CBCS SCHEME

USN | | | | | | | | | | |

22MCA31

Third Semester MCA Degree Examination, Dec.2024/Jan.2025
Data Analytics using Python

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Describe arithmetic operators, assignment operators, logical operators and comparision operators in detail with example	10	L2	CO1
	b.	Explain with syntax and example different types of python data types and type( ) function.	10	L2	CO1
		OR			
Q.2	a.	Explain how to define and call a user defined function in python. Write a python program to check whether a given number is odd or even using function.	10	L2	CO1
	b.	Discuss different forms of if control statements with example. Explain the need of break and continue statements.	10	L2	CO1
		Module – 2			
Q.3	a.	Explain any five operations performed on string with an example.	10	L2	CO2
	b.	Explain list creation, indexing and built-in functions used on lists with syntax and examples.	10	L2	CO2
		OR			
Q.4	a.	Create a function product and demonstrate function overloading by accepting input and print their product.	10	L3	CO2
	b.	What is a file? What are the different modes of opening a file?	10	L3	CO2
		Module – 3			
Q.5	a.	What is data preprocessing? Explain the steps involved in preprocessing the data.	10	L3	CO3
	b.	Discuss in detail about data transformations.	10	L3	CO3
		OR			
Q.6	a.	Write a note on string manipulation using the regular expression methods.	10	L3	CO3
	b.	Explain combining and merging data-sets with an example.	10	L3	CO3

		Module – 4			
Q.7	a.	What is web scrapping? Explain the python libraries used for web scrapping.	10	L3	CO4
	b.	Explain the steps involved in fetching the data by submitting the form.	10	L3	CO4
		OR			
Q.8	a.	Explain NumPy array attributes with example.	10	L3	CO4
	b.	Write a note on aggregation functions available in NumPy with example.	10	L3	CO4
		Module – 5			
Q.9	a.	Implement a python program to demonstrate data visualization using matplotlib.	10	L3	CO4
	b.	Write short notes on: i) Matplot library ii) Seaborn library.	10	L3	CO4
		OR			
Q.10	a.	Create a python program to demonstrate data visualization to plot a line plot, scatter plot using seaborn.	10	L3	CO4
	b.	Write a note on multiple subplots with examples.	10	L3	CO4

* * * * *

1.a) Operators

Arithmetic Operators

Arithmetic operators are used to performing mathematical operations like addition, subtraction, multiplication, and division.

Operator	Description	Syntax
+	Addition: adds two operands	x + y
–	Subtraction: subtracts two operands	x – y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power: Returns first raised to power second	x ** y

Example: Arithmetic operators in Python

```
# Examples of Arithmetic Operator
a = 9
b = 4

# Addition of numbers
add = a + b

# Subtraction of numbers
sub = a - b

# Multiplication of number
mul = a * b

# Division(float) of number
div1 = a / b

# Division(floor) of number
div2 = a // b
```

```
# Modulo of both number
mod = a % b

# Power
p = a ** b

# print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
print(p)
```

Output

```
13
5
36
2.25
2
1
6561
```

Note: Refer to Differences between / and // for some interesting facts about these two operators.

Comparison Operators

Comparison of Relational operators compares the values. It either returns **True** or **False** according to the condition.

Operator	Description	Syntax
>	Greater than: True if the left operand is greater than the right	x > y
<	Less than: True if the left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to – True if operands are not equal	x != y
>=	Greater than or equal to True if the left operand is greater than or equal to the right	x >= y

Operator	Description	Syntax
<=	Less than or equal to True if the left operand is less than or equal to the right	x <= y

Example: Comparison Operators in Python

```
# Examples of Relational Operators
a = 13
b = 33

# a > b is False
print(a > b)

# a < b is True
print(a < b)

# a == b is False
print(a == b)

# a != b is True
print(a != b)

# a >= b is False
print(a >= b)

# a <= b is True
print(a <= b)
```

Output

```
False
True
False
True
False
True
```

Logical Operators

Logical operators perform **Logical AND**, **Logical OR**, and **Logical NOT** operations. It is used to combine conditional statements.

Operator	Description	Syntax
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y

Operator	Description	Syntax
not	Logical NOT: True if the operand is false	not x

Example: Logical Operators in Python

```
# Examples of Logical Operator
a = True
b = False

# Print a and b is False
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(not a)
```

Output

```
False
True
False
```

Bitwise Operators

Bitwise operators act on bits and perform the bit-by-bit operations. These are used to operate on binary numbers.

Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Example: Bitwise Operators in Python

```
# Examples of Bitwise operators
a = 10
b = 4

# Print bitwise AND operation
print(a & b)

# Print bitwise OR operation
print(a | b)

# Print bitwise NOT operation
print(~a)

# print bitwise XOR operation
print(a ^ b)

# print bitwise right shift operation
print(a >> 2)

# print bitwise left shift operation
print(a << 2)
```

Output

```
0
14
-11
14
2
40
```

Assignment Operators

Assignment operators are used to assigning values to the variables.

Operator	Description	Syntax
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right-side operand with left side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a*=b a=a*b

Operator	Description	Syntax
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign the result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b a= a << b

Example: Assignment Operators in Python

```
# Examples of Assignment Operators
a = 10

# Assign value
b = a
print(b)

# Add and assign value
b += a
print(b)
```

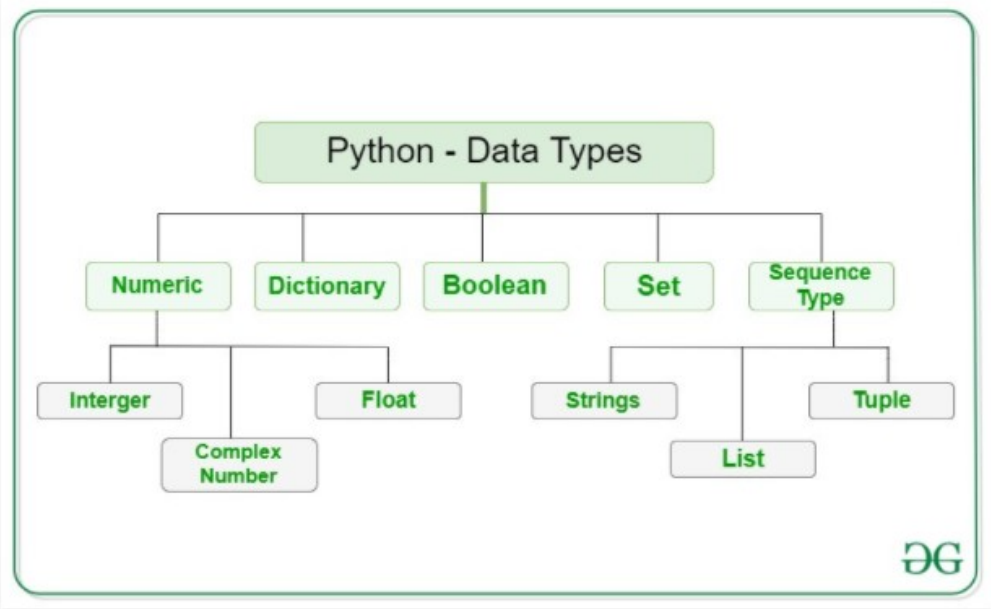
```
# Subtract and assign value
b -= a
print(b)

# multiply and assign
b *= a
print(b)

# bitwise lishift operator
b <<= a
print(b)
```

Output

```
10
20
10
100
102400
```

1.b) Datatypes

1. **Data Types,**



In programming, data type is an important concept.
Variables can store data of different types, and different types can do different things.
Python has the following data types built-in by default, in these categories:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

Getting the Data Type
You can get the data type of any object by using the type() function:
Example
Print the data type of the variable x:

```
x = 5
print(type(x))
```

Setting the Data Type
In Python, the data type is set when you assign a value to a variable:

Example	Data Type

x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Setting the Specific Data Type
If you want to specify the data type, you can use the following constructor functions:

Example	**Data Type**
x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	list
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict

x = set(("apple", "banana", "cherry"))	set
x = frozenset(("apple", "banana", "cherry"))	frozenset
x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

2. User defined function

In Python, a **user-defined function** is a reusable block of code created using the def keyword. Here's a simple example showing how to define and call a user-defined function.

