Lesson Plan:

Introduction to Python

PW SKILLS

Topics to be covered:

1. Overview of Programming Languages
2. Python vs. Other Programming Languages
3. What is Pseudocode?
4. Pseudocode Implementation
5. How to write Computer Code Logic?
6. Introduction to Python
7. Why should you learn Python programming?
8. Setting Up Python Environment
9. What can be done with Python?

1. Overview of Programming Languages:

- A programming language is a formal language for creating instructions for a computer to follow. It is a set of rules and symbols that define the process of writing a computer programme. Software programmes, websites, and embedded devices are all developed using programming languages.

- To learn a programming language, you need to understand the basics of syntax, data types, variables, operators, and control structures. You should also learn how to use functions and modules to organize your code. Once you have a good understanding of the basics, you can start to learn about more advanced topics, such as object-oriented programming and design patterns.

Analogy of Programming Languages:

- A programming language is like a set of special rules and symbols that you use to give instructions to a computer.
  - **Example:** Think of it like a recipe. Just like you follow a recipe to cook your favorite dish, a computer follows instructions in a programming language to perform tasks.

- You can use a programming language to create different things, such as:
  - Software programs (like computer games, word processors, or web browsers)
    Websites
  - Embedded devices (like the software inside your smartphone or smartwatch)
  - **Example:** Imagine you want to build a robot. You can use a programming language to tell the robot how to move, what to do when it senses an obstacle, and so on.

To learn a programming language, you start with the basics. You need to understand things like:

- **Syntax:** How to write the instructions in a way the computer understands. It's like following grammar rules in a language.
- **Algorithm:** An algorithm is a well-defined, step-by-step set of instructions or a systematic approach for solving a specific problem or accomplishing a particular task.
- **Data types:** A data type is a classification that specifies which type of value a variable can hold in computer programming i.e. Different kinds of information, like numbers, text, and more.
- **Variables:** A variable is a fundamental concept in computer programming and mathematics, which holds information. For example, a variable as a box that can store numbers, words, or other stuff.
- **Operators:** Operators are symbols or special keywords in programming that perform operations on data or variables or special symbols to do things like math or comparisons.
- **Control Structures:** Control structures are fundamental elements in computer programming that determine the flow of a program, to make decisions and repeat actions in your program(if, else, for, while loops).
- **Functions and modules:** A function is a self-contained and reusable block of code that performs a specific task or set of tasks. Modules are collections of related functions, variables, and other code elements that are bundled together for a specific purpose.

Example: Learning a programming language is a bit like learning a new sport. You start with the basic rules, like how to hold the ball and move your body. Then, you learn more advanced techniques, like strategies and teamwork.

There are many different programming languages, each with its own strengths and weaknesses. Some of the most popular programming languages include:

Python: A general-purpose language noted for its ease of use and readability. It is frequently used in web development, data science, and machine learning.

Java: A general-purpose programming language noted for its portability and security. It is frequently used in the creation of corporate applications and mobile apps.

C/C++: High-performance programming languages that are often used in operating systems, gaming, and embedded devices.

JavaScript: A programming language designed to add interactivity to web pages and to create online apps.

Go: A more recent language that is gaining popularity due to its ease of use and concurrency capabilities.

Programming languages can be classified into different types, based on their characteristics and how they are utilized. Some examples of common programming languages are:

- **General-purpose languages:** These languages can be used to develop a wide variety of software applications. Examples include Python, Java, and C++.
- **Special-purpose languages:** These languages are designed for specific tasks, such as web development, data science, or game development. Examples include JavaScript, R, and Unityscript.
- **Scripting languages:** These languages are used to automate tasks and to add interactivity to web pages. Examples include JavaScript, Python, and Bash.
- **Compiled languages:** These languages are converted into machine code before they can be executed by a computer. Examples include Java, C++, and Go.
- **Interpreted languages:** These languages are executed directly by the computer, without being converted into machine code first. Examples include Python, JavaScript, and PHP.

Learning a programming language might be difficult, but it can also be quite rewarding. Programming is a highly sought-after ability that may lead to a variety of career options. If you want to learn to code, there are several resources accessible to assist you.

2. Python vs. Other Programming Languages:

Python is a popular programming language known for its simplicity and readability. When comparing Python to other programming languages, there are several factors to consider:

2.1. Ease of Learning: Python is frequently suggested for beginners because of its simple syntax. Other languages, such as C++ or Java, may have more difficult learning curves.

2.2. Versatility: Python is a flexible programming language that is utilized in a variety of disciplines such as web development, data analysis, machine learning, and scientific computing. It is not confined to a particular application.

2.3. Readability: Python's code is simple to understand and maintain, making it ideal for collaborative projects.

2.4. Large Standard Library: Python has a comprehensive standard library that simplifies typical programming tasks and eliminates the need to develop code from scratch.

2.5. Community and Ecosystem: Python has a huge and active community, which means there is plenty of help, libraries, and frameworks accessible. This is true for other popular languages as well, but Python's environment is especially robust.

2.6. Cross-Platform: Python is cross-platform, which means you can execute your code on different operating systems without changing anything.

2.7. Scripting Language: Python is frequently used for scripting activities and automation due to its ease of use and rapid development capabilities.

2.8. Integration: Python can be easily integrated with other languages like C and C++ for performance-critical parts of an application.

2.9. Performance: Python is an interpreted language, which can make it slower than compiled languages like C++ in certain situations. However, Python's performance has been improving with the development of JIT (Just-In-Time) compilers like PyPy and libraries like NumPy and Cython for numeric computations.

2.10. Use Cases: Python is a great choice for web development (Django, Flask), data analysis (Pandas, NumPy), and machine learning (TensorFlow, PyTorch). For systems programming, C and Rust are more common. For mobile app development, languages like Swift (iOS) and Kotlin (Android) are preferred.

