VTU Question Paper Solution & Scheme
Introduction to Python Programming
Session June/July 2024

Q 1 a Explain with example print( ), input() and len( ) [6 Marks]
Explanation & Example - [3 Marks]
Explanation & Code-[3 Marks]
The print() Function
The print() function displays the string value inside the parentheses on the screen.
Example:
print("Hello World")
The line print('Hello world!') means "Print out the text in the string 'Hello world!'." When Python executes this line, you say that Python is calling the print() function and the string value is being passed to the func- tion.

The input() Function
The input() function waits for the user to type some text on the keyboard and press enter.
Example:
myName=input()
This function call evaluates to a string equal to the user's text, and the previous line of code assigns the myName variable to this string value.
The len() Function
You can pass the len() function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.
Example:
>>> len('hello')
5

Q 1 b Explain elif, for, while statement in python with example.[6 Marks]
Explanation - [5 Marks]
Explanation & Code-[3 Marks]
elif Statements:
The elif statement is an "else if" statement that always follows an if or another elif statement. It provides another condition that is checked only if any of the previous con- ditions were False.
In code, an elif statement always consists of the following:
• The elif keyword
• A condition (that is, an expression that evaluates to True or False)

• A colon Starting on the next line, an indented block of code (called the elif clause)
Let's add an elif to the name checker to see this statement in action.

Example:

```
if name == 'Alice':
print('Hi, Alice.')
elif age < 12:
print('You are not Alice, kiddo.')
```

for Loops:

The for keyword
• A variable name
• The in keyword
• A call to the range() method with up to three integers passed to it
• A colon Starting on the next line, an indented block of code (called the for clause)

Example:

```
print('My name is')
for i in range(5):
print('Jimmy Five Times (' + str(i) + ')')
```

while Loop Statements:

The code in a while clause will be executed as long as the while state- ment's condition is True. In code, a while statement always consists of the following:
• The while keyword
• A condition (that is, an expression that evaluates to True or False)
• A colon Starting on the next line, an indented block of code (called the while clause)

Examples of while

```
# Prints "Hello, world." to the console five times:
spam = 0
while spam < 5:
print('Hello, world.')
spam = spam + 1
```

Q 1 c Develop a program to genratë Fibonacci sequence of Length(N). Read N from the console. [8 Marks]

Full Program- [5 Marks]

Logic-[3 Marks]

Program:

```
n=int(input("enter the number"))
a=0
b=1
sum=0
i=0
print("fibonacci series")
while(i<=n):
```

```
    print(sum)
    a=b
    b=sum
    sum=a+b
    i=i+1
```

Output:

N=5

Fibonacci sequence =0 1 1 2 3

Q 2 a What are functions? Explain python function with parameters and return statement. [6 Marks]

A function is like a mini-program within a program.

def Statements with Parameters:

def hello(name):

print('Hello ' + name)

hello('Alice')

hello('Bob')

Output:

Hello Alice

Hello Bob

Return Values and return Statements:

When you call the len() function and pass it an argument such as 'Hello', the function call evaluates to the integer value 5, which is the length of the string you passed it. In general, the value that a function call evaluates to is called the return value of the function.

When creating a function using the def statement, you can specify what the return value should be with a return statement. A return statement con- sists of the following:

• The return keyword

• The value or expression that the function should return

Q 2b How to handle exception in python with example [6 Marks]

getting an error, or exception, in your Python program means the entire program will crash. You don't want this to happen in real-world pro- grams. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a "divide-by-zero" error. Open a new file editor window and enter the following code, saving it as zeroDivide.py:

def spam(divideBy):

return 42 / divideBy

print(spam(2))

print(spam(12))

print(spam(0))

print(spam(1))

We've defined a function called spam, given it a parameter, and then printed the value of that function with various parameters to see what hap- pens. This is the output you get when you run the previous code:
21.0
3.5
A ZeroDivisionError happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the return statement in spam() is causing an error.
Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause. The program execu- tion moves to the start of a following except clause if an error happens.
You can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
try:
return 42 / divideBy
except ZeroDivisionError:
print('Error: Invalid argument.')
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Q 2 c Explain Local and Global scope with variables for each [8 Marks]
Parameters and variables that are assigned in a called function are said
to exist in that function's local scope. Variables that are assigned outside all functions are said to exist in the global scope. A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable. A variable must be one or the other; it cannot be both local and global.
A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call this function, the local variables will not remember the values stored in them from the last time the function was called.
Scopes matter for several reasons:
• Code in the global scope cannot use any local variables.
• However, a local scope can access global variables.
• Code in a function's local scope cannot use variables in any other local scope.
• You can use the same name for different variables if they are in dif- ferent scopes. That

is, there can be a local variable named spam and a global variable also named spam.
Global Variables Can Be Read from a Local Scope
Consider the following program:

```
def spam():
print(eggs)
eggs = 42
spam()
print(eggs)
```

