for the community.

4. **In computing / computer science what is a program?**

A computer program is a collection of instructions that can be executed by a computer to perform a specific task.

A computer program is usually written by a computer programmer in a programming language. From the program in its human-readable form of source code, a compiler or assembler can derive machine code—a form consisting of instructions that the computer can directly execute. Alternatively, a computer program may be executed with the aid of an interpreter. ✓

A collection of computer programs, libraries, and related data are referred to as software. Computer programs may be categorized along functional lines, such as application software and system software. The underlying method used for some calculation or manipulation is known as an algorithm.

5. **In computing / computer science what is a process?**

A process is basically the program in execution. When you start an application in your computer (like a browser or text editor), the operating system creates a process.

In computing, a process is the instance of a computer program that is being executed by one or many threads. It contains the program code and its activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently (all at the same time). ✓

While a computer program is a passive collection of instructions typically stored in a file on disk, a process is the execution of those instructions after being loaded from the disk into memory. Several processes may be associated with the same program; for example, opening up several instances of the same program often results in more than one process being executed.

6. **In computing / computer science what is cache?**

Cache is a small amount of memory which is a part of the CPU - closer to the CPU than RAM. It is used to temporarily hold instructions and data that the CPU is likely to reuse. ✓

The CPU control unit automatically checks cache for instructions before requesting data from RAM. This saves fetching the instructions and data repeatedly from RAM – a relatively slow process which might otherwise keep the CPU waiting. Transfers to and from cache take less time than transfers to and from RAM. The more cache there is, the more data can be stored closer to the CPU.

Cache is graded as Level 1 (L1), Level 2 (L2) and Level 3 (L3):

- L1 is usually part of the CPU chip itself and is both the smallest and the fastest to access. Its size is often restricted to between 8 KB and 64 KB.
- L2 and L3 caches are bigger than L1. They are extra caches built between the CPU and the RAM. Sometimes L2 is built into the CPU with L1. L2 and L3 caches take slightly longer to access than L1. The more L2 and L3 memory available, the faster a computer can run. ✓

Not a lot of physical space is allocated for cache. There is more space for RAM, which is usually larger and less expensive.

7. **In computing / computer science what is a thread and what do we mean by multithreading?**

A **thread** is a unit of exection on concurrent programming. **Multithreading** is a technique which allows a CPU to execute many tasks of one process at the same time. These threads can execute individually while sharing their process resources.

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.

Multithreading is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution on a multiprocessing system. ✓

Multithreading in Python programming is a well-known technique in which multiple threads in a process share their data space with the main thread which makes information sharing and communication within threads easy and efficient. Threads are lighter than processes. Multi threads may execute individually while sharing their process resources. The purpose of multithreading is to run multiple tasks and function cells at the same time. ✓

8. **In computing / computer science what is concurrency and parallelism and whatare the differences?**

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine. ✓

Parallelism is when tasks literally run at the same time, e.g., on a multicore processor. ✓

9. **What is GIL in Python and how does it work?**

The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code. ✓

The GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, so the GIL has gained a reputation as an "infamous" feature of Python.

10. **What do these software development principles mean: DRY, KISS, BDUF**

DRY – Don't Repeat Yourself
DRY refers to code writing methodology. DRY usually refers to code duplication. If we write the same logic more than once, we should "DRY up our code." A common way to DRY up code is to wrap our duplicated logic with a function and replace all places it appears with function calls. ✓

KISS – Keep It Simple Stupid
Keeping it simple is surpsingly hard. Usually when someone tries to over-engineer a solution to a problem. For example an Architect suggests creating a Microservice framework for a simple website. The Engineer will then say: "Let's KISS it and do something simpler". ✓

BDUF – Big Design Up Front
This is a relic from the waterfall era before everyone became cool and Agile. This acronym is here to remind us not get over carried with super complex architecture. We shouldn't spend 3 months designing our application before even writing the first line of code. Start small and iterate. BDUF is basically what happens when you don't KISS and end up with a lot of stuff.

11. **What is a Garbage Collector in Python and how does it work?**

Python garbage collection is the memory management mechanism in python.
Garbage collection is the process of cleaning shared computer memory which is currently being put to use by a running program when that program no longer needs that memory. With Garbage collection, that chunk of memory is cleaned so that other programs (or same program) can use it again. ✓

12. **How is memory managed in Python?**

The process of memory management in Python is straightforward. Python handles its objects by keeping a count to the references each object has in the program, which means, each object stores how many times it is referenced in the program. This count is updated with the program runtime and when it reaches zero, this means it is not reachable from the program anymore. Hence, the memory for

this object can be reclaimed and be freed by the interpreter.