2.11. Concurrency and Parallelism: Python's Global Interpreter Lock (GIL) can limit its ability to fully utilize multi-core processors for parallel processing. Languages like Go or Rust provide better support for concurrency.

2.12. Security: Python's simplicity and readability can be advantageous for writing secure code. However, security depends more on the developer's practices and awareness.

2.13. Job Opportunities: Python has a strong job market, particularly in data science, web development, and machine learning. The demand for other languages may vary depending on the region and industry.

2.14. Legacy Systems: Some older systems and software rely on languages like COBOL or Fortran, so Python may not be a suitable choice for maintaining or integrating with such systems.

In conclusion, Python is a versatile and beginner-friendly language that excels in many areas, particularly data science and web development. However, the choice of a programming language should be based on the specific needs and goals of your project or application. Different languages have their own strengths and weaknesses, and the best choice depends on the context and requirements.

3. What is Pseudo code?

- Pseudocode is a term which is often used in programming and It is a methodology that allows the programmer to represent the implementation of an algorithm.
- It is a way of writing programs in which you represent the sequence of actions and instructions (aka algorithms) in a form that humans can easily understand.
- Pseudocodes are simple to interpret by everyone irrespective of their programming background.
- Pseudocode, as the name suggests, is a false code or a representation of code which can be understood by even a layman with some school level programming knowledge.

4. Pseudo code Implementation:

Example 1:

Code to find largest among two nos:

```
BEGIN
    NUM num1, num2
    DISPLAY "Enter num1 value: "
    INPUT num1
    DISPLAY "Enter num2 value: "
    INPUT num2

    IF num1 > num2 THEN
        DISPLAY "num1 is larger than num2"
    ELSE
        DISPLAY "num2 is larger than num1"
END
```

Example 2:

Find area of circle:

```
BEGIN
    NUM radius, area, PI
    PI = 22/7
    DISPLAY "ENTER THE RADIUS OF CIRCLE : "
    INPUT radius
    area = PI*radius*radius
    DISPLAY "AREA OF CIRCLE : " area
END
```

5. How to write Computer Code Logic?

Practice writing a lot of code:
Start reading lots of open source code and try to make contributions to open source code.

Check solutions by other people:
It's a good programming practice to read code written by others, it will help you in understanding different approaches for the same problem.

Use a pen and paper to work out solutions:
It is always a good practice to write down the logic of the code on pen and paper, It will help you build strong logic for your programs.

Keep learning new things:
Whenever you find something new just code it out, that's the best way to learn programming.

Be consistent:
Daily Assign some amount of time apart from your regular for just programming. This will keep you up-to-date.

Face problems head-on:
If you have problems early on while learning it's good because it will lay a solid foundation for programming.

6. Introduction to Python:

- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.
- It was created by Guido van Rossum and first released in 1991.
- It has the following features that makes it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together:
    - High-level built in data structures
    - Dynamic typing
    - Dynamic binding
- Python's simple, easy to learn syntax emphasizes on code readability with the use of whitespaces and therefore reduces the cost of program maintenance.
- Python supports modules and packages, which encourages program modularity and code reuse.
- The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.
- Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing.

7. Why should you learn Python programming?

Among numerous languages available in the market why should you choose python? This is the first question that arises in the mind of new users.

Following are the some of the reasons why people select python:

- **Quality of software:** Python was meant for readability. It's reusable and maintainable as compared to other languages. It's easier to understand. It supports all the modern features like OOPs and functional programming.

- **Productivity of Developers:** The same program which is written in other high-level languages like c++ or java can be written in one-third or one-fifth line of code. That means debugging can be easy and it will be less prone to error which in turn increases the productivity of the developers.

- **Portability:** Mostly it's platform-independent. It can run on any platform or OS with minor or no change at all which makes it a highly portable language. Now you can use Micro Python to interact with hardware as well. It can be used on most of the edge devices.

- **Supporting Libraries:** Python already has a lot of inbuilt libraries that come with the standard python package which you download from its official site. With these libraries, you can build lots of basic applications or day to day automation tasks like copying data in bulk from one place to another. Apart from this, there's a huge list of third-party libraries like Numpy, Matplotlib, Scikit Learn, etc.

- **Fun to use:** Its simplicity and availability of lots of supporting libraries plus huge open source community support make development in python a breeze. That's why its widely preferred by hobbyists as well.

8. What can be done with Python?

- System Programming
- Graphical User Interface
- Web Scraping
- Managing Database
- Fast Prototyping
- Numeric / Scientific Programming
- Game development
- Image Processing
- Robotics
- Automation
- Data science
- Data Mining

Lesson Plan:

Predefined Objects in Python

PW SKILLS

There are several predefined objects and data types in Python. These objects are readily available for use without the need for any additional imports. Here are some of the most common predefined objects in Python

1. **Numeric Types:**
- int: Integer numbers **(e.g., 42, -10).**
- float: Floating-point numbers **(e.g., 3.14, -0.5).**
- complex: Complex numbers with real and imaginary parts **(e.g., 2 + 3j).**

2. **Sequence Types:**
- str: Strings of characters **(e.g., "hello", 'world').**
- list: Mutable sequences of elements **(e.g., [1, 2, 3])**.
- tuple: Immutable sequences of elements **(e.g., (1, 2, 3))**.

3. **Mapping Types:**
- dict: Mutable collections of key-value pairs **(e.g., {'a': 1, 'b': 2}).**

4. **Set Types:**
- set: Mutable collections of unique elements **(e.g., {1, 2, 3}).**
- frozenset: Immutable collections of unique elements **(e.g., frozenset({1, 2, 3})).**

5. **Boolean Type:**
- bool: Boolean values representing True or False.

6. **NoneType:**
- None: A special object representing the absence of a value or a null value.