Local Variables Cannot Be Used in the Global Scope
Consider this program, which will cause an error when you run it:

```
def spam():
eggs = 31337
spam()
print(eggs)
```

If you run this program, the output will look like this:

```
Traceback (most recent call last):
File "C:/test3784.py", line 4, in <module>
print(eggs)
NameError: name 'eggs' is not defined
```

Q 3 a Explain the use of in and not in operator in list with examples. [6 Marks]
Explanation - [3 Marks]
Example - [3 Marks]

The in and not in Operators

- We can determine whether a value is or isn't in a list with the in and not in operators.
- in and not in are used in expressions and connect two values: a value to look for in a list and the list where it may be found and these expressions will evaluate to a Boolean value.

Example:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
```

False

>>> 'howdy' not in spam

False

>>> 'cat' not in spam

True

- The following program lets the user type in a pet name and then checks to see whether the name is in a list of pets

```
myPets = [‘Zophie’, ‘Pooka’, ‘Fat-tail’]
print(‘Enter a pet name:’)
name = input()
if name not in myPets:
        print(‘ I do not have a pet named’ +name)
else:
        print(name + ‘is my pet’)
```

Output:
Enter a pet name:
Muffin
I do not have a pet named Muffin

Q 3 b Explain negative Indexing, slicing, index( ), åppend( ), remove( ). pop( ), insert( ) and sort() with suitable examples. [8 Marks]
Explanation - [4 Marks]
Example - [4 Marks]

<u>Negative Indexing</u>
•Indexes start at 0

•Negative index:

•-1 refers to the last index in a list

•-2 refers to the second-to-last index in a list

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
```

<u>Slicing:</u>

•Slice can get several values from a list, in the form of a new list.

•Has two integers separated by a colon.

•First integer is the index where the slice starts.

•Second integer is the index where the slice ends.

•A slice evaluates to a new list value.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

index():

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
    >>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
    >>> spam.index('Pooka')
1
```

append():

append() method call adds the argument to the end of the list.

```
>>> spam = ['cat', 'dog', 'bat']
```

```
>>> spam.append('moose')

>>> spam
['cat', 'dog', 'bat', 'moose']
```

remove():

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam.remove('bat')

>>> spam
['cat', 'rat', 'elephant']
```

- Attempting to delete a value that does not exist in the list will result in a ValueError error.
- If the value appears multiple times in the list, only the first instance of the value will be removed.

pop():

The pop() method removes the element at the specified position.

example:

```
fruits = ['apple', 'banana', 'cherry']

fruits.pop(1)
```

insert():

The insert() method can insert a value at any index in the list.

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.insert(1, 'chicken')

>>> spam
['cat', 'chicken', 'dog', 'bat']
```

sort():

```
>>> spam = [2, 5, 3.14, 1, -7]

>>> spam.sort()

>>> spam

[-7, 1, 2, 3.14, 5]

>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']

>>> spam.sort()

>>> spam

['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

Q 3 c Write about Mutable and Immutable data type in list [6 Marks]
Explanation - [4 Marks]
Example - [2 Marks]

- A mutable object is an entity whose value can be altered post its creation. Imagine it as a whiteboard: you can write on it, erase your inscriptions, and jot something new. This characteristic epitomizes mutable Python variables.
- Conversely, an immutable object is an entity whose value remains constant post its creation. Visualize it as a printed book: once printed, you cannot simply erase a paragraph and overwrite it. The words are permanent and unalterable.
- Python, being a dynamically-typed language, offers a variety of mutable and immutable objects. Lists, dictionaries, and sets are instances of mutable objects, while numbers, strings, and tuples are instances of immutable objects.