13. **What is a Python module?**

A module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized. Consider a module to be the same as a code library.

14. **What is docstring in Python?**

Python docstrings are the string literals that appear right after the definition of a function, method, class, or module.

Example:

```
def square(n):
   '''Takes in a number n, returns the square of n'''
   return n**2

print(square.__doc__)
```

Inside the triple quotation marks is the docstring of the function square() as it appears right after its definition. They are used to document our code. We can access these docstrings using the __doc__ attribute. We can later use this attribute to retrieve this docstring.

15. **What is pickling and unpickling in Python? Example usage.**

The pickle module is used for implementing binary protocols for serializing and de-serializing a Python object structure.

Pickling: It is a process where a Python object hierarchy is converted into a byte stream.
Unpickling: It is the inverse of Pickling process where a byte stream is converted into an object hierarchy.

Module Interface :
dumps() – This function is called to serialize an object hierarchy.
loads() – This function is called to de-serialize a data stream.

```
# Python program to illustrate
# pickle.loads()
import pickle
import pprint

data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print ('BEFORE:',)
pprint.pprint(data1)

data1_string = pickle.dumps(data1)

data2 = pickle.loads(data1_string)
print ('AFTER:',)
pprint.pprint(data2)

print ('SAME?:', (data1 is data2))
print ('EQUAL?:', (data1 == data2))
```

16. **What are the tools that help to find bugs or perform static analysis?**

If using a 'clever' IDE e.g. PyCharm, we can use in-built debugging functionality to run a program in debugging mode.

Pychecker and Pylint are the static analysis tools that help to find bugs in python.

Pychecker is an opensource tool for static analysis that detects the bugs from source code and warns about the style and complexity of the bug.

Pylint is highly configurable and it acts like special programs to control warnings and errors, it is an extensive configuration file Pylint is also an opensource tool for static code analysis it looks for programming errors and is used for coding standard. it checks the length of each programming line. it checks the variable names according to the project style. it can also be used as a standalone program, it also integrates with python IDEs such as Pycharm, Spyder, Eclipse, and Jupyter

17. **How are arguments passed in Python by value or by reference? Give an example.**

Python uses a mechanism, which is known as "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing"

If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like Call by-value. It's different, if we pass mutable arguments.

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Example:

```
student={'Archana':28,'krishna':25,'Ramesh':32,'vineeth':25}

def test(student):

    new={'alok':30,'Nevadan':28}

    student.update(new)

    print("Inside the function:",student)

    return

test(student)

print("Outside the function:",student)
```

Output:

```
Inside the function: {'Archana': 28, 'krishna': 25, 'Ramesh': 32,
'vineeth': 25, 'alok': 30, 'Nevadan': 28}
Outside the function: {'Archana': 28, 'krishna': 25, 'Ramesh': 32,
'vineeth': 25, 'alok': 30, 'Nevadan': 28}
```

18. **What are Dictionary and List comprehensions in Python? Provide examples.**

List comprehensions provide a more compact and elegant way to create lists than for-loops, and also allow you to create lists from existing lists. List comprehensions are constructed from brackets containing an expression, which is followed by a for clause, that is [item-expression for item in iterator] or [x for x in iterator], and can then be followed by further for or if clauses: [item-expression for item in iterator if conditional].

How list comprehensions work:

x = [i for i in range(10)]

Produces the result:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

And utilizing a conditional statement:
x = [i for i in range(10) if i > 5]

Returns the list:
[6, 7, 8, 9]
because the condition i > 5 is checked.

As well as being more concise and readable than their for-loop equivalents, list comprehensions are also notably faster. ✓

Dictionary comprehensions are very similar to list comprehensions in Python – only for dictionaries. They provide an elegant method of creating a dictionary from an iterable or transforming one dictionary into another. The syntax is similar to that used for list comprehension, namely {key: item-expression for item in iterator}, but note the inclusion of the expression pair (key:value).