In the above you came across these two terminologies: **Mutable** and **Immutable**. Let's understand what they really mean:

Mutable	Immutable
Objects whose state can be modified after creation are called mutable objects.	Objects whose state cannot be modified after creation are called immutable objects.
Lists, dictionaries, sets, and user-defined classes (unless explicitly designed to be immutable) are examples of mutable objects.	Examples include integers, floating-point numbers, strings, tuples, and frozensets.
Mutable objects can be altered in place; that is, you can change their contents without creating a new object.	Immutable objects cannot be changed once they are created. Any operation that appears to modify an immutable object actually creates a new object.

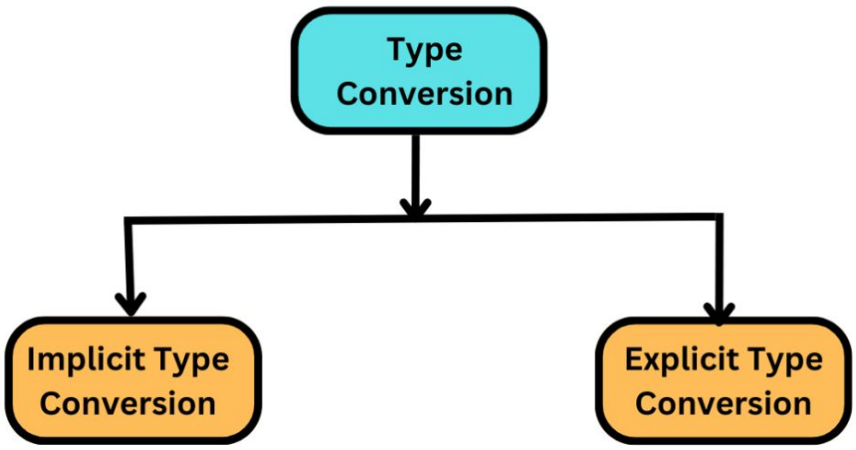
Lesson Plan:

Python Type Conversion

PW SKILLS

Topics to be covered:

1. Type Conversion
2. Implicit Type Conversion
3. Explicit Type Conversion
4. Type Conversion Between Strings and Other Types



1. Type Conversion:

- The process of changing the data type of a value or object in Python is known as type conversion or type casting.
- It allows you to convert a value from one data type to another, which is useful when executing operations that require several data types or when checking data compatibility.
- Python includes built-in type conversion methods and strategies.

Here's how Python type conversion works:

2. Implicit Type Conversion:

- Python automatically executes type conversion to ensure that operations are compatible. This is referred to as implicit type conversion.
- For example, if you add an integer and a float, Python will implicitly convert the integer to a float first.

Example:

```
x = 5       # integer
y = 2.5     # float
result = x + y   # Python implicitly converts x to a float
before addition
```

Why Implicit Type Conversion ?

Implicit type conversion, also known as type coercion, is needed in programming to allow for operations involving different data types without causing errors. It's a feature that automatically converts data from one type to another when necessary to perform an operation. Here's why it's needed with an example:

1. Compatibility: Different data types have different rules for how they can interact with one another. Implicit type conversion helps ensure that operations are compatible and can be carried out smoothly, even when the data types are not an exact match.

2. Convenience: Implicit type conversion simplifies coding by allowing developers to write more concise and readable code. It reduces the need for explicit type casting and conversion functions in many cases.

Analogy of Implicit Type Conversion:

An analogy for implicit type conversion is like a universal remote control.

- In a household, you have various devices like the TV, stereo, and lights. Each device may have its unique remote control with specific buttons and functions tailored for that device. However, it can be inconvenient to have separate remotes for everything.

- The universal remote, in this analogy, is similar to implicit type conversion. It's a single remote that can adapt to work with different devices. When you press a button to change the TV's channel, it sends the appropriate signals for the TV. When you adjust the stereo's volume, it adjusts the sound. The universal remote "implicitly" figures out which device you're controlling and adapts to work with it, saving you from the hassle of having a separate remote for each device.

- Similarly, in programming, implicit type conversion allows you to perform operations with different data types as if they were the same type. The system "implicitly" converts one or both of them to a compatible type for the operation, making your code more flexible and convenient.

3. Explicit Type Conversion:

- Explicit type conversion is the process of modifying the data type of a value or object using built-in functions.

- Common functions for explicit type conversion include:
  - **int():** Converts a value to an integer.
  - **float():** Converts a value to a float.
  - **str():** Converts a value to a string.
  - **bool():** Converts a value to a boolean.
- These functions are invoked by providing a value or variable as an argument.

Example:

```
x = 22.5
y = int(x)     # Explicitly converts the float to an integer
```

Why Implicit Type Conversion ?

Explicit type conversion, also known as type casting, is needed in programming to change the data type of a value or variable from one type to another. It's necessary when you want to perform operations that require data of a specific type or when you need to ensure data consistency in your code.

Analogy of Implicit Type Conversion:

Explicit type conversion like using a translator or interpreter when you're communicating with someone who speaks a different language. Here's an analogy:

Imagine you're talking to a friend who only speaks French, but you want to share a message in English. In this scenario:

1. Your Friend (Data Type): Your friend represents a specific data type, like a string or an integer.

2. Your Message (Value): Your message is the value you want to use in your code, which might be in a different data type than your friend understands.

3. Translator (Explicit Type Conversion): To make your message understandable, you use a translator, who translates your English message into French. The translator acts as explicit type conversion, ensuring your message (value) matches the data type your friend (data type) can comprehend.

So, in programming, explicit type conversion serves as the translator, helping you communicate or perform operations with data of different types in a way that makes sense and doesn't result in errors.

4. Type Conversion Between Strings and Other Types:

- Using explicit type conversion, you can convert strings to other data types such as integers and floats.