☑Defining a Function

```
def greet(name):
    print("Hello, " + name + "!")
```

This defines a function called greet that takes one parameter name.

☑Calling the Function

```
greet("Alice")
greet("Bob")
```

□ Output:

```
Hello, Alice!
Hello, Bob!
```

More Examples

Example 1: Function with No Parameters

```
def say_hello():
    print("Hello, World!")

say_hello()
```

Example 2: Function with Return Value

```
def add(a, b):
```

```
    return a + b

result = add(5, 3)
print("Sum is:", result)
```

Odd or even :

```
def check_odd_even(number):
    if number % 2 == 0:
        print(f"{number} is Even")
    else:
        print(f"{number} is Odd")
```

1. **b)**

Syntax:

```
if condition1:
        statement(s)
else:
        statement(s)
```

```
x = 34 - 23                 # A comment.
y = "Good"                  # Another one.
z = 3.45
if z == 3.45 or y == "Session":
    x = x + 1
    y = y + "Session"       # String concat.
print(x)
print(y)
```

Decision making is required when we want to execute a code only if a certain condition is satisfied.
The if…elif…else statement is used in Python for decision making.

Python if Statement Syntax

```
if test expression:
    statement(s)
```

Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True.
If the test expression is False, the statement(s) is not executed.
In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end.
Python interprets non-zero values as True. None and 0 are interpreted as False.

Python if Statement Flowchart

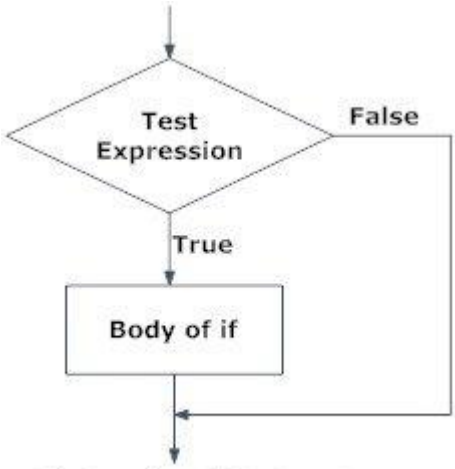


Fig: Operation of if statement

Flowchart of if statement in Python programming

Example: Python if Statement

```
# If the number is positive, we print an appropriate message

num = 3
if num > 0:
    print(num, "is a positive number.")
print("This is always printed.")

num = -1
if num > 0:
    print(num, "is a positive number.")
print("This is also always printed.")
```

When you run the program, the output will be:

```
3 is a positive number
This is always printed
This is also always printed.
```

In the above example, num > 0 is the test expression.
The body of if is executed only if this evaluates to True.
When the variable num is equal to 3, test expression is true and statements inside the body of if are executed.
If the variable num is equal to -1, test expression is false and statements inside the body of if are skipped.
The print() statement falls outside of the if block (unindented). Hence, it is executed regardless of the test expression.

Python if...else Statement

Syntax of if...else

```
if test expression:
   Body of if
else:
   Body of else
```

The if..else statement evaluates test expression and will execute the body of if only when the test condition is True.
If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

Python if..else Flowchart

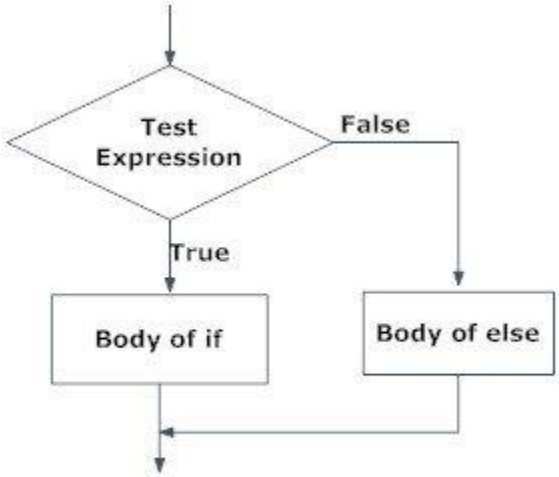


Fig: Operation of if...else statement

Flowchart of if...else statement in Python

Example of if...else

```
# Program checks if the number is positive or negative
# And displays an appropriate message

num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
   print("Positive or Zero")
else:
   print("Negative number")
```

Output

```
Positive or Zero
```

In the above example, when num is equal to 3, the test expression is true and the body of if is executed and the body of else is skipped.
If num is equal to -5, the test expression is false and the body of else is executed and the body of if is skipped.
If num is equal to 0, the test expression is true and body of if is executed and body of else is skipped.

Python if...elif...else Statement

Syntax of if...elif...else

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

The elif is short for else if. It allows us to check for multiple expressions.
If the condition for if is False, it checks the condition of the next elif block and so on.
If all the conditions are False, the body of else is executed.
Only one block among the several if...elif...else blocks is executed according to the condition.
The if block can have only one else block. But it can have multiple elif blocks.

Flowchart of if...elif...else

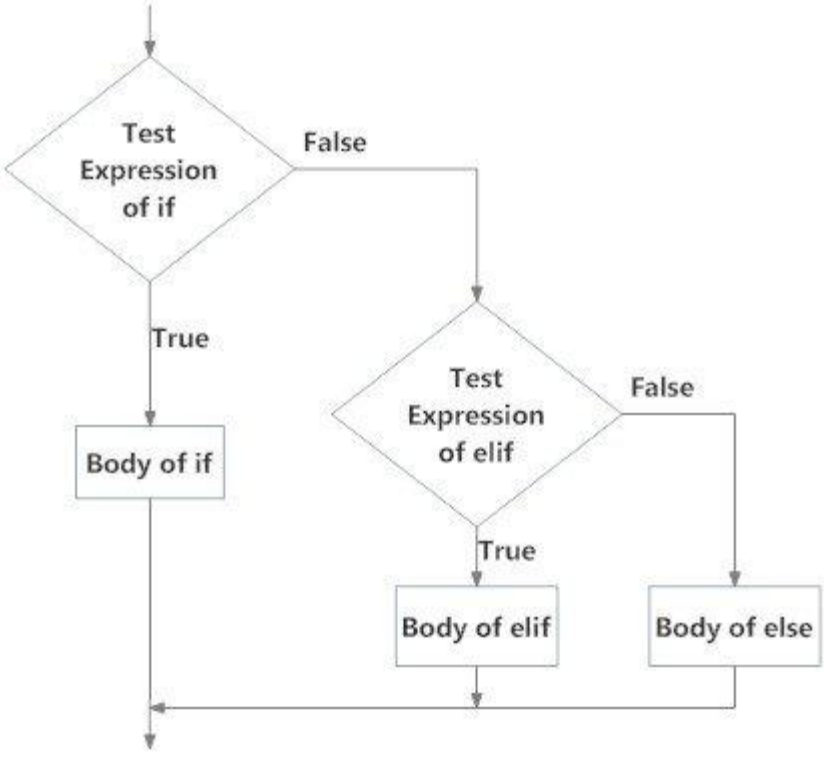


Fig: Operation of if...elif...else statement

Flowchart of if...elif....else statement in Python

Example of if...elif...else

```
'''In this program,
we check if the number is positive or
negative or zero and
display an appropriate message'''

num = 3.4

# Try these two variations as well:
# num = 0
# num = -4.5

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

When variable num is positive, Positive number is printed.
If num is equal to 0, Zero is printed.
If num is negative, Negative number is printed.

Python Nested if statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

Python Nested if Example