Mutable Objects	Immutable Objects
Lists	Numbers
Dictionaries	Strings
Sets	Tuples

Example for mutable objects:

```
# List
my_list = [1, 2, 3]
my_list[0] = 'a'  # The list now becomes ['a', 2, 3]

# Dictionary
my_dict = {'name': 'John', 'age': 30}
my_dict['age'] = 31  # The dictionary now becomes {'name': 'John', 'age': 31}

# Set
my_set = {1, 2, 3}
my_set.add(4)  # The set now becomes {1, 2, 3, 4}
```

Q 4 a Explain the following list methods with examples:
i) index () ii) append( ) iii) insert() iv) sort() v) reverse() vi) List concatenation and Replication [10 Marks]

Explanation - [6 Marks]
Example - [4 Marks]

i) index():
The index() method returns the position at the first occurrence of the specified value.

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
    >>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
    >>> spam.index('Pooka')
1
```

ii) append():
append() method call adds the argument to the end of

the list.

```
    >>> spam = ['cat', 'dog', 'bat']
    >>> spam.append('moose')
    >>> spam
['cat', 'dog', 'bat', 'moose']
```

iii) insert():
The insert() method can insert a value at any index in the list.

```
    >>> spam = ['cat', 'dog', 'bat']
    >>> spam.insert(1, 'chicken')
    >>> spam
```

['cat', 'chicken', 'dog', 'bat']

iv) sort():

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

v) reverse():

The reverse() method reverses the sorting order of the elements.

```
>>> fruits = ['apple', 'banana', 'cherry']
>>> fruits.reverse()
```

vi) List concatenation and Replication

➢ The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value.

➢ The * operator can also be used with a list and an integer value to replicate the list.

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

Q 4 b : Develop a program to read the student details like Name, USN and Marks in three subjects: Display the student details, total marks and percentage with suitable messages. [10 Marks]

Full Program- [7 Marks]
Logic-[3 Marks]

```
name = input("Enter your name: ")
usn = input("Enter your USN: ")
sub01 = int(input("Enter your subject 01 marks out of 100 : "))
sub02 = int(input("Enter your subject 02 marks out of 100 "))
sub03 = int(input("Enter your subject 03 marks out of 100 "))