For example, to convert a string representing a number to an actual number, you can use int() or float():

```
num_str = "33"
num_int = int(num_str)
num_float = float(num_str)
```

To convert numbers to strings, you can use str():

```
num = 33
num_str = str(num)
```

5. Type Conversion in Data Structures:

- When working with data structures containing elements of different data types, such as lists and tuples, type conversion may be required.

- To change the data type of individual items in a data structure, use explicit type conversion.

Example:

```
mixed_list = [1, "two", 3.0]
int_value = int(mixed_list[0])    # Converts the first
element to an integer
str_value = str(mixed_list[1])    # Converts the second
element to a string
```

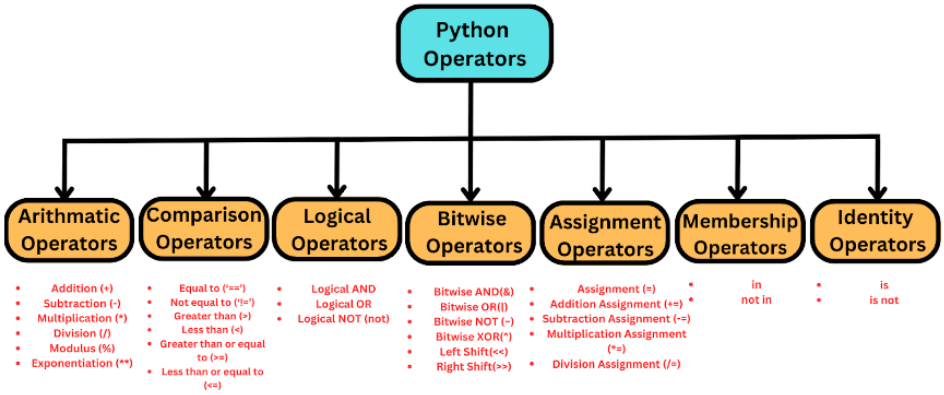
Lesson Plan:

Python Operators

PW SKILLS

Topics to be covered:

1. Python Operators
2. Operators Types
3. Operators Precedence
4. Operators associativity



1. Python Operators:

- Python operators are symbols or special keywords that are used to perform operations on values or variables.

- In Python programming, Operators allow you to manage data, do computations, and make decisions.

2. Operators Types:

There are several types of python operators as follows:

2.1. Arithmetic Operators: Arithmetic operators are used for mathematical operations.

Addition (+): Combining two or more values.

```python
add  = 10 + 5   # add is now 15
```

Subtraction (-): Finding the difference between two values.

```python
sub = 20 - 8  # sub is now 12
```

Multiplication (*): Multiplying values.

```python
mult = 3 * 4  # mult is now 12
```

Division (/): Dividing values.

```python
divide = 24 / 3  # PhysicsWallah is now 8.0
```

Modulus (%): Finding the remainder of a division.

```python
remainder = 17 % 4  # remainder is 1
```

Exponentiation (**): Raising a value to a power.

```python
squared = 5 ** 2  # squared is 25
```

2.2. Comparison Operators:

Comparison operators are used to compare two values and return a Boolean result (True or False).

Equal to ('=='): Checks if two values are equal.

```python
result = 2 == 2  # result is True
```

Not equal to ('!='): Checks if two values are equal.

```python


result = 3 != 2  # result is True
```

Greater than (>): Checks if one value is greater than another.

```python
result = 10 > 9  # result is True
```

Less than (<): Checks if one value is less than another.

```python
result = 10 < 9  # result is False
```

Greater than or equal to (>=): Checks if one value is greater than or equal to another.

```python
result = PhysicsWallah >= PWskills # result is True
```

Less than or equal to (<=): Checks if one value is less than or equal to another.

```python
result = PhysicsWallah <= PWskills # result is False
```

2.3. Logical Operators: Logical operators are used to combine and manipulate Boolean values.

Logical AND (and): Returns True if both conditions are True.

```python
result = (7 > 5) and (8 < 15)  # result is True
```

Logical OR (or): Returns True if at least one condition is True

```python
result = (7 > 5) or (20 < 15)  # result is True
```

Logical NOT (not): Negates a condition (True becomes False, and False becomes True).

```python
result = not (PWskills == Course)  # result is True
```

2.4. Bitwise Operators: In Python, bitwise operators are used to perform operations at the bit-level, manipulating individual bits within integers.

Bitwise AND (&): Returns 1 for each bit position where both operands have a 1.

AND Operation Table

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```python
a = 5   # 101 in binary
b = 3   # 011 in binary

result = a & b # Bitwise AND

# Binary result: 001, which is 1 in decimal
print("a & b =", result)

#output: a & b = 1
```

Bitwise OR (|): Returns 1 for each bit position where at least one of the operands has a 1.

OR Operation Table

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Example:

```python
a = 5 # 101 in binary
b = 3 # 011 in binary

result = a | b # Bitwise OR
# Binary result: 111, which is 7 in decimal
print("a | b =", result)

#output: a | b = 7
```

Bitwise NOT (~): Inverts the bits; 1s become 0s and vice versa.

NOT Operation Table

a	~a
0	1
1	0

Example:

```python

a = 5 # 101 in binary

result = ~a # Bitwise NOT
# Binary result: 11111010, which is -6 in two's complement
print("~a =", result)

output: ~a = -6
```

Bitwise XOR (^): Returns 1 for each bit position where exactly one of the operands has a 1.

XOR Operation Table

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Example:

```python
a = 5 # 101 in binary
b = 3 # 011 in binary

result = a ^ b # Bitwise XOR

# Binary result: 110, which is 6 in decimal
print("a ^ b =", result)

output: a ^ b = 6
```

Left Shift (<<): Shifts the bits to the left by a specified number of positions, filling with 0s on the right.

```python
a = 5 # 101 in binary

result = a << 2 # Left shift by 2 positions

# Binary result: 10100, which is 20 in decimal
print("a << 2 =", result)

output: a << 2 = 20
```

Right Shift (>>): Shifts the bits to the right by a specified number of positions, filling with 0s on the left (for non-negative numbers) or sign bits (for negative numbers).