```
'''In this program, we input a number
check if the number is positive or
negative or zero and display
an appropriate message
This time we use nested if statement'''

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Output 1

```
Enter a number: 5
Positive number
```

Output 2

```
Enter a number: -1
Negative number
```

Output 3

```
Enter a number: 0
Zero
```

Break and Continue:

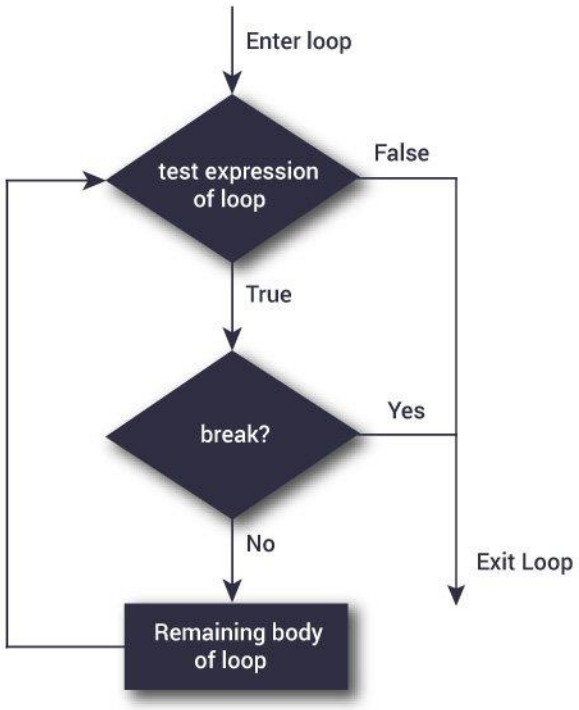
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

Syntax of break

```
break
```

Flowchart of break



Flowchart of break statement in Python

The working of break statement in for loop and while loop is shown below.

```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

Working of the break statement

Example: Python break

```
# Use of break statement inside the loop
```

```
for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

Output

```
s
t
r
The end
```

In this program, we iterate through the "string" sequence. We check if the letter is i, upon which we break from the loop. Hence, we see in our output that all the letters up till i gets printed. After that, the loop terminates.

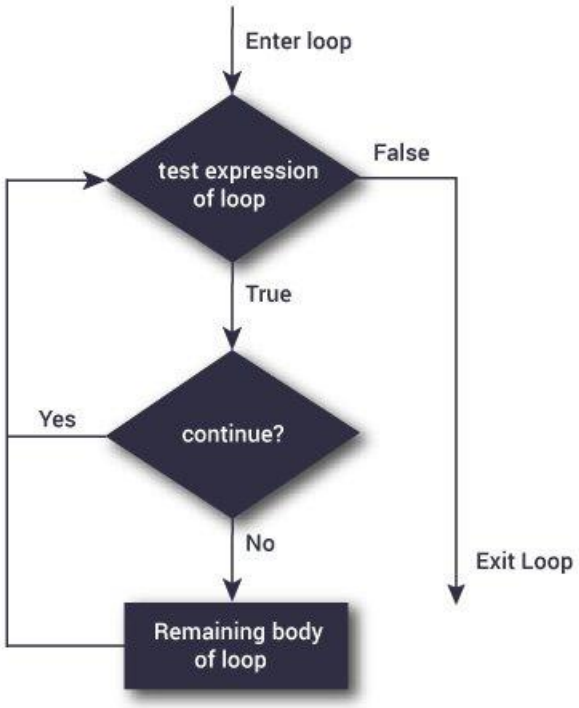
Python continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Syntax of Continue

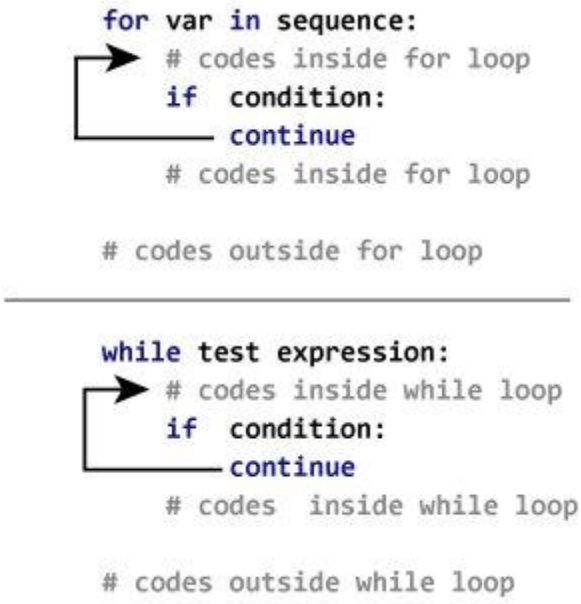
```
continue
```

Flowchart of continue



Flowchart of continue statement in Python

The working of the continue statement in for and while loop is shown below.



How continue statement works in python

Example: Python continue

```
# Program to show the use of continue statement inside loops

for val in "string":
   if val == "i":
      continue
   print(val)

print("The end")
```

Output

```
s
t
r
n
g
The end
```

This program is same as the above example except the break statement has been replaced with continue.
We continue with the loop, if the string is i, not executing the rest of the block. Hence, we see in our output that all the letters except i gets printed.

What is pass statement in Python?

In Python programming, the pass statement is a null statement. The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored.
However, nothing happens when the pass is executed. It results in no operation (NOP).

3. a) String operations

Traversal with a for **loop**

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit): letter =
    fruit[index] print letter
        index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

The following example shows how to use concatenation (string addition) and a for loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'


for letter in prefixes: print letter +
    suffix
    The output is:
    Jack
    Kack
    Lack
    Mack
    Nack
    Oack
    Pack
    Qack
```

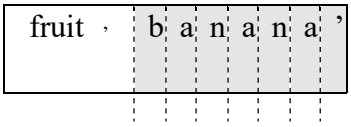
Of course, that's not quite right because "Ouack" and "Quack" are misspelled.

2. String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:13]
Python
```

The operator [n:m] returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3] ''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

3. Strings are immutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence.

The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

4. Searching

What does the following function do?

```
def find(word, letter): index = 0
        while index < len(word):
        if word[index] == letter: return
            index
            index = index + 1
        return -1
```

In a sense, find is the opposite of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1.

This is the first example we have seen of a return statement inside a loop. If word[index] == letter, the function breaks out of the loop and returns immediately.
If the character doesn't appear in the string, the program exits the loop normally and returns -1.
This pattern of computation—traversing a sequence and returning when we find what we are looking for—is a called a **search**.

5. Looping and counting

The following program counts the number of times the letter a appears in a string:

```
        word = 'banana'
        count = 0
    for letter in word: if letter ==
        'a':
                count = count + 1
        print count
```

This program demonstrates another pattern of computation called a **counter**. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result—the total number of a's.