Simple example to make a dictionary:

The code can be written as:
dict([(i, i+10) for i in range(4)])

which is equivalent to:
{i : i+10 for i in range(4)}

In both cases, the return value is:
{0: 10, 1: 11, 2: 12, 3: 13}

This basic syntax can also be followed by additional for or if clauses: {key: item-expression for item in iterator if conditional}.

Using an if statement allows you to filter out values to create your new dictionary.

For example:
{i : i+10 for i in range(10) if i > 5} ✓

Returns:
{6: 16, 7: 17, 8: 18, 9: 19}

Similar to list comprehensions, dictionary comprehensions are also a powerful alternative to for-loops and lambda functions. For-loops, and nested for-loops in particular, can become complicated and confusing. Dictionary comprehensions offer a more compact way of writing the same code, making it easier to read and understand.

19. **What is namespace in Python?**

A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary. For example, in a directory-file system structure in computers, one can have multiple directories having a file with the same name inside every directory. But one can get directed to the file one wishes, just by specifying the absolute path to the file.

The role of a namespace is like a surname. One might not find a single "Alice" in the class there might be multiple "Alice" but when you particularly ask for "Alice Lee" or "Alice Clark" (with a surname), there will be only one (time being don't think of both first name and surname are same for multiple students).

On similar lines, the Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace. So, the division of the word itself gives a little more information. Its Name (which means name, a unique identifier) + Space (which talks something related to scope). Here, a name might be of any Python method or variable and space depends upon the location from where is trying to access a variable or a method.

20. **What is pass in Python?**

The pass statement in Python is used when a statement is required syntactically, but you do not want any command or code to execute.

The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet.

21. **What is unit test in Python?**

Unit testing is an important part of "Test-Driven Development (TDD)". The unit test is one of the testing methods by which an individual unit of the program, such as a function, class or module, is put under various test cases to determine whether they work as expected. It helps us write robust code.

22. **In Python what is slicing?**

Slicing is the extraction of a part of a string, list, or tuple. It enables users to access the specific range of elements by mentioning their indices.

Syntax: Object [start:stop:step]

- "Start" specifies the starting index of a slice
- "Stop" specifies the ending element of a slice
- You can use one of these if you want to skip certain items

Other notes:

```
a[start:stop]       # items start through stop-1
a[start:]           # items start through the rest of the array
a[:stop]            # items from the beginning through stop-1
a[:]                # a copy of the whole array
a[start:stop:step]  # start through not past stop, by step
```

The other feature is that start or stop may be a negative number, which means it counts from the end of the array instead of the beginning. So:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

Similarly, step may be a negative number:

```
a[::-1]    # all items in the array, reversed
a[1::-1]   # the first two items, reversed
a[:-3:-1]  # the last two items, reversed
a[-3::-1]  # everything except the last two items, reversed
```

Python is kind to the programmer if there are fewer items than you ask for. For example, if you ask for a[:-2] and a only contains one element, you get an empty list instead of an error. Sometimes you would prefer the error, so you have to be aware that this may happen.

23. **What is a negative index in Python?**

It is just reverse. When you use the negative index, the program will fetch the item from the end.

Python supports negative indexing of arrays, something which is not available in arrays in most other programming languages. This means that the index value of -1 gives the last element, and -2 gives the second last element of an array. The negative indexing starts from where the array ends.

24. **How can the ternary operators be used in python? Give an example.**

Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false. It was added to Python in version 2.5. It simply allows to test a condition in a single line replacing the multiline if-else making the code compact.

Syntax :
[on_true] if [expression] else [on_false]

Simple Method to use ternary operator:

```
# Program to demonstrate conditional operator
a, b = 10, 20
# Copy value of a in min if a < b else copy b
min = a if a < b else b
print(min)
```

Output: 10

25. **What does this mean: *args, **kwargs? And why would we use it?**

*args and **kwargs allow you to pass multiple arguments or keyword arguments to a function.

Using the Python *args Variable in Function Definitions:
There are a few ways you can pass a varying number of arguments to a function. The first way is often the most intuitive for people that have experience with collections. You simply pass a list or a set of all the arguments to your function. So for my_sum(), you could pass a list of all the integers you need to add. This would works, but whenever you call this function you'll also need to create a list of arguments to pass to it. This can be inconvenient, especially if you don't know up front all the values that should go into the list.