```python
a = 16 # 10000 in binary

result = a >> 2 # Right shift by 2 positions
# Binary result: 100, which is 4 in decimal
print("a >> 2 =", result)

output: a >> 2 = 4
```

2.5. Assignment Operators: Assignment operators are used to assign values to variables.

Assignment (=): Assigns a value to a variable.

```python
course = 20
```

Addition Assignment (+=): Adds a value to the variable and updates it.

```python
Course += 5  # Course is now 25
```

Subtraction Assignment (-=): Subtracts a value from the variable and updates it.

```python
Course -= 2  # Course is now 23
```

Multiplication Assignment (*=): Multiplies the variable by a value and updates it.

```python
Course *= 2  # Course is now 46
```

Division Assignment (/=): Divides the variable by a value and updates it.

```python
Course /= 4  # Course is now 11
```

2.6. Membership Operators: Membership operators are used to test if a value is present in a sequence.

in: Checks if a value is present in a sequence (e.g., a string, list, or tuple).

```python
result = "skills" in "PWskills"  # result is True
```

not in: Checks if a value is not present in a sequence.

```python
result = "DataScience" not in "PWskills"  # result is True
```

2.7. Identity Operators: Identity operators are used to compare the memory location of two objects.

is: Returns True if both variables point to the same object.

```python
x = "PWskills"
y = x
result = x is y  # result is True
```

is not: Returns True if both variables point to different objects..

```python
x = "PWskills"
y = "LPU"
result = x is not y  # result is True
```

3. Operator's Precedence:

- Python operators have different levels of precedence, which determine the order in which they are evaluated in an expression. Higher precedence operators are evaluated first.

- Here's a simplified Python operator precedence table, arranged from highest precedence to lowest precedence:

Operators	Meaning
()	Parentheses
**	Exponent
+X, -X, ~X (X is variable)	Unary Plus, Unary Minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor Division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise Shift Operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

1. Parentheses():

Example:

```python
# Example 1: Using parentheses to change order of operations

result = (2 + 3) * 4  # Parentheses ensure addition is done first
print(result)  # Output: 20

# Example 2: Grouping expressions for clarity

total = (10 + 5) + (8 / 2)  # Using parentheses for clarity
print(total)  # Output: 19.0
```

2. Exponentiation' ** ': This operator raises a number to a power.

Example:

```python
# Example 1: Calculating 2 to the power of 3

result = 2 ** 3
print(result)  # Output: 8

# Example 2: Calculating 5 squared
squared = 5 ** 2
print(squared)  # Output: 25
```

3. Unary Plus +X, Unary Minus -X, Bitwise NOT ~X (X is variable):

Example:

```python
# Example 1: Calculating 2 to the power of 3

result = 2 ** 3
print(result)  # Output: 8

# Example 2: Calculating 5 squared
squared = 5 ** 2
print(squared)  # Output: 25
```

4. Multiplication '*', Division '/', Floor Division '//', Modulus '%':

Example:

```python
# Example 1: Multiplication

result = 3 * 4
print(result)  # Output: 12

# Example 2: Division and Modulus

quotient = 15 / 4
remainder = 15 % 4
print(quotient, remainder)  # Output: 3.75 3
```

5. Addition +, Subtraction :

Example:

```python
# Example 1: Addition
sum = 7 + 3
print(sum)  # Output: 10

# Example 2: Subtraction
difference = 12 - 5
print(difference)  # Output: 7
```

6. Bitwise Shift Operators <<, >>:

Example:

```python
# Example 1: Left Shift

result = 8 << 2  # Shift 8 two bits to the left
print(result)  # Output: 32

# Example 2: Right Shift

value = 32
shifted = value >> 2  # Shift 32 two bits to the right
print(shifted)  # Output: 8
```

7. Bitwise AND '&', Bitwise XOR '^', Bitwise OR '|':

Example:

```python
# Declaring variables
a = 10 # Binary representation: 1010
b =6   # Binary representation: 0110

# Example 1: Bitwise AND(&)

result = a & b  # Bitwise AND
print(result)  # Output: 2 Since, 0010 is the result after
applying bitwise AND, its decimal representation is 2

# Example 2: Bitwise XOR(^)

result = a ^ b # Bitwise XOR
print(result)  # Output: 12 Since, 1100 is the result after
applying Bitwise XOR, its decimal representation is 12

# Example 3: Bitwise OR(|)
result = a | b # Bitwise OR
print(result)  # Output: 14 Since, 1110 is the result after
applying Bitwise OR, its decimal representation is 14
```

8. Comparisons, Identity, Membership Operators:

Example:

```python

# Example 1: Comparison Operator (==)

result = 5 == 5
print(result)  # Output: True

# Example 2: Identity Operator (is)

a = [1, 2, 3]
b = a
result = a is b
print(result)  # Output: True

# Example 3: Membership Operator (in)

PWskills = ["DataScience", "DataAnalytics", "DataEngineering"]
result = "DataScience" in PWskills
print(result)  # Output: True
```

9. Logical NOT ‘not’, Logical AND ‘and’, Logical OR ‘or’:

Example:

```python
# Example 1: Logical NOT
result = not True
print(result)  # Output: False