6. String **methods**

A **method** is similar to a function—it takes arguments and returns a value—but the syntax is dif- ferent. For example, the method upper takes a string and returns a new string with all uppercase letters:

Instead of the function syntax upper(word), it uses the method syntax word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking upper on the word. As it turns out, there is a string method named find that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index 1
```

In this example, we invoke find on word and pass the letter we are looking for as a parameter.

Actually, the find method is more general than our function; it can find substrings, not just charac- ters:

```
>>> word.find('na') 2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

And as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because b does not appear in the index range from 1 to 2 (not including 2).

7. **The** in **operator**

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 'a' in 'banana' True
>>> 'seed' in 'banana' False
```

For example, the following function prints all the letters from word1 that also appear in word2:

```
def in_both(word1, word2): for
    letter in word1:
```

```
    if letter in word2: print
        letter
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Here’s what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges') a
e
sstri
ng
com
pari
son
```

The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print    'All right, bananas.'
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.' elif
word > 'banana':
    print 'Your word,' + word + ', comes after banana.' else:
    print 'All right, bananas.'
```

Python does not handle uppercase and lowercase letters the same way that people do. All the upper- case letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

A common way to address this problem is to convert strings to a standard format, such as all low- ercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

B) List

A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don’t have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

['spam', 2.0, 5, [10, 20]]

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, []. As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty ['Cheddar',
'Edam', 'Gouda'] [17, 123] []
```

Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

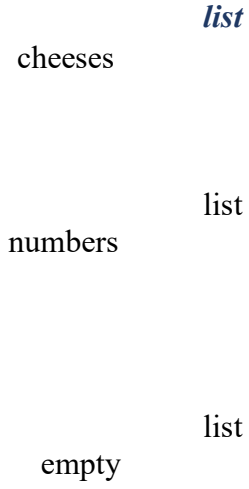
```
>>> print cheeses[0]
Cheddar
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a **mapping**; each index "maps to" one of the elements. Here is a state diagram showing cheeses, numbers and empty:



Lists are represented by boxes with the word "list" outside and the elements of the list inside. cheeses refers to a list with three elements indexed 0, 1 and 2. numbers

contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. empty refers to a list with no elements.
List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an IndexError.
- If an index has a negative value, it counts backward from the end of the list. The in operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

Traversing a list

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
for cheese in cheeses: print
    cheese
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions range and len:

```
for i in range(len(numbers)): numbers[i]
    = numbers[i] * 2
```

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to *n* 1, where *n* is the length of the list. Each time through the loop i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

A for loop over an empty list never executes the body:

```
for x in empty:
    print 'This never happens.'
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]

List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] ['b',
'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

List methods

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

extend takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

Map, filter and reduce

To add up all the numbers in a list, you can use a loop like this:

```
def add_all(t): total = 0
    for x in t:
        total += x
    return total
```

total is initialized to 0. Each time through the loop, x gets one element from the list. The += operator provides a short way to update a variable:

```
    total += x
```

is equivalent to:

```
    total = total + x
```

As the loop executes, total accumulates the sum of the elements; a variable used this way is sometimes called an **accumulator**.
Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

```
>>> t = [1, 2, 3]
>>> sum(t) 6
```

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce.**
Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
def capitalize_all(t): res = []
        for s in t:
            res.append(s.capitalize())
        return res
```

res is initialized with an empty list; each time through the loop, we append the next element. So res is another kind of accumulator.
An operation like capitalize_all is sometimes called a **map** because it "maps" a function (in this case the method capitalize) onto each of the elements in a sequence.
Another common operation is to select some of the elements from a list and return a sublist. For ex- ample, the following function takes a list of strings and returns a list that contains only the uppercase strings: def only_upper(t): res = []

```
for s in t:
if s.isupper():
    res.append(s)
return res
```

isupper is a string method that returns True if the string contains only upper case letters.
An operation like only_upper is called a **filter** because it selects some of the elements and filters out the others.
Most common list operations can be expressed as a combination of map, filter and reduce. Because these operations are so common, Python provides language features to support them, including the built-in function map and an operator called a "list comprehension."

Deleting elements
There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>>
print['a','c']
>>> print    x
b
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don’t need the removed value, you can use the del operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a','c']
```

If you know the element you want to remove (but not the index), you can use remove:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t ['a',
'c']
```

The return value from remove is None.

To remove more than one element, you can use del with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t ['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

Because list is the name of a built-in function, you should avoid using it as a variable name. I also avoid l because it looks too much like 1. So that’s why I use t.

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
```

An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
```

```
>>> s.split(delimiter) ['spam',
'spam', 'spam']
```

join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so join puts a space between words. To concatenate strings without spaces, you can use the empty string, '', as a delimiter.

Objects and values

If we execute these assignment statements:

```
a = 'banana' b =
'banana'
```

We know that a and b both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:

In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the is operator.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In this example, Python only created one string object, and both a and b refer to it. But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

So the state diagram looks like this:

In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you execute a = [1,2,3], a refers to a list

object whose value is a particular sequence of elements. If another list has the same elements, we would say it has the same value.

List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, delete_head removes the first element from a list:

```
def delete_head(t): del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters ['b',
'c']
```

The parameter t and the variable letters are aliases for the same object. The stack diagram looks like this:

Since the list is shared by two frames, I drew it between them.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1 [1,
2, 3]
>>> print t2
None
```

```
>>> t3 = t1 + [3]
>>> print t3 [1,
2, 3]
>>> t2 is t3
False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]  # WRONG!
```

The slice operator creates a new list and the assignment makes t refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, tail returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b','c']
```

Q4 (a):

Answer:

Python does not support traditional function overloading like C++ or Java. However, we can simulate it using default arguments, *args, or by checking argument types inside the function.

Example using *args to simulate overloading:

```
def product(*args):
    result = 1
    for num in args:
        result *= num
    print("Product is:", result)

# Demonstration
product(4, 5)          # Product of 2 numbers
product(2, 3, 4)       # Product of 3 numbers
product(7)             # Product of 1 number
```

Explanation:

- *args allows the function to accept any number of arguments.
- This mimics function overloading behavior by handling different input sizes.

Q4 (b): What is a file? What are the different modes of opening a file?

Answer:

File:
A file is a named location on disk used to store related data. In Python, you can use the open() function to interact with files.

File Modes in Python:

Mode	Description
'r'	Read mode (default). Opens the file for reading.
'w'	Write mode. Creates a new file or overwrites an existing file.
'a'	Append mode. Adds new data at the end of the file.
'r+'	Read and write mode. The file pointer is placed at the beginning.
'w+'	Write and read mode. Overwrites the file if it exists.
'a+'	Append and read mode. Adds data to the end and allows reading.
'b'	Binary mode. Used to handle non-text files like images. (E.g., 'rb', 'wb')

Example:

```
# Opening a file in write mode
f = open("example.txt", "w")
f.write("Hello, file!")
f.close()
```

5. a)Data Preprocessing

Cleansing data is a critical step in the data preparation process to ensure accuracy, consistency, and reliability in your datasets.

Python offers several libraries and techniques for cleansing data efficiently.

Steps to do data cleaning

Importing necessary libraries

Loading data

Identifying missing values

Handling Duplicates

Data Transformation

Removing Duplicates

Replacing values

Handling Outliers

Data Validation

Importing necessary libraries

We generally use numpy and pandas to perform data cleansing.

import pandas as pd

import numpy as np

Loading data

We load data(reading and writing) in the form of a csv file or a json file.