total = sub01+sub02+sub03
per = (total/300)*100

print("Name: ", name)
print("USN: ", usn)
print("Subject 01 Marks: ", sub01)
print("Subject 02 Marks: ", sub02)
print("Subject 03 Marks: ", sub03)
print("Total Marks: ", total)
print("Percentage: ", per, "%")
```

Q 5 a Illustrate with example opening of a life with open( ) function, reading the contents of the file with read() and writing to files with write (). [10 Marks]

The File Reading/Writing Process

Once you are comfortable working with folders and relative paths, you'll be able to specify the location of files to read and write. The functions covered in the next few sections will apply to plaintext files. *Plaintext files* contain only basic text characters and do not include font, size, or color information. Text files with the *.txt* extension or Python script files with the *.py* extension are examples of plaintext files. These can be opened with Windows's Notepad or OS X's TextEdit application. Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense, like in Figure 8-5.

Figure 8-5: The Windows `calc.exe` program opened in Notepad

Since every different type of binary file must be handled in its own way, this book will not go into reading and writing raw binary files directly. Fortunately, many modules make working with binary files easier—you will explore one of them, the `shelve` module, later in this chapter.

There are three steps to reading or writing files in Python.

1. Call the `open()` function to return a `File` object.
2. Call the `read()` or `write()` method on the `File` object.
3. Close the file by calling the `close()` method on the `File` object.

Reading the Contents of Files

Now that you have a `File` object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the `File` object's `read()` method. Let's continue with the *hello.txt* `File` object you stored in `helloFile` . Enter the following into the interactive shell:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

If you think of the contents of a file as a single large string value, the `read()` method returns the string that is stored in the file.

Alternatively, you can use the `readlines()` method to get a *list* of string values from the file, one string for each line of text. For example, create a file named *sonnet29.txt* in the same directory as *hello.txt* and write the following text in it:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Writing to Files

Python allows you to write content to a file in a way similar to how the `print()` function "writes" strings to the screen. You can't write to a file you've opened in read mode, though. Instead, you need to open it in "write plaintext" mode or "append plaintext" mode, or *write mode* and *append mode* for short.

Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value. Pass `'w'` as the second argument to `open()` to open the file in write mode. Append mode, on the other hand, will append text to the end of the existing file. You can think of this as appending to a list in a variable, rather than overwriting the variable altogether. Pass `'a'` as the second argument to `open()` to open the file in append mode.

If the filename passed to `open()` does not exist, both write and append mode will create a new, blank file. After reading or writing a file, call the `close()` method before opening the file again.

Let's put these concepts together. Enter the following into the interactive shell:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

Q5 b Explain how to save vårіable with the shelve module. [10 Marks]

Saving Variables with the shelve Module

You can save variables in your Python programs to binary shelf files using the `shelve` module. This way, your program can restore data to variables from the hard drive. The `shelve` module will let you add Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

Enter the following into the interactive shell:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

To read and write data using the `shelve` module, you first import `shelve`. Call `shelve.open()` and pass it a filename, and then store the returned shelf value in a variable. You can make changes to the shelf value as if it were a dictionary. When you're done, call `close()` on the shelf value. Here, our shelf value is stored in `shelfFile`. We create a list `cats` and write `shelfFile['cats'] = cats` to store the list in `shelfFile` as a value associated with the key `'cats'` (like in a dictionary). Then we call `close()` on `shelfFile`.

After running the previous code on Windows, you will see three new files in the current working directory: *mydata.bak*, *mydata.dat*, and *mydata.dir*. On OS X, only a single *mydata.db* file will be created.

These binary files contain the data you stored in your shelf. The format of these binary files is not important; you only need to know what the `shelve` module does, not how it does it. The module frees you from worrying about how to store your program's data to a file.

Your programs can use the `shelve` module to later reopen and retrieve the data from these shelf files. Shelf values don't have to be opened in read or write mode—they can do both once opened. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Here, we open the shelf files to check that our data was stored correctly. Entering `shelfFile['cats']` returns the same list that we stored earlier, so we know that the list is correctly stored, and we call `close()`.

Q 6 a Explain the following string methods with examples
i) isalpha() ii) isalnum() iii)isdecimal() iv)isspace() v) istitle() [10 Marks]

The isX String Methods

➢ There are several string methods that have names beginning with the word is. These methods return a Boolean value that describes the nature of the string.

➢ Here are some common isX string methods:

o isalpha() returns True if the string consists only of letters and is not blank.
o isalnum() returns True if the string consists only of letters and numbers and is not blank.
o isdecimal() returns True if the string consists only of numeric characters and is not blank.
o isspace() returns True if the string consists only of spaces, tabs, and newlines and is not blank.

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '     '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

Q 6 b Explain about in and not in operators in string. [5 Marks]

The in and not in Operators with Strings

➢ The in and not in operators can be used with strings just like with list values.

➢ An expression with two strings joined using in or not in will evaluate to a Boolean True or False.

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

➢ These expressions test whether the first string (the exact string, case sensitive) can be found within the second string.

Q 6 c Explain about pyperclip module. [5 Marks]

The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer's clipboard.

```
>>> import pyperclip
>>> pyperclip.copy('Hello world!')
>>> pyperclip.paste()
'Hello world!'
```

```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

➢ **O**f course, if something outside of your program changes the clipboard contents, the paste() function will return it.

Q 7 a What are Assertions? Write the conteñts of an assert statement. Explain them with examples. [10 Marks]

Assertions

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised. In code, an assert statement consists of the following:

• The assert keyword

• A condition (that is, an expression that evaluates to True or False)

• A comma

• A string to display when the condition is False

For example, enter the following into the interactive shell:

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can't do that.''
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

Here we've set podBayDoorStatus to 'open', so from now on, we fully expect the value of this variable to be 'open'. In a program that uses this variable, we might have written a lot of code under the assumption that the value is 'open'—code that depends on its being 'open' in order to work we expect. So we add an assertion to make sure we're right to assume podBayDoorStatus is 'open'. Here, we include the message 'The pod bay doors need to be "open".' so it'll be easy to see what's wrong if the assertion fails. Later, say we make the obvious mistake of assigning podBayDoorStatus

another value, but don't notice it among many lines of code. The assertion catches this mistake and clearly tells us what's wrong.

Assertions are for programmer errors, not user errors. For errors that can be recovered from (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an assert statement.

Disabling Assertions

Assertions can be disabled by passing the `-O` option when running Python. This is good for when you have finished writing and testing your program and don't want it to be slowed down by performing sanity checks (although most of the time `assert` statements do not cause a noticeable speed difference). Assertions are for development, not the final product. By the time you hand off your program to someone else to run, it should be free of bugs and not require the sanity checks. See Appendix B for details about how to launch your probably-not-insane programs with the `-O` option.