This is where *args can be really useful, because it allows you to pass a varying number of positional

arguments. Take the following example:

```
# sum_integers_args.py
def my_sum(*args):
    result = 0
    # Iterating over the Python args tuple
    for x in args:
        result += x
    return result

print(my_sum(1, 2, 3))
```

In this example, you're no longer passing a list to my_sum(). Instead, you're passing three different positional arguments. my_sum() takes all the parameters that are provided in the input and packs them all into a single iterable object named args.

Note that args is just a name. You're not required to use the name args. You can choose any name that you prefer, such as integers. The function still works, even if you pass the iterable object as integers instead of args. All that matters here is that you use the unpacking operator (*).

Using the Python kwargs Variable in Function Definitions

kwargs works just like *args, but instead of accepting positional arguments it accepts keyword (or **named) arguments. Take the following example:

```
# concatenate.py
def concatenate(**kwargs):
    result = ""
    # Iterating over the Python kwargs dictionary
    for arg in kwargs.values():
        result += arg
    return result

print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

When you execute the script above, concatenate() will iterate through the Python kwargs dictionary and concatenate all the values it finds:

```
$ python concatenate.py
RealPythonIsGreat!
```

Like args, kwargs is just a name that can be changed to whatever you want. Again, what is important here is the use of the **unpacking operator** (**).

So, the previous example could be written like this:

```python
# concatenate_2.py
def concatenate(**words):
    result = ""
    for arg in words.values():
        result += arg
    return result

print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

Note that in the example above the iterable object is a standard dict. If you iterate over the dictionary and want to return its values, like in the example shown, then you must use .values().

In fact, if you forget to use this method, you will find yourself iterating through the keys of your Python kwargs dictionary instead, like in the following example:

```python
# concatenate_keys.py
def concatenate(**kwargs):
    result = ""
    # Iterating over the keys of the Python kwargs dictionary
    for arg in kwargs:
        result += arg
    return result

print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

26. How are range and xrange different from one another?

In Python 2.x:

- range creates a list, so if you do range(1, 10000000) it creates a list in memory with 9999999 elements.
- xrange is a sequence object that evaluates lazily.

In Python 3:

- range does the equivalent of Python 2's xrange. To get the list, you have to explicitly use list(range(...)).
- xrange no longer exists.

27. What is Flask and what can we use it for?

Flask is a web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.

Flask offers suggestions, but doesn't enforce any dependencies or project layout. It is up to the developer to choose the tools and libraries they want to use. There are many extensions provided by the community that make adding new functionality easy.

```
# save this as app.py
from flask import Flask, escape, request

app = Flask(__name__)

@app.route('/')
def hello():
    name = request.args.get("name", "World")
    return f'Hello, {escape(name)}!'
```

```
$ flask run
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

28. **What are clustered and non-clustered index in a relational database?**

With a clustered index the rows are stored physically on the disk in the same order as the index. Therefore, there can be only one clustered index.

With a non-clustered index there is a second list that has pointers to the physical rows. You can have many non-clustered indices, although each new index will increase the time it takes to write new records.

It is generally faster to read from a clustered index if you want to get back all the columns. You do not have to go first to the index and then to the table. Writing to a table with a clustered index can be slower, if there is a need to rearrange the data.

29. **What is a 'deadlock' a relational database?**

In a database, a deadlock is a situation in which two or more transactions are waiting for one another to give up locks. For example, Transaction A might hold a lock on some rows in the Accounts table and needs to update some rows in the Orders table to finish.

30. **What is a 'livelock' a relational database?**

A livelock is one where a request for exclusive lock is denied continuously because a series of overlapping shared locks keep on interfering each other and to adapt from each other they keep on changing the status which further prevents them to complete the task.