Handling Missing Data

Missing data occurs commonly in many data analysis applications. One of the goals of pandas is to make working with missing data as much as possible.

For numeric data, pandas uses the floating-point value NaN (Not a Number) to represent missing data. We call this a sentinel value that can be easily detected.

Identifying and handling missing values

isnull(): This method returns a Boolean DataFrame showing True for cells containing missing values (NaN), and False otherwise.

notnull(): This method returns a Boolean DataFrame

showing False for cells containing missing values (NaN), and True otherwise.

Dropping missing data

info(): This method provides a concise summary of a DataFrame, including the data types, non-null values, memory usage, and other essential information.

dropna(): This method allows you to remove missing values from a DataFrame or Series based on specified axis (rows or columns) and parameters like how (all, any).

Filling missing data

fillna(): This method allows you to fill missing values in a DataFrame or Series with specified values like a constant, mean, median, mode, forward-fill (ffill), or backward-fill (bfill).

Handling and removing Duplicates

Handling duplicates is an essential step in data cleansing to ensure data integrity and accuracy in your datasets.

Python's pandas library provides various methods to identify and remove duplicate rows from a DataFrame.

Data Transformation

Perform data transformation operations like converting data types, renaming columns, or creating new features.

Replacing values

Replacing values is a common operation in data cleansing to handle missing values, correct inaccuracies, or transform data for analysis.

In Python's pandas library, you can use the replace() method to replace specific values in a DataFrame or Series.

Handling Outliers

Handling outliers in Python data cleansing refers to the

process of identifying and managing outliers or extreme values in a dataset to ensure that they do not influence the analysis, modeling, or interpretation of data.

It is crucial to detect and address outliers appropriately during the data cleansing and preprocessing stages.

Data Validation

Perform data validation checks to ensure data consistency, accuracy, and integrity.

5. b) Data transformation

Data Transformation is a critical step in the **data preprocessing pipeline**, where raw data is converted into a clean and usable format for analysis or machine learning.

□ Definition:

Data Transformation is the process of converting data values, structures, or formats to ensure consistency, accuracy, and compatibility with analytical tools.

□ Common Data Transformation Techniques (with Python Examples)

1. Handling Missing Values

```
import pandas as pd

df = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, None, 30]
})

df['Age'].fillna(df['Age'].mean(), inplace=True)  # Replace missing with mean
```

---

2. Encoding Categorical Variables

```
df = pd.DataFrame({
   'Gender': ['Male', 'Female', 'Male']
})

df_encoded = pd.get_dummies(df, columns=['Gender'])
```

---

3. Normalization / Scaling

```
from sklearn.preprocessing import MinMaxScaler

df = pd.DataFrame({'Marks': [50, 80, 90]})
scaler = MinMaxScaler()
df['Normalized'] = scaler.fit_transform(df[['Marks']])
```

---

4. Changing Data Types

```
df['Marks'] = df['Marks'].astype(float)
```

---

5. Removing Duplicates

```
df.drop_duplicates(inplace=True)
```

---

6. Text (String) Transformation

```
df['Name'] = df['Name'].str.upper()  # Convert to uppercase
```

---

7. Feature Extraction (from Date, Text, etc.)

```
df = pd.DataFrame({'Date': pd.to_datetime(['2024-01-01', '2024-05-01'])})
df['Month'] = df['Date'].dt.month
```

6.a) String manipulation using regular expression:

- Regular expressions are a powerful language for matching text patterns.
- It is extremely useful for extracting information from text such as code, files, log, spreadsheets or even documents.
- While using the regular expression the first thing is to recognize is that everything is essentially a character, and we are writing patterns to match a specific sequence of characters also referred as string.
- The regular expression library re must be imported into your program before we can use it.
  https://docs.python.org/3/library/re.html
- The simplest use of the regular expression library is the search() function. The following program demonstrates a trivial use of the search function.

For example:

In mbox.txt

```
From stephen.marquard@uct.ac.za Sat Jan 5
09:14:16 2008 Return-Path:
<postmaster@collab.sakaiproject.org> Received:
from murder (mail.umich.edu [141.211.14.90])
        by frankenstein.mail.umich.edu (Cyrus v2.3.8) with
        LMTPA; Sat, 05 Jan 2008 09:14:16 -0500
X-Sieve: CMU Sieve 2.3
Received: from murder ([unix socket])
        by mail.umich.edu (Cyrus v2.2.12) with
        LMTPA; Sat, 05 Jan 2008 09:14:16 -0500
Received: from holes.mr.itd.umich.edu (holes.mr.itd.umich.edu
        [141.211.14.79]) by flawless.mail.umich.edu () with ESMTP id
        m05EEFR1013674;
        Sat, 5 Jan 2008 09:14:15 -0500
Received: FROM paploo.uhi.ac.uk (app1.prod.collab.uhi.ac.uk
        [194.35.219.184]) BY holes.mr.itd.umich.edu ID
        477F90B0.2DB2F.12494 ;
         5 Jan 2008 09:14:10 -0500
Received: from paploo.uhi.ac.uk (localhost [127.0.0.1])
        by paploo.uhi.ac.uk (Postfix) with ESMTP id
        5F919BC2F2; Sat, 5 Jan 2008 14:10:05 +0000
        (GMT)
```

☑String Manipulation Using Regular Expressions in Python

Regular expressions (RegEx) allow powerful and flexible pattern matching and string manipulation. In Python, this is done using the **`re`** module.

□ 1. Searching for a Pattern

```
text = "My phone number is 9876543210"
match = re.search(r'\d{10}', text)
if match:
    print("Phone number found:", match.group())
```

Output: `Phone number found: 9876543210`

□ 2. Finding All Matches

```
text = "Emails: alice@gmail.com, bob@yahoo.com"
emails = re.findall(r'\S+@\S+', text)
print(emails)
```

Output: `['alice@gmail.com', 'bob@yahoo.com']`

□ 3. Replacing Text

```
text = "This is a bad example."
clean_text = re.sub(r'bad', 'good', text)
print(clean_text)
```

Output: `This is a good example.`

□ 4. Splitting a String by Pattern

```
data = "one, two; three|four"
split_data = re.split(r'[;,|]\s*', data)
print(split_data)
```

Output: `['one', 'two', 'three', 'four']`

□ 5. Validating Input (e.g., Email)

```
email = "user@example.com"
if re.fullmatch(r'\w+@\w+\.\w+', email):
    print("Valid Email")
else:
    print("Invalid Email")
```

□ 6. Extracting Digits

```
text = "Order #12345 was placed on 2024-01-01"
numbers = re.findall(r'\d+', text)
print(numbers)
```

Output: `['12345', '2024', '01', '01']`

6. b)Combining and merging data sets:

Concat operation in data frame

Pandas provides various facilities for easily combining together **Series, DataFrame.**

```
pd.concat(objs, axis=0, join='outer', join_axes=None,ignore_index=False)
```

- **objs** – This is a sequence or mapping of Series, DataFrame, or Panel objects.
- **axis** – {0, 1, ...}, default 0. This is the axis to concatenate along.
- **join** – {'inner', 'outer'}, default 'outer'. How to handle indexes on other axis(es). Outer for union and inner for intersection.
- **ignore_index** – boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1.
- **join_axes** – This is the list of Index objects. Specific indexes to use for the other (n-1) axes instead of performing inner/outer set logic.