Q 7b Develop a program with a function named DivExp which takes Two parameters a, b and return_ a, vàlde c(c = a/b), write suitable ássertion for a >0 in function DivExp and /raise an exception for when b =0. Develop a suitable program which reads two values from the console and calls a function DivExp. [10 Marks]

```
def DivExp(a, b):
    # Assertion to ensure a > 0
    assert a > 0, "Value of 'a' must be greater than 0"

    # Exception handling for division by zero
    if b == 0:
        raise ValueError("Value of 'b' cannot be 0")

    # Perform the division
    c = a / b
    return c
```

```
# Function to read two values from the console
def read_values():
    try:
        a = float(input("Enter value for 'a': "))
        b = float(input("Enter value for 'b': "))
        return a, b
    except ValueError:
        print("Invalid input. Please enter numeric values.")
        return read_values()

# Main program
def main():
    a, b = read_values()
    try:
        result = DivExp(a, b)
        print(f"The result of a/b is: {result}")
    except AssertionError as e:
        print(e)
    except ValueError as e:
        print(e)

# Call the main function
if __name__ == "__main__":
    main()
```

Q 8 a Explain about files and folders can be copied using shutil module. [10 Marks]

i) Copying files and folders

The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the shutil functions, you will first need to use import shutil. The shutil module provides functions for copying files, as well as entire folders. Calling shutil.copy(source, destination) will copy the file at the path source to the folder at the path

destination. (Both source and destination are strings.) If destination is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.
While shutil.copy() will copy a single file, shutil.copytree() will copy an entire folder and every folder and file contained in it. Calling shutil.copytree(source ,destination) will copy the folder at the path source, along with all of its files and subfolders, to the folder at the path destination.

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
>>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

ii) Moving file & Folders

Calling shutil.move(source, destination) will move the file or folder at the path source to the path destination and will return a string of the absolute path of the new location. If destination points to a folder, the source file gets moved into destination and keeps its current filename.

```
>>>import shutil
>>>shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

iii) Permanently deleting file & Folders

You can delete a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, you use the shutil module.

• Calling os.unlink(path) will delete the file at path.

• Calling os.rmdir(path) will delete the folder at path. This folder must be empty of any files or folders.

• Calling shutil.rmtree(path) will remove the folder at path, and all files and folders it contains will also be deleted.

Code:

```
import os
for filename in os.listdir():
        if filename.endswith('.txt'):
                os.unlink(filename)
```

Q 8 b Explain about Debug control window [10 Marks]

The program will stay paused until you press one of the five buttons in the Debug Control window: Go, Step, Over, Out, or Quit.

Go Clicking the Go button will cause the program to execute normally until it terminates or reaches a breakpoint. If you are done debugging and want the program to continue normally, click the Go button.

Step Clicking the Step button will cause the debugger to execute the next line of code and then pause again. The Debug Control window's list of global and local variables will be updated if their values change. If the next line of code is a function call, the debugger will "step into" that function and jump to the first line of code of that function.

Over Clicking the Over button will execute the next line of code, similar to the Step button. However, if the next line of code is a function call, the Over button will "step over" the code in the function. The function's code will be executed at full speed, and the debugger will pause as soon as the function call returns.

Out Clicking the Out button will cause the debugger to execute lines of code at full speed until it returns from the current function. If you have stepped into a function call with the Step button and now simply want to keep executing instructions until you get back out, click the Out button to "step out" of the current function call.

Quit If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the Quit button. The Quit button will immediately terminate the program. If you want to run your program normally again, select Debug 4 Debugger again to disable the debugger.

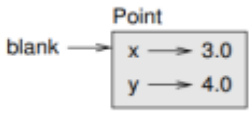
Q 9 a Explain about class and objects [10 Marks]

A programmer-defined type is called a class. A class definition looks like this:

class Point:
"""Represents a point in 2-D space."""
The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition.
Defining a class named Point creates a class object.
>>> Point
<class __main__.Point>
Because Point is defined at the top level, its "full name" is __main__.Point. The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.
>>> blank = Point()
>>> blank

```
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to blank. Creating a new object is called instantiation, and the object is an instance of the class. When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal). Every object is an instance of some class, so "object" and "instance" are interchangeable.
You can assign values to an instance using dot notation:
>>> blank.x = 3.0
>>> blank.y = 4.0
though, we are assigning values to named elements of an object. These elements are called attributes. A state diagram that shows an object and its attributes is called an object diagram.