2. Python string methods: describe each method and provide an example	**29 points**

METHOD	DESCRIPTION	EXAMPLE

✓ **capitalize()**	Returns a copy of the original string and changes the first character of the string into a capital letter (whilst the rest remain lowercase)	txt = “python is awesome” print(txt.capitalize()) Output: Python is awesome
✓ **casefold()**	Returns a string where all the characters are lower case	txt = "Hello And Welcome" x = txt.casefold() print(x) Output: hello and welcome
✓ **center()**	Center() method will center align string, using the specified number to centre it to (or speace can be default), and also the specified character as the fill character if you don’t want empty space.	txt = "Nosheen" x = txt.center(20, "O") print(x) Output: OOOOOONosheenOOOOOOO
✓ **count()**	Returns the number of elements with the specified value. Can be used with string, number, list, tuple, etc.	fruits = ['cherry', 'apple', 'banana', 'cherry'] x = fruits.count("cherry") print(x) Output: 2
✓ **endswith()**	Returns True if the string ends with the specified value, otherwise False	txt = "Python is awesome." x = txt.endswith("some.") print(x) Output: True
✓ **find()**	Find() method finds where the first occurrence of the specified value is along the string. If the value is not found, find() method returns -1.	txt = "Hello and welcome" x = txt.find(“welcome”) print(x) Output: 10
✓ **format()**	Format() method formats the specified values and inserts them inside the strings placeholder, which is defined with curly brackets {}. Returns the formatted string.	txt = "My name is {}, I'm {} years old".format("Nosheen", 38) print(txt) Output: My name is Nosheen, I'm 38 years old
✓ **index()**	Returns the position at the first occurrence of the specified value	fruits = ['apple', 'banana', 'cherry'] x = fruits.index("cherry") print(x) Output: 2
✓ **isalnum()**	Returns True if all the characters are alphanumeric – meaning they contain alphabetic letters and numbers only	txt = "Company12" x = txt.isalnum() print(x) Output: True

isalpha()	returns True if all the characters are alphabet letters (a-z)	txt = "python is awesome" x = txt.isalpha() print(x) Output: True
isdigit()	The isdigit() method returns True if all the characters are digits, otherwise False.	txt = "50800" x = txt.isdigit() print(x) Output: True
islower()	The islower() method returns True if all the characters are in lower case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.	txt = "hello world!" x = txt.islower() print(x) Output: True
isnumeric()	The isnumeric() method returns True if all the characters are numeric (0-9), otherwise False. Exponents, like ² and ¾ are also considered to be numeric values. "-1" and "1.5" are NOT considered numeric values, because *all* the characters in the string must be numeric, and the - and the . are not.	txt = "565543" x = txt.isnumeric() print(x) Output: True
isspace()	The isspace() method returns True if all the characters in a string are whitespaces, otherwise False.	txt = " " x = txt.isspace() print(x) Output: True
istitle()	The istitle() method returns True if all words in a text start with a upper case letter, AND the rest of the word are lower case letters, otherwise False. Symbols and numbers are ignored.	txt = "Hello, And Welcome To My World!" x = txt.istitle() print(x) Output: True
isupper()	The isupper() method returns True if all the characters are in upper case, otherwise False. Numbers, symbols and spaces are not checked,	txt = "THIS IS NOW!" x = txt.isupper() print(x)

	only alphabet characters.	Output: True
join()	The join() method takes all items in an iterable and joins them into one string. A string must be specified as the separator.	myTuple = ("John", "Peter", "Vicky") x = "#".join(myTuple) print(x) Output: John#Peter#Vicky
lower()	The lower() method returns a string where all characters are lower case. Symbols and Numbers are ignored.	txt = "Hello my FRIENDS" x = txt.lower() print(x) Output: hello my friends
lstrip()	The lstrip() method removes any leading characters (space is the default leading character to remove)	txt = " banana " x = txt.lstrip() print("of all fruits", x, "is my favorite") Output: of all fruits banana is my favorite
replace()	The replace() method replaces a specified phrase with another specified phrase. Note: All occurrences of the specified phrase will be replaced, if nothing else is specified.	txt = "I like bananas" x = txt.replace("bananas", "apples") print(x) Output: I like apples
rsplit()	The rsplit() method splits a string into a list, starting from the right. If no "max" is specified, this method will return the same as the split() method. Note: When maxsplit is specified, the list will contain the specified number of elements plus one.	txt = "apple, banana, cherry" x = txt.rsplit(", ") print(x) Output: ['apple', 'banana', 'cherry']
rstrip()	The rstrip() method removes any trailing characters (characters at the end a string), space is the default trailing character to remove. altho already implied by the example, what is the difference between rstrip and lstrip?	txt = " banana " x = txt.rstrip() print("of all fruits", x, "is my favorite") Output: of all fruits banana is my favorite