The Concat() performs concatenation operations along an axis.

Example-1

```
import pandas as pd
dic1= { 'id': ['1', '2', '3', '4', '5'],'Value1': ['A', 'C', 'E', 'G', 'I'],
        'Value2': ['B', 'D', 'F', 'H', 'J']}
dic2= {'id': ['2', '3', '6', '7', '8'],'Value1': ['K', 'M', 'O', 'Q', 'S'],
        'Value2': ['L', 'N', 'P', 'R', 'T']}
df1=pd.DataFrame(dic1)
df2=pd.DataFrame(dic2)
df3=pd.concat([df1,df2])
print(df3)

```

```
  id Value1 Value2
0  1      A      B
1  2      C      D
2  3      E      F
3  4      G      H
4  5      I      J
0  2      K      L
1  3      M      N
2  6      O      P
3  7      Q      R
4  8      S      T
```

Example-2

```
import pandas as pd
dic1= { 'id': ['1', '2', '3', '4', '5'],'Value1': ['A', 'C', 'E', 'G', 'I'],
        'Value2': ['B', 'D', 'F', 'H', 'J']}
dic2= {'id': ['2', '3', '6', '7', '8'],'Value1': ['K', 'M', 'O', 'Q', 'S'],
        'Value2': ['L', 'N', 'P', 'R', 'T']}
df1=pd.DataFrame(dic1)
df2=pd.DataFrame(dic2)
df3=pd.concat([df1,df2],ignore_index=True)
print(df3)

```

Merge operation in data frame

Two DataFrames might hold different kinds of information about the same entity and linked by some common feature/column. To join these DataFrames, pandas provides multiple functions like merge(), join() etc.

Example-1

```
import pandas as pd
dic1= { 'id': ['1', '2', '3', '4', '5'],'Value1': ['A', 'C', 'E', 'G', 'I'],
        'Value2': ['B', 'D', 'F', 'H', 'J']}
dic2= {'id': ['2', '3', '6', '7', '8'],'Value1': ['K', 'M', 'O', 'Q', 'S'],
        'Value2': ['L', 'N', 'P', 'R', 'T']}
dic3 = {'id': ['1', '2', '3', '4', '5', '7', '8', '9', '10', '11'],
        'Value3': [12, 13, 14, 15, 16, 17, 15, 12, 13, 23]}
df1=pd.DataFrame(dic1)
df2=pd.DataFrame(dic2)
df3=pd.concat([df1,df2])
df4=pd.DataFrame(dic3)
df5=pd.merge(df3,df4,on='id')
print(df5)
```

	id	Value1	Value2	Value3
0	1	A	B	12
1	2	C	D	13
2	2	K	L	13
3	3	E	F	14
4	3	M	N	14
5	4	G	H	15
6	5	I	J	16
7	7	Q	R	17
8	8	S	T	15

This will give the common rows between the two data frames for the corresponding column values ('id').

☑Types of Joins in Python (Using Pandas)

In Python, **joins** are used to combine rows from two or more **DataFrames** based on a common column or index. The `pandas` library provides SQL-style join operations using the `merge()` function.

□ 1. INNER JOIN

- Returns rows with matching values in both DataFrames.

```
import pandas as pd

df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Score': [85, 90, 95]})

result = pd.merge(df1, df2, on='ID', how='inner')
print(result)
```

Output:

```
   ID     Name  Score
0   2      Bob     85
1   3  Charlie     90
```

□ 2. LEFT JOIN

- Returns all rows from the **left** DataFrame and matched rows from the right.

```
result = pd.merge(df1, df2, on='ID', how='left')
print(result)
```

Output:

```
   ID     Name  Score
0   1    Alice    NaN
1   2      Bob   85.0
2   3  Charlie   90.0
```

□ 3. RIGHT JOIN

- Returns all rows from the **right** DataFrame and matched rows from the left.

```
result = pd.merge(df1, df2, on='ID', how='right')
print(result)
```

Output:

```
   ID     Name  Score
0   2      Bob     85
1   3  Charlie     90
2   4      NaN     95
```

□ 4. OUTER JOIN (FULL JOIN)

- Returns all rows when there is a match in one of the DataFrames.

```
result = pd.merge(df1, df2, on='ID', how='outer')
print(result)
```

Output:

```
   ID     Name  Score
0   1    Alice    NaN
1   2      Bob   85.0
2   3  Charlie   90.0
3   4      NaN   95.0
```

7. A) Web Scraping:

Web scraping in Python refers to the process of extracting data from websites.

It involves fetching the HTML content of a web page and then parsing it to extract the relevant information.

Python provides several libraries and tools that make web scraping relatively easy.

Data acquisition by scraping web applications

Data acquisition by scraping web applications involves extracting information from dynamic websites or web applications.

Unlike static websites, web applications often use JavaScript to load and manipulate content dynamically.

To scrape data from such sites, you need to consider tools and techniques that handle dynamic content.

Techniques and tools commonly used for web scraping

Inspect the Website:

Use your web browser's developer tools (usually accessible by right-clicking on a webpage and selecting "Inspect" or "Inspect Element") to analyze the structure of the HTML and identify the elements containing the data you want.

Understand AJAX Requests:

Many web applications use AJAX (Asynchronous JavaScript and XML) to load data dynamically. Investigate the network tab in your browser's developer tools to understand the AJAX requests that retrieve additional data after the initial page load.

Selenium:

Selenium is a powerful tool for web scraping that allows you to automate browser actions, including interaction with dynamic content. You can use it to control a browser, navigate through pages, and interact with elements.

Headless Browsers:

Headless browsers like pyppeteer (for Puppeteer) or selenium with a headless option can be used to run the browser without a graphical interface, which is useful for server-side scraping.

Handling Dynamic Content:

Some websites load data dynamically after the initial page load using JavaScript. In such cases, you might need to wait for elements to appear or use explicit waits in Selenium to ensure the data is loaded before attempting to scrape it.

APIs:

Check if the web application provides an API for accessing data. Using an API is often a more reliable and efficient way to obtain structured data.

Regular Expressions (Regex):

In some cases, you might need to use regular expressions to extract specific patterns from the HTML content.

7.b)
GET Request:

The get() method is used to send a GET request to the specified URL.

GET requests are used to retrieve data from the server.

Parameters are included in the URL's query string.

GET requests are generally used for retrieving data that does not require any sensitive information.

How get() works?

Sending a GET Request: When you call requests.get(url), the requests module sends an HTTP GET request to the specified url.

Retrieving the Response: The server responds to the GET request with a response, which includes the HTML content of the webpage (or other data, depending on the request).

Accessing the Response Content: You can access the content of the response using the text attribute. This attribute contains the raw HTML content of the webpage as a string.

Submitting a form using post()

In web scraping, the HTTP POST method is used to submit data to a server to create or update a resource.

This method is commonly used when interacting with web forms, as it allows you to send data to the server in the body of the request.

POST Request:

The post() method is used to send a POST request to the specified URL.

POST requests are used to submit data to the server.

Parameters are sent in the request body.

POST requests are commonly used for submitting forms or sending sensitive information (such as login credentials) to the server.

how post() works?

Identifying the Form: Before using the POST command, you need to identify the form on the webpage that you want to submit. This involves inspecting the HTML source code of the webpage to locate the form element and its input fields.

Gathering Form Data: Once you've identified the form, you need to gather the data that you want to submit. This typically involves collecting values for each input field in the form. You can do this manually or programmatically, depending on your specific use case.