The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number. You can read the value of an attribute using the same syntax

```
import math

def print_point(p):
    print('(%g, %g)' % (p.x, p.y))

def distance(q):
    d = math.sqrt((q.x)**2+(q.y)**2)
    print ('Distance is %g'% (d))

class Point:
    """Represents a point in 2-D space."""

blank = Point()
blank.x = 3.0
blank.y = 4.0

print_point(blank)
distance(blank)
```

```
(3, 4)
Distance is 5
```

Q 9 b Explain about pure function and modifier.[10 Marks]

PURE FUNCTION

The function that creates a new Time object, initializes its attributes with new values and returns a reference to the new object. This is called a pure function because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

```
class Time:
    """Represents the time of day.
    attributes: hour, minute, second"""


def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum

def print_time(t):
    print('Hour:', t.hour, '\nMinute: ', t.minute, '\nSeconds: ', t.second)

start = Time()
start.hour = 9
start.minute = 45
start.second = 0

duration = Time()
duration.hour = 1
duration.minute = 35
duration.second = 0

done = add_time(start, duration)
print_time(done)
print(start.hour)
```

```
Hour: 11
Minute:  20
Seconds:  0
9
```

MODIFIER FUNCTION

Sometimes it is useful for a function to modify the objects or instances it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called modifiers. increment() function adds a given number of seconds to a Time object or instance which is visible to the called function.

```
class Time:
    """Represents the time of day.
    attributes: hour, minute, second"""

def increment(time, seconds):
    time.second += seconds

    m = int (time.second/60)              # 5000 // 60= 83
    time.second = time.second - (m*60)    # 5000-(83*60)=20
    time.minute += m                      # 45+83 = 128

    h = int (time.minute/60)              # 128//60= 2
    time.minute = time.minute - (h*60)    # 128-2*60=128-120=8
    time.hour += h                        # 9 + 2 = 11


def print_time(t):
    print('Hour:', t.hour, '\nMinute: ', t.minute, '\nSeconds: ', t.second)

start = Time()
start.hour = 9
start.minute = 45
start.second = 0

seconds = 5000
#print_time(start)
increment(start, seconds)
print_time(start)
print(start.hour)
```

```
Hour: 11
Minute:  8
Seconds:  20
11
```

Q 10 a Explain the concept of prototyping Vs planning [10 Marks]

The development plan I am demonstrating is called “prototype and patch”. For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

This approach can be effective, especially if you don’t yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is designed development, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60 (see http://en.wikipedia.org/wiki/Sexagesimal.)! The second attribute is the

“ones column”, the minute attribute is the “sixties column”, and the hour attribute is the “thirty-six hundreds column”.

When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts Times to integers:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that **time_to_int(int_to_time(x)) == x** for many values of x. This is an example of a consistency check.

Once you are convinced they are correct, you can use them to rewrite add_time:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

This version is shorter than the original, and easier to verify. As an exercise, rewrite increment using time_to_int and int_to_time.

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (time_to_int and int_to_time), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct. Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

Q 10 bExplain __init__ and __str__ methods with examples. [10 Marks]

_ _init_ _ ( ) method:

The init method ("initialization") is a special method that gets invoked when an object is instantiated. Its full name is _ _init_ _ (two underscore characters, followed by init, and then two more underscores).

```
class Time:
    """Represents the time of day. attributes: hour, minute, second"""

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def print_time(t):
        print('Hour:', t.hour, '\nMinute: ', t.minute, '\nSeconds: ', t.second)

print('----------------')
time1 = Time()
time1.print_time()

print('----------------')
time2 = Time(10)
time2.print_time()

print('----------------')
time2 = Time(10,20)
time2.print_time()

print('----------------')
time2 = Time(10,20,30)
time2.print_time()
```

```
----------------
Hour: 0
Minute:  0
Seconds:  0
----------------
Hour: 10
Minute:  0
Seconds:  0
----------------
Hour: 10
Minute:  20
Seconds:  0
----------------
Hour: 10
Minute:  20
Seconds:  30
```

_ _str_ _ ( ) method

__str__ is a special method, like __init__, that is supposed to return a string representation of an object. When you print an object, Python invokes the str method

```
class Time:
    """Represents the time of day. attributes: hour, minute, second"""

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

print('----------------')
time1 = Time()
print(time1)

print('----------------')
time2 = Time(10, 20)
print(time2)
```

```
----------------
00:00:00
----------------
10:20:00
```