split()	The split() method splits a string into a list. You can specify the separator, default separator is any whitespace. Note: When maxsplit is specified, the list will contain the specified number of elements plus one.	txt = "welcome to the jungle" x = txt.split() print(x) Output: ['welcome', 'to', 'the', 'jungle']
splitlines()	The splitlines() method splits a string into a list. The splitting is done at line breaks.	txt = "Thank you for the music\nWelcome to the jungle" x = txt.splitlines() print(x) Output: ['Thank you for the music', 'Welcome to the jungle']
startswith()	The startswith() method returns True if the string starts with the specified value, otherwise False.	txt = "Hello, welcome to my world." x = txt.startswith("Hello") print(x) Output: True
strip()	The strip() method removes any leading (spaces at the beginning) and trailing (spaces at the end) characters (space is the default leading character to remove)	txt = " banana " x = txt.strip() print("of all fruits", x, "is my favorite") Output: of all fruits banana is my favorite
swapcase()	The swapcase() method returns a string where all the upper case letters are lower case and vice versa.	txt = "Hello My Name Is PETER" x = txt.swapcase() print(x) Output: hELLO mY nAME iS peter

title()	The title() method returns a string where the first character in every word is upper case. Like a header, or a title. If the word contains a number or a symbol, the first letter after that will be converted to upper case.	txt = "Welcome to my world" x = txt.title() print(x) Output: Welcome To My World
upper()	The upper() method returns a string where all characters are in upper case. Symbols and Numbers are ignored.	txt = "Hello my friends" x = txt.upper() print(x) Output: HELLO MY FRIENDS

3. Python list methods: describe each method and provide an example	**11 points**

Method	Description	Example
append()	The append() method appends an element to the end of the list.	fruits = ['apple', 'banana', 'cherry'] fruits.append("orange") print(fruits) Output: ['apple', 'banana', 'cherry', 'orange']
clear()	The clear() method removes all the elements from a list.	fruits = ['apple', 'banana', 'cherry', 'orange'] fruits.clear() print(fruits) Output: []
copy()	The copy() method returns a copy of the specified list.	fruits = ['apple', 'banana', 'cherry', 'orange'] x = fruits.copy() print(x) example output is mi--ssing 'orange'? Output: ['apple', 'banana', 'cherry']
count()	The count() method returns the number of elements with the specified value.	fruits = ["apple", "banana", "cherry"] x = fruits.count("cherry") print(x) Output: 1

✓ **extend()**	The extend() method adds the specified list elements (or any iterable((list, set, tuple, etc.)) to the end of the current list.	fruits = ['apple', 'banana', 'cherry'] cars = ['Ford', 'BMW', 'Volvo'] fruits.extend(cars) print(fruits) Output: ['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']
✓ **index()**	The index() method returns the position at the first occurrence of the specified value.	fruits = ['apple', 'banana', 'cherry'] x = fruits.index("cherry") print(x) Output:2
✓ **insert()**	The insert() method inserts the specified value at the specified position.	fruits = ['apple', 'banana', 'cherry'] fruits.insert(1, "orange") print(fruits) Output: ['apple', 'orange', 'banana', 'cherry']
✓ **pop()**	The pop() method removes the element at the specified position. If you do not specify a position, it will remove the last item from the list.	fruits = ['apple', 'banana', 'cherry'] fruits.pop(1) print(fruits) Output: ['apple', 'cherry']
✓ **remove()**	The remove() method removes the first occurrence of the element with the specified value.	fruits = ['apple', 'banana', 'cherry'] fruits.remove("banana") print(fruits) Output: ['apple', 'cherry']
✓ **reverse()**	The reverse() method reverses the sorting order of the elements.	fruits = ['apple', 'banana', 'cherry'] fruits.reverse() print(fruits) Output: ['cherry', 'banana', 'apple']
✓ **sort()**	The sort() method sorts the list ascending by default. You can also make a function to decide the sorting criteria(s).	cars = ['Ford', 'BMW', 'Volvo'] cars.sort() print(cars) Output: ['BMW', 'Ford', 'Volvo']

4. Python tuple methods: describe each method and provide an example	2 points

Method	Description	Example
count()	The count() method returns the number of times a specified value appears in the tuple.	thistuple = (1, 3, 7, 7, 5, 4, 6, 8, 5) x = thistuple.count(5) print(x) Output: 2
index()	The index() method finds the first occurrence of the specified value. The index() method raises an exception if the value is not found.	thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5) x = thistuple.index(8) print(x) Output: 3

5. Python dictionary methods: describe each method and provide an example	11 points

Method	Description	Example

clear()	The clear() method removes all the elements from a dictionary.	car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		car.clear()
		
		print(car)
		