Constructing the POST Request: After gathering the form data, you construct a POST request using the requests module in Python. The requests.post() function is used to send a POST request to the server. You provide the URL of the form submission endpoint as the first argument and the form data as the data parameter.

Submitting the Request: Once you've constructed the POST request, you send it to the server by calling the requests.post() function. The server processes the request and returns a response, which you can then inspect to determine if the form submission was successful.

```
import requests

# ----------- GET Request -----------
get_response = requests.get("https://jsonplaceholder.typicode.com/posts/1")

print("GET Response:")
print(get_response.status_code)  # 200 means OK
print(get_response.json())       # JSON response content

# ----------- POST Request -----------
payload = {
   "title": "foo",
   "body": "bar",
   "userId": 1
}

post_response = requests.post("https://jsonplaceholder.typicode.com/posts", json=payload)

print("\nPOST Response:")
```

```
print(post_response.status_code)  # 201 means Created
print(post_response.json())       # JSON response from server
```

8. a) Numpy attributes

1. ndarray.ndim

- **Description**: This attribute returns the number of dimensions (axes) of the array.
- **Example**:
- `import numpy as np`
- `arr = np.array([[1, 2, 3], [4, 5, 6]])`
- `print(arr.ndim)  # Output: 2`

2. ndarray.shape

- **Description**: This attribute gives the shape of the array as a tuple. The shape represents the size of the array along each dimension (axis).
- **Example**:
- `arr = np.array([[1, 2, 3], [4, 5, 6]])`
- `print(arr.shape)  # Output: (2, 3)`

  Here, the array has 2 rows and 3 columns.

3. ndarray.size

- **Description**: This attribute returns the total number of elements in the array. It is equivalent to the product of the dimensions in the shape.
- **Example**:
- `arr = np.array([[1, 2], [3, 4], [5, 6]])`
- `print(arr.size)  # Output: 6`

4. ndarray.itemsize

- **Description**: This attribute returns the size (in bytes) of one element in the array. It gives the memory consumption per element.
- **Example**:
- `arr = np.array([1, 2, 3])`
- `print(arr.itemsize)  # Output: 8 (assuming a dtype of np.int64)`

5. ndarray.dtype

- **Description**: This attribute provides the data type (dtype) of the elements in the array. NumPy arrays can store elements of various types such as integers, floats, and more.
- **Example**:
- `arr = np.array([1.0, 2.0, 3.0])`

- `print(arr.dtype)  # Output: float64`

6. ndarray.T

- **Description**: This attribute gives the transpose of the array. Transposing swaps the rows and columns of the array.
- **Example**:
- `arr = np.array([[1, 2, 3], [4, 5, 6]])`
- `print(arr.T)`
- `# Output:`
- `# [[1 4]`
- `#  [2 5]`
- `#  [3 6]]`

8. b) Aggregation functions:

NumPy provides a wide range of **aggregation functions** that allow you to perform operations like summing, averaging, finding the minimum/maximum, and more, across arrays. These functions are very efficient and often used in data analysis and numerical computations. Below are the most commonly used aggregation functions in NumPy:

1. `np.sum()`

- **Description**: Computes the sum of array elements along a specified axis (or the entire array if no axis is specified).
- **Example**:
- `arr = np.array([1, 2, 3, 4])`
- `print(np.sum(arr))  # Output: 10`
- **Along an axis**:
- `arr = np.array([[1, 2], [3, 4]])`
- `print(np.sum(arr, axis=0))  # Output: [4 6]`
- `print(np.sum(arr, axis=1))  # Output: [3 7]`

2. `np.prod()`

- **Description**: Computes the product of array elements along a specified axis (or the entire array if no axis is specified).
- **Example**:
- `arr = np.array([1, 2, 3, 4])`
- `print(np.prod(arr))  # Output: 24`

3. `np.mean()`

- **Description**: Computes the arithmetic mean of array elements along a specified axis.
- **Example**:

- `arr = np.array([1, 2, 3, 4])`
- `print(np.mean(arr))  # Output: 2.5`
- **Along an axis**:
- `arr = np.array([[1, 2], [3, 4]])`
- `print(np.mean(arr, axis=0))  # Output: [2. 3.]`
- `print(np.mean(arr, axis=1))  # Output: [1.5 3.5]`

4. `np.median()`

- **Description**: Computes the median (the middle value) of the array along a specified axis.
- **Example**:
- `arr = np.array([1, 2, 3, 4, 5])`
- `print(np.median(arr))  # Output: 3.0`
- **Along an axis**:
- `arr = np.array([[1, 2, 3], [4, 5, 6]])`
- `print(np.median(arr, axis=0))  # Output: [2.5 3.5 4.5]`
- `print(np.median(arr, axis=1))  # Output: [2. 5.]`

5. `np.std()`

- **Description**: Computes the standard deviation (a measure of the spread or dispersion) of array elements along a specified axis.
- **Example**:
- `arr = np.array([1, 2, 3, 4, 5])`
- `print(np.std(arr))  # Output: 1.4142135623730951`

6. `np.var()`

- **Description**: Computes the variance (a measure of how far a set of numbers are spread out) of array elements along a specified axis.
- **Example**:
- `arr = np.array([1, 2, 3, 4, 5])`
- `print(np.var(arr))  # Output: 2.0`

7. `np.min()`

- **Description**: Finds the minimum value in the array, or along a specified axis.
- **Example**:
- `arr = np.array([1, 2, 3, 4])`
- `print(np.min(arr))  # Output: 1`
- **Along an axis**:
- `arr = np.array([[1, 2], [3, 4]])`
- `print(np.min(arr, axis=0))  # Output: [1 2]`
- `print(np.min(arr, axis=1))  # Output: [1 3]`

8. `np.max()`

- **Description**: Finds the maximum value in the array, or along a specified axis.
- **Example**:
- `arr = np.array([1, 2, 3, 4])`
- `print(np.max(arr))  # Output: 4`
- **Along an axis**:
- `arr = np.array([[1, 2], [3, 4]])`
- `print(np.max(arr, axis=0))  # Output: [3 4]`
- `print(np.max(arr, axis=1))  # Output: [2 4]`

9. a) Data visualization using matplotlib

#plotting
import numpy as np
import matplotlib.pyplot as plt
x=np.arange(0,3*np.pi,0.1)
print("x=",x)
y_sin=np.sin(x)
y_cos=np.cos(x)
plt.plot(x,y_sin)
plt.plot(x,y_cos)
plt.xlabel('x values')
plt.ylabel('y sine and cosine values')
plt.title('Sine and Cosine')
plt.legend(['Sine','Cosine'])
plt.show()
Output:

Bar Graph
##Bar Plot(for categorical variables)
import numpy as np
import matplotlib.pyplot as plt
counts=[979,120,12]
fuelType=('Petrol','Diesel','CNG')
index=np.arange(len(fuelType))
plt.bar(index,counts,color=['red','blue','cyan'])
plt.title('Bar plot of fuel types')
plt.xlabel('Fuel Types')
plt.ylabel('frequency')
plt.xticks(index,fuelType,rotation=0)
Plt.show()

Scatter plot
import matplotlib.pyplot as plt
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]

```
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
plt.scatter(x, y)
plt.show()
```

Histogram

```
import matplotlib.pyplot as plt
import numpy as np
# Generate random data for the histogram
data = np.random.randn(1000)
#print(data)
# Plotting a basic histogram
plt.hist(data, bins=30, color=&#39;skyblue&#39;, edgecolor=&#39;black&#39;)
# Adding labels and title
plt.xlabel(&#39;Values&#39;)
plt.ylabel(&#39;Frequency&#39;)
plt.title(&#39;Basic Histogram&#39;)
# Display the plot
plt.show()
```

9.b) Matplotlib and seaborn:

a) Matplotlib

Matplotlib is a widely used Python library for creating static, interactive, and animated visualizations. It provides a lot of flexibility in creating plots and charts and is often used alongside other libraries like NumPy and Pandas for data analysis and visualization.