# Example 2: Logical AND
result = True and False
print(result)  # Output: False

# Example 3: Logical OR
result = True or False
print(result)  # Output: True
```

4. Operators Associativity:

Operator associativity in Python is an important concept that determines how operators are grouped and evaluated in expressions. Here's a concise explanation in four bullet points:

1. Order of Evaluation: Operator associativity defines the order in which operators of the same precedence are evaluated within an expression. It specifies whether operators are evaluated from left to right or right to left.

2. Left-to-Right Default: By default, most operators in Python have left-to-right associativity, which means they are evaluated from left to right when they appear in sequence in an expression.

3. Right-to-Left Associativity: Some operators, like the exponentiation operator `**`, have right-to-left associativity, meaning they are evaluated from right to left. This can affect the order of operations in expressions with multiple operators of the same precedence.

4. Control with Parentheses: If you need to override the default associativity and explicitly control the order of evaluation, you can use parentheses `()` to group operations. This allows you to ensure that specific operations are performed before others.

Python Operator's Associativity:

Operator	Associativity	Example	First Evaluation
Parentheses '()'	Left to Right	(3 - 2) * 4	3 - 2
Exponentiation '**'	Right to Left	2 ** 3 ** 2	3 ** 2
Unary Plus +X	Left to Right	+5 + 3	+5
Unary Minus -X	Left to Right	-8 - 2	-8
Bitwise NOT ~X	Left to Right	~10 & 7	~10
Multiplication '*'	Left to Right	2 * 3 * 4	2 * 3
Division '/'	Left to Right	10 / 2 / 5	10 / 2
Floor Division '//'	Left to Right	11 // 3 // 2	11 // 3
Modulus '%'	Left to Right	15 % 4 % 2	15 % 4
Addition '+'	Left to Right	7 + 3 - 1	7 + 3
Subtraction '-	Left to Right	12 - 5 - 2	12 - 5
Bitwise Shift Operators	Left to Right	8 << 2 >> 1	8 << 2
Bitwise AND &	Left to Right	10 & 7 & 5	10 & 7
Bitwise XOR ^	Left to Right	15 ^ 9 ^ 3	15 ^ 9
Bitwise OR |	Left to Right	15 | 9 | 3	15 | 9
Comparison Operators	Left to Right	5 == 5 != 3	5 == 5
Identity Operators	Left to Right	a is b is not c	a is b
Membership Operators	Left to Right	"DataScience" in PWskills	"DataScience" in PWskills
Logical NOT 'not'	Left to Right	not True and False	not True
Logical AND 'and'	Left to Right	True and False or True	True and False
Logical OR 'or'	Left to Right	True or False or True	True or False

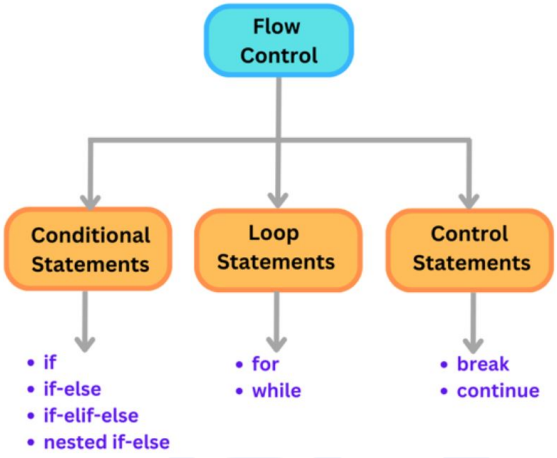
Lesson Plan:

Conditions (If Else, If-Elif-Else)

PW SKILLS

Topics to be covered:

1. Conditional Statements
2. If statements
3. If-else statements
4. if-elif-else statements
5. Nested if-else statements
6. Analogy of Conditional Statements



1. Conditional Statements:

- Conditional statements, often known as control structures, are an essential component of programming. They enable you to make code decisions based on predefined conditions.

- The primary conditional statements in Python are if, elif (short for "else if"), and else.

Purpose of Conditional Statements:

The purpose of conditional statements in programming is to enable the execution of different code blocks based on specific conditions. Here are four key purposes of conditional statements:

1. Decision Making: Conditional statements allow a program to make decisions by evaluating conditions. They help determine which set of instructions to execute depending on whether the conditions are met or not.

2. Control Flow: Conditional statements control the flow of a program, ensuring that the code follows a specific path or branches based on the given conditions. This helps create responsive and adaptable programs.

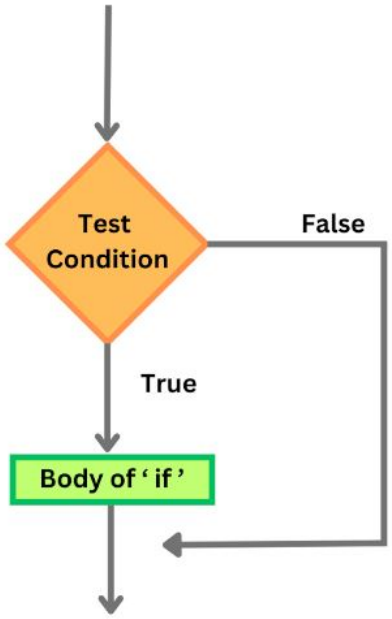
3. Error Handling: Conditional statements are used for error handling. They can identify and respond to exceptional situations, preventing program crashes or unexpected behavior by executing specific error-handling code.

4. Customization: Conditional statements provide the means to customize the behavior of a program for different scenarios or inputs. They allow programmers to tailor the program's response to various user interactions or data inputs, making the software more versatile and user-friendly.

2. 'if' statements:

- The if statement is the most basic type of control statement. It accepts a condition and determines whether it is True or False.

- If the condition is True, the True piece of code is run; otherwise, the block of code is bypassed, and the controller proceeds to the next line.



Examples:

```
x = 10

if x > 5:
    print("x is greater than 5")
```

```
temperature = 30

if temperature > 25:
    print("It's a hot day")
```

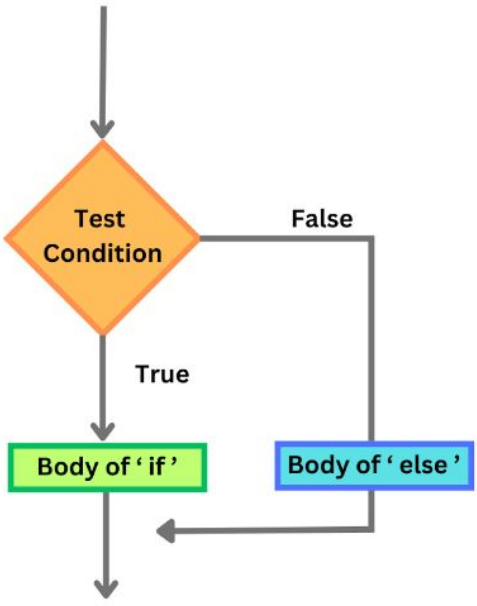
```
is_datascience_course = True

if is_datascience_course:
    print("Don't forget to join PWskills")
```

```
age = 18
if age ⩾ 18:
    print("You are an adult.")
else:
    print("You are not an adult.")
```

```
score = 75
passing_score = 70
if score ⩾ passing_score:
    print("Congratulations, you passed!")
else:
    if score ⩾ passing_score - 5:
        print("You almost passed.")
    else:
        print("You didn't pass.")
```

3. 'if-else' statements:

- The if-else statement checks the condition and executes the if block of code when the condition is True, and if the condition is False, it will execute the else block of code.



Examples:

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

```
age = 20
if age ⩾ 18:
    print("You are eligible to vote")
else:
    print("You are not eligible to vote")
```

```
is_datascience_course = True

if is_datascience_course:
    print("Don't forget to join PWskills")
else:
    print("ASAP this course will be there")
```

```
num = 7
if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

```
score = 85
result = "Pass" if score ⩾ 70 else "Fail"
print(f"You {result}.")
```

4. 'if-elif-else' statements:

- The if-elif-else condition statement in Python uses elif blocks to link multiple conditions one after the other. This is useful when you need to check numerous conditions at the same time.

- We can make a difficult decision with the help of if-elif-else. The elif statement checks each condition one by one and executes the code if the condition is met.

Examples:

```
x = 10
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
```

```
score = 75
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
elif score >= 70:
    print("C")
```

```
hour = 14

if hour < 12:
    print("Good morning")
elif hour < 17:
    print("Good afternoon")
else:
    print("Good evening")
```

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

```
age = 30
if age < 18:
    print("You are a minor.")
elif 18 <= age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

5. 'Nested if-else' statements:

- In Python, a nested 'if-else' statement is a construct in which one 'if' statement is contained within another 'if' or 'else' block.
- This enables you to design a condition hierarchy in which the inner 'if-else' statements are evaluated only if the outer condition is true.
- When you need to test many circumstances in an organized manner, nested if-else statements are useful.

Examples:

```
x = 10
y = 5

if x > 5:
    if y > 5:
        print("Both x and y are greater than 5.")
    else:
        print("x is greater than 5, but y is not.")
else:
    print("x is not greater than 5.")
```

```
is_weekend = False
is_sunny = True

if is_weekend:
    if is_sunny:
        print("Go for a picnic.")
    else:
        print("Stay in and relax.")
else:
    print("It's a workday.")
```

```
is_student = True
is_teacher = False

if is_student:
    if is_teacher:
        print("You are both a student and a teacher.")
    else:
        print("You are a student but not a teacher.")
else:
    if is_teacher:
```

```
        print("You are a teacher but not a student.")
    else:
        print("You are neither a student nor a teacher.")
```