		Output: {}

copy()	The copy() method returns a copy of the specified dictionary.	car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}

		x = car.copy() print(x) Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
✓ **fromkeys()**	The fromkeys() method returns a dictionary with the specified keys and the specified value.	x = ('key1', 'key2', 'key3') y = 0 thisdict = dict.fromkeys(x, y) print(thisdict) Output: ['key1': 0, 'key2': 0, 'key3': 0]

✓ **get()**	The get() method returns the value of the item with the specified key.	car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		x = car.get("model")
		
		print(x)
		
		Output: Mustang

✓ **items()**	The items() method returns a view object. The view object contains the key-value pairs of the dictionary, as tuples in a list.
	
	The view object will reflect any changes done to the dictionary, see example below.
		car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		x = car.items()
		
		print(x)
		
		Output: dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

✓ **keys()**	The keys() method returns a view object. The view object contains the keys of the dictionary, as a list.
	
	The view object will reflect any changes done to the dictionary.
		car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		x = car.keys()
		
		print(x)
		
		Output: dict_keys(['brand', 'model', 'year'])_

pop()	The pop() method removes the specified item from the dictionary.
	
	The value of the removed item is the return value of the pop() method.
		car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		car.pop("model")
		
		print(car)
		
		Output: {'brand': 'Ford', 'year': 1964}

popitem()	The popitem() method removes the item that was last inserted into the dictionary. In versions before 3.7, the popitem() method removes a random item.
	
	The removed item is the return value of the popitem() method, as a tuple.
		car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		car.popitem()
		
		print(car)
		
		Output: {'brand': 'Ford', 'model': 'Mustang'}

setdefault()	The setdefault() method returns the value of the item with the specified key.
	
	If the key does not exist, insert the key, with the specified value.
		Get the value of the "model" item:
		
		car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		x = car.setdefault("model", "Bronco")
		
		print(x)
		
		Output: Mustang
		
		Get the value of the "color" item, if the "color" item does not exist, insert "color" with the value "white":
		
		car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		x = car.setdefault("color", "white")
		
		print(x)
		
		Output: White

update()	The update() method inserts the specified items to the dictionary.
	
	The specified items can be a dictionary, or an iterable object with key value pairs.
		Insert an item to the dictionary:
		
		car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}
		
		car.update({"color": "White"})
		
		print(car)
		
		Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}

values()	The values() method returns a view object. The view object contains the values of the dictionary, as a list.
	
	The view object will reflect any changes done to the
		car = {
		"brand": "Ford",
		"model": "Mustang",
		"year": 1964
		}

	dictionary, see example below.	x = car.values() print(x) Output: dict_values(['Ford', 'Mustang', 1964])

6. Python set methods: describe each method and provide an example	**12 points**

Method	Description	Example
add()	The add() method adds an element to the set. If the element already exists, the add() method does not add the element.	fruits = {"apple", "banana", "cherry"} fruits.add("orange") print(fruits) Output: {'banana', 'apple', 'orange', 'cherry'}
clear()	The clear() method removes all elements in a set.	fruits = {"apple", "banana", "cherry"} fruits.clear() print(fruits) Output: set()
copy()	The copy() method copies the set.	fruits = {"apple", "banana", "cherry"} x = fruits.copy() print(x) Output: {'banana', 'cherry', 'apple'}
difference()	The difference() method returns a set that contains the difference between two sets. Meaning: The returned set contains items that exist only in the first set, and not in both sets.	x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} z = x.difference(y) print(z) Output: {'cherry', 'banana'}
intersection()	The intersection() method returns a set that	x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} z = x.intersection(y)

	contains the similarity between two or more sets. Meaning: The returned set contains only items that exist in both sets, or in all sets if the comparison is done with more than two sets.	print(z) Output: {'apple'}
✓ **issubset()**	The issubset() method returns True if all items in the set exists in the specified set, otherwise it returns False.	Return True if all items in set x are present in set y: x = {"a", "b", "c"} y = {"f", "e", "d", "c", "b", "a"} z = x.issubset(y) print(z) Output: True
✓ **issuperset()**	The issuperset() method returns True if all items in the specified set exists in the original set, otherwise it retuns False.	Return True if all items set y are present in set x: x = {"f", "e", "d", "c", "b", "a"} y = {"a", "b", "c"} z = x.issuperset(y) print(z) Output: True
✓ **pop()**	The pop() method removes a random item from the set. This method returns the removed item.	Remove a random item from the set: fruits = {"apple", "banana", "cherry"} fruits.pop() print(fruits) Output: {'cherry', 'apple'}
✓ **remove()**	The remove() method removes the specified element from the set. This method is different from the discard() method, because the remove() method *will raise an error* if the specified item does not exist, and the discard() method *will not*. ✓	Remove "banana" from the set: fruits = {"apple", "banana", "cherry"} fruits.remove("banana") print(fruits) Output: {'apple', 'cherry'}