Key Features:

- **Simple Syntax**: It is easy to use and integrates well with other libraries like Pandas and NumPy.
- **Customization**: Offers extensive customization options for plots (titles, labels, ticks, line styles, markers, etc.).
- **Support for Multiple Plots**: Can create a wide variety of plots such as line plots, scatter plots, bar charts, histograms, etc.
- **Interactivity**: Can create interactive plots in Jupyter notebooks or web applications.

Example: Creating a Simple Line Plot

```
import matplotlib.pyplot as plt

# Sample data
x = [0, 1, 2, 3, 4, 5]
y = [0, 1, 4, 9, 16, 25]

# Create a line plot
plt.plot(x, y)
```

```
# Add labels and title
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Simple Line Plot')

# Show the plot
plt.show()
```

Example: Creating a Bar Chart

```
import matplotlib.pyplot as plt

# Sample data
categories = ['A', 'B', 'C', 'D']
values = [3, 7, 2, 5]

# Create a bar chart
plt.bar(categories, values)

# Add labels and title
plt.xlabel('Category')
plt.ylabel('Value')
plt.title('Bar Chart Example')

# Show the plot
plt.show()
```

---

b) Seaborn

Seaborn is a Python data visualization library based on Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics. It simplifies the process of creating more complex plots, with a focus on statistical data visualization.

Key Features:

- **Ease of Use**: Provides simple syntax for creating complex statistical plots.
- **Integration with Pandas**: Works seamlessly with Pandas DataFrames, which makes it easy to visualize data.
- **Built-in Themes**: Includes several built-in themes for creating aesthetically pleasing plots.
- **Advanced Plot Types**: Includes advanced statistical plots like heatmaps, violin plots, pair plots, and more.

Example: Creating a Simple Scatter Plot with Regression Line

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
tips = sns.load_dataset('tips')

# Create a scatter plot with a regression line
sns.regplot(x='total_bill', y='tip', data=tips)
```

```
# Add title
plt.title('Scatter Plot with Regression Line')

# Show the plot
plt.show()
```

Example: Creating a Heatmap

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data (correlation matrix)
data = sns.load_dataset('flights')
pivot_data = data.pivot_table(index='month', columns='year',
values='passengers')

# Create a heatmap
sns.heatmap(pivot_data, annot=True, cmap='YlGnBu')

# Add title
plt.title('Heatmap of Flights Data')

# Show the plot
plt.show()
```

Example: Creating a Box Plot

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
tips = sns.load_dataset('tips')

# Create a box plot
sns.boxplot(x='day', y='total_bill', data=tips)

# Add title
plt.title('Box Plot of Total Bill by Day')

# Show the plot
plt.show()
```

Key Differences Between Matplotlib and Seaborn:

- **Ease of Use**: Seaborn provides a simpler interface for statistical plotting. It requires fewer lines of code to create complex plots like violin plots, heatmaps, and pair plots compared to Matplotlib.
- **Aesthetics**: Seaborn automatically applies better aesthetics and provides built-in themes, making it more visually appealing than Matplotlib by default.
- **Statistical Plots**: Seaborn includes several statistical plot types such as regression plots, pair plots, and categorical plots, which Matplotlib doesn't offer directly.

10. a) Graphs in seaborn:

1. Line Plot using Seaborn

A **Line Plot** is used to display the relationship between two continuous variables. The `sns.lineplot()` function is typically used for this purpose in Seaborn.

Program for Line Plot:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the built-in 'tips' dataset
tips = sns.load_dataset('tips')

# Create a line plot for total bill and tip
sns.lineplot(x='total_bill', y='tip', data=tips)

# Add title and labels
plt.title('Line Plot: Total Bill vs Tip')
plt.xlabel('Total Bill')
plt.ylabel('Tip')

# Display the plot
plt.show()
```

Explanation:

- `sns.lineplot()` creates the line plot by plotting `total_bill` on the x-axis and `tip` on the y-axis.
- The `tips` dataset comes preloaded with Seaborn.

---

2. Scatter Plot using Seaborn

A **Scatter Plot** is used to show the relationship between two continuous variables using individual data points. The `sns.scatterplot()` function is used to create scatter plots in Seaborn.

Program for Scatter Plot:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the built-in 'tips' dataset
tips = sns.load_dataset('tips')

# Create a scatter plot for total bill and tip
sns.scatterplot(x='total_bill', y='tip', data=tips, color='blue')

# Add title and labels
```

```
plt.title('Scatter Plot: Total Bill vs Tip')
plt.xlabel('Total Bill')
plt.ylabel('Tip')

# Display the plot
plt.show()
```

10.b) Multiple sub plots

In Python, you can create multiple subplots in a single figure using the **Matplotlib** library. Subplots allow you to arrange multiple plots in a grid layout within one figure. The `plt.subplots()` function is commonly used for this purpose.

Key Concepts:

- **`plt.subplots()`**: This function creates multiple subplots in a single figure. You can specify the number of rows and columns of subplots.
- **Axes**: The subplots are created on separate axes, and each axis can contain a different plot.
- **`figsize`**: You can adjust the overall size of the figure (which contains all the subplots) using this argument.

Syntax of `plt.subplots()`:

```
fig, axes = plt.subplots(nrows, ncols, figsize=(width, height))
```

- **`nrows`**: Number of rows of subplots.
- **`ncols`**: Number of columns of subplots.
- **`figsize`**: Optional argument to set the overall figure size (width, height).

Example: Multiple Subplots

Let's create a figure with multiple subplots and plot different kinds of charts.

```
import matplotlib.pyplot as plt
import numpy as np

# Create sample data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = x ** 2
y4 = np.log(x + 1)

# Create a figure with 2 rows and 2 columns of subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# First subplot: Line plot of sine function
axes[0, 0].plot(x, y1, 'r')
```

```
axes[0, 0].set_title('Sine Wave')

# Second subplot: Line plot of cosine function
axes[0, 1].plot(x, y2, 'g')
axes[0, 1].set_title('Cosine Wave')

# Third subplot: Line plot of x^2
axes[1, 0].plot(x, y3, 'b')
axes[1, 0].set_title('x^2')

# Fourth subplot: Line plot of logarithmic function
axes[1, 1].plot(x, y4, 'purple')
axes[1, 1].set_title('Logarithmic Function')

# Adjust layout to prevent overlapping subplots
plt.tight_layout()

# Show the plot
plt.show()
```

Explanation:

- **`plt.subplots(2, 2, figsize=(10, 8))`**: Creates a 2x2 grid of subplots (4 subplots in total) with a figure size of 10x8 inches.
- **`axes[0, 0]`**, **`axes[0, 1]`**, **`axes[1, 0]`**, **`axes[1, 1]`**: These represent individual subplots in a 2x2 grid. You can access each subplot using row and column indices.
- **`axes[row, col].plot()`**: Plots the data on the specified subplot.
- **`plt.tight_layout()`**: Adjusts the spacing between subplots to prevent overlap of labels or titles.