```
is_vip = True
age = 30

if is_vip:
    if age >= 18:
        if age < 65:
            print("Welcome, VIP customer!")
        else:
            print("You're a VIP, but you qualify for senior
discounts.")
    else:
        print("VIP status is for adults only.")
else:
    print("Regular pricing applies.")
```

Analogy of Conditional Statements:

- Control statements, using `if`, `if-elif-else`, and `if-else`, can be compared to making decisions in everyday life:

Using `if` Statements:

- Imagine you're deciding whether to go outside to play. If the weather is sunny, you'll go outside; otherwise, you'll stay indoors.
    - If it's sunny (condition met), you go outside.
    - If it's not sunny (condition not met), you stay indoors.

Using `if-elif-else` Statements:

- Now, think of planning a day at an amusement park. You have different age groups with different ticket prices: children, adults, and seniors.
    - If you are a child, you pay the child's price.
    - If you are an adult, you pay the adult's price.
    - If you are a senior, you pay the senior's price.

Using `if-else` Statements:

- Consider a scenario where you're checking whether a fruit is ripe before eating it. If it's ripe, you eat it; otherwise, you leave it for later.
    - If the fruit is ripe (condition met), you eat it.
    - If the fruit is not ripe (condition not met), you don't eat it.
- In all these analogies, the `if` statements determine the course of action based on a single condition. The `if-elif-else` statements help you make choices from a range of options, and the `if-else` statements provide a simple binary choice.

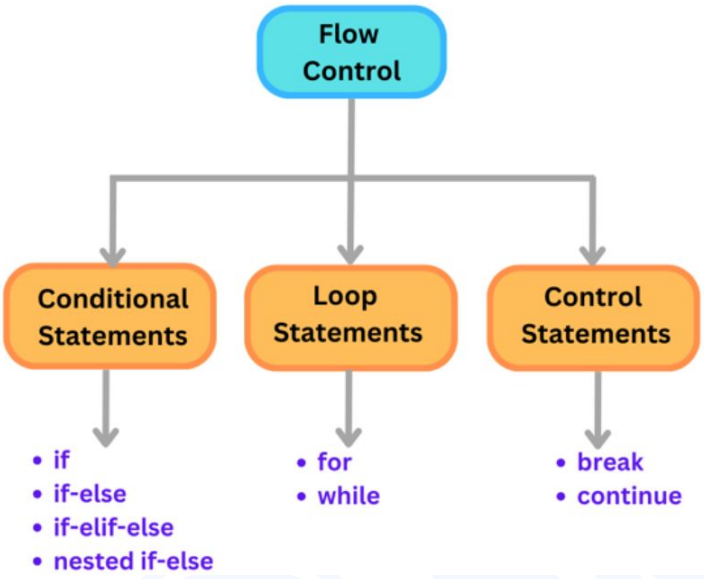
Lesson Plan:

Loops (While, For)

PW SKILLS

Topics to be covered:

1. Loop Statements
2. 'for' loop
3. Nested 'for' loop
4. 'While' loop



1. Loop Statements:

- Loop statements, often referred to as "loops," are a fundamental part of programming that allow you to repeatedly execute a block of code based on a specified condition.

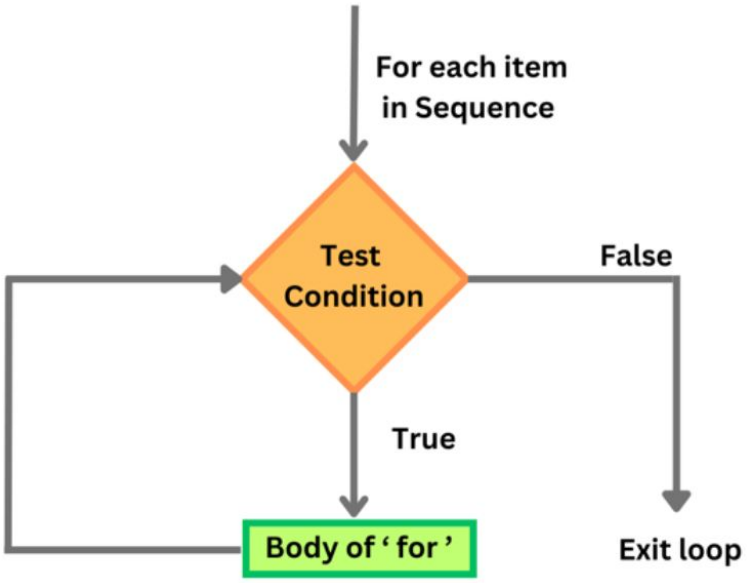
- Loops are used to automate repetitive tasks and iterate through data structures, making your code more efficient and less repetitive.

In Python, there are two main types of loop statements: for loops and while loops.

2. 'For' Loops:

- The `for` loop is used to iterate over a sequence of elements, such as a list or a string. It executes a block of code for each item in the sequence.

- The loop variable takes the value of each item in the sequence one by one. This loop is helpful when you know the number of iterations in advance or when you want to process each element of a collection.



Examples:

Iterating through a List:

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

for fruit in fruits:
    print("I like", fruit)
```

Generating a Number Sequence:

```
for i in range(1, 6):
    print("Number:", i)
```

3. Nested ‘for’ Loops:

- In Python, a nested for loop is one loop inside another loop. This is frequently used for activities where you have to repeatedly operate on values that fall within a certain range or iterate over elements in a grid or two-dimensional list.

A nested for loop is demonstrated here:

A nested for loop is demonstrated here:

Examples:

Pattern 1: Right Triangle

```
for i in range(5):
    for j in range(i + 1):
        print("*", end=" ")
    print()
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Pattern 2: Square

```
for i in range(4):
    for j in range(4):
        print("*", end=" ")
    print()
```

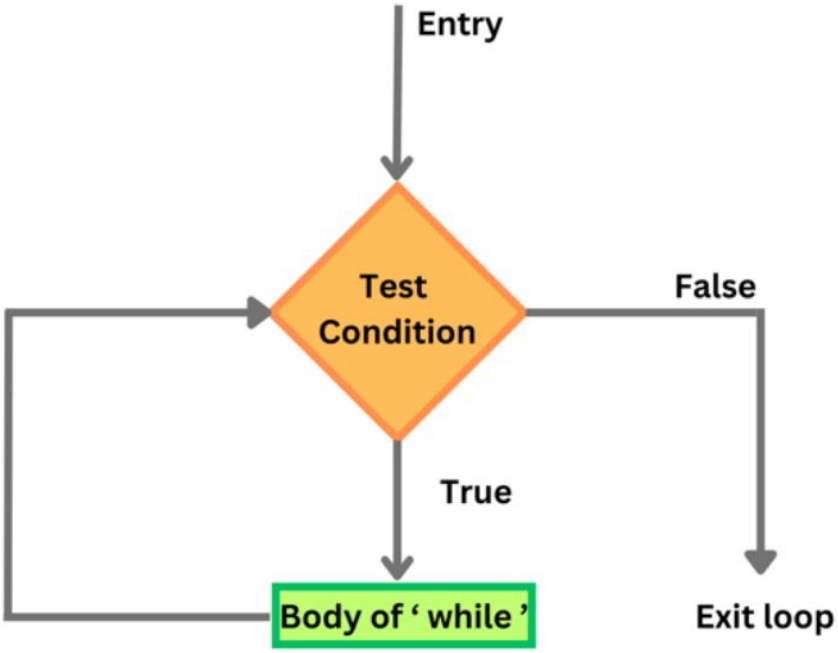
Output:

```
* * * *
* * * *
* * * *
* * * *
```

4. 'while' Loops:

- The `while` loop repeatedly executes a block of code as long as a given condition is true. The condition is checked before each iteration, and if it becomes false, the loop is exited.

- This allows for repetitive execution until a certain condition is met. Care must be taken to avoid infinite loops by ensuring the condition eventually becomes false.



Examples:

Counting Down:

```
count = 5

while count > 0:
    print(count)
    count -= 1
```

```
valid_input = False

while not valid_input:
    user_input = input("Enter 'yes' or 'no': ")
    if user_input.lower() in ["yes", "no"]:
        valid_input = True
    else:
        print("Invalid input. Please enter 'yes' or 'no'.")
```