symmetric_difference()	The symmetric_difference() method returns a set that contains all items from both set, but not the items that are present in both sets. Meaning: The returned set contains a mix of items that are not present in both sets.	Return a set that contains all items from both sets, except items that are present in both sets: x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} z = x.symmetric_difference(y) print(z) Output: {'microsoft', 'banana', 'cherry', 'google'}
union()	The union() method returns a set that contains all items from the original set, and all items from the specified set(s). You can specify as many sets you want, separated by commas. It does not have to be a set, it can be any iterable object. If an item is present in more than one set, the result will contain only one appearance of this item.	Return a set that contains all items from both sets, duplicates are excluded: x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} z = x.union(y) print(z) Output: {'microsoft', 'google', 'banana', 'apple', 'cherry'}
update()	The update() method updates the current set, by adding items from another set (or any other iterable). If an item is present in both sets, only one appearance of this item will be present in the updated set.	Insert the items from set y into set x: x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} x.update(y) print(x) Output: {'apple', 'cherry', 'google', 'microsoft', 'banana'}

7. Python file methods: describe each method and provide an example	**5 points**

Method	Description	Example

read()	The read() method returns the specified number of bytes from the file. Default is -1 which means the whole file.	Read the content of the file "demofile.txt": f = open("demofile.txt", "r") print(f.read()) Output: C:\Users\My Name>python demo_file_read.py Hello! Welcome to demofile.txt This file is for testing purposes. Good Luck!
readline()	The readline() method returns one line from the file. You can also specified how many bytes from the line to return, by using the size parameter.	Read the first line of the file "demofile.txt": f = open("demofile.txt", "r") print(f.readline()) Output: C:\Users\My Name>python demo_file_read.py Hello! Welcome to demofile.txt
readlines()	The readlines() method returns a list containing each line in the file as a list item. Use the hint parameter to limit the number of lines returned. If the total number of bytes returned exceeds the specified number, no more lines are returned.	Return all lines in the file, as a list where each line is an item in the list object: f = open("demofile.txt", "r") print(f.readlines()) Output: C:\Users\My Name>python demo_file_readlines.py ['Hello! Welcome to demofile.txt\n', 'This file is for testing purposes.\n', 'Good Luck!']

write()	The write() method writes a specified text to the file.
	
	Where the specified text will be inserted depends on the file mode and stream position.
	
	"a": The text will be inserted at the current file stream position, default at the end of the file.
	
	"w": The file will be emptied before the text will be inserted at the current file stream position, default 0.
		Open the file with "a" for appending, then add some text to the file:
		
		f = open("demofile2.txt", "a")
		f.write("See you soon!")
		f.close()
		
		#open and read the file after the appending:
		f = open("demofile2.txt", "r")
		print(f.read())
		
		Output:
		C:\Users\My Name>python demo_file_readlines.py
		['Hello! Welcome to demofile.txt\n', 'This file is for testing purposes.\n', 'Good Luck!']

writelines()
 	
✓
	The writelines() method writes the items of a list to the file.
	
	Where the texts will be inserted depends on the file mode and stream position.
	
	"a": The texts will be inserted at the current file stream position, default at the end of the file.
	
	"w": The file will be emptied before the texts will be inserted at the current file stream position, default 0.
		Open the file with "a" for appending, then add a list of texts to append to the file:
		
		f = open("demofile3.txt", "a")
		f.writelines(["See you soon!", "Over and out."])
		f.close()
		
		#open and read the file after the appending:
		f = open("demofile3.txt", "r")
		print(f.read())
		
		Output: C:\Users\My Name>python demo_file_writelines.py
		Hello! Welcome to demofile2.txt
		This file is for testing purposes.
		Good Luck!See you soon!Over and out.