Lecture 1: Introduction to Python language

What is a Language?

What is a Programming language

1. A **programming language** is a formal system of instructions used to communicate with a computer.
2. Enables developers to write programs for various tasks such as web development, data analysis, and automation.
3. **Examples**: C, Java, Python, JavaScript.

Why Python

1. **Ease of Use**: Simple syntax, ideal for beginners.
2. **Versatility**: Supports multiple programming paradigms (object-oriented, procedural, functional).
3. **Extensive Libraries**: Rich ecosystem for tasks like data analysis, machine learning, and web development.
4. **Community Support**: Large and active community for troubleshooting and learning.
5. **Interpreted Language**: Python is executed line by line, which makes debugging easier.
6. **Cross-Platform**: Python programs can run on various platforms without modification (Windows, macOS, Linux).
7. **Scalability:** While often seen as a scripting language, Python scales well for larger applications when combined with appropriate frameworks or tools (e.g., Django)

What is Python

1. **Python** is a high-level, interpreted programming language known for its simplicity and readability. It was created by **Guido van Rossum** in the late 1980s and first released in **1991**. Python emphasizes **code readability** and supports multiple programming paradigms, including **object-oriented**, **procedural**, and **functional programming**.
    a. Monty Python’s Flying Circus
2. Older than Java (1995).

Where Python is Used?

1. **Web Development**: Frameworks: Django, Flask.
2. **Data Science and Machine Learning**: Predictive modeling and data analysis using NumPy, Pandas, TensorFlow, scikit-learn.

3. **Automation (Scripting)**: Automating repetitive tasks.
4. **Game Development**: Framework: PyGame.
5. **Scientific Computing**: Libraries: SciPy, Matplotlib.
6. **Artificial Intelligence (AI) and Deep Learning**: Libraries: Keras, PyTorch.
7. **Web Scraping**: Libraries: BeautifulSoup, Scrapy.
8. **Cybersecurity**: Used for creating tools to test system vulnerabilities.

Companies using Python: Google**,** Netflix, Instagram, Spotify, Dropbox, NASA, Uber, Reddit, IBM, 10000 coders website

Some disadvantages of Python:

1. Slower Execution Speed
   - Interpreted language, making it slower than compiled languages like C++ or Java.
   - Not ideal for performance-critical applications (e.g., real-time systems, gaming engines).
2. Memory Consumption
   - Python's dynamic typing and garbage collection can lead to higher memory usage.
   - Not suitable for applications where memory optimization is crucial.
3. Global Interpreter Lock (GIL)
   - The GIL restricts the execution of multiple threads at the same time.
   - This makes Python less efficient for multithreaded applications, especially on multi-core processors.
4. Runtime Errors
   - Python's dynamic nature allows runtime errors to occur if code is not thoroughly tested.
   - This can lead to bugs in production if not handled properly.
5. Limited Support for Low-Level Programming
   - Languages like C or Rust are better suited for such tasks.

Python Installation:

1. Download link: Download Python | Python.org
2. Commands to check installations:
   a. In cmd: python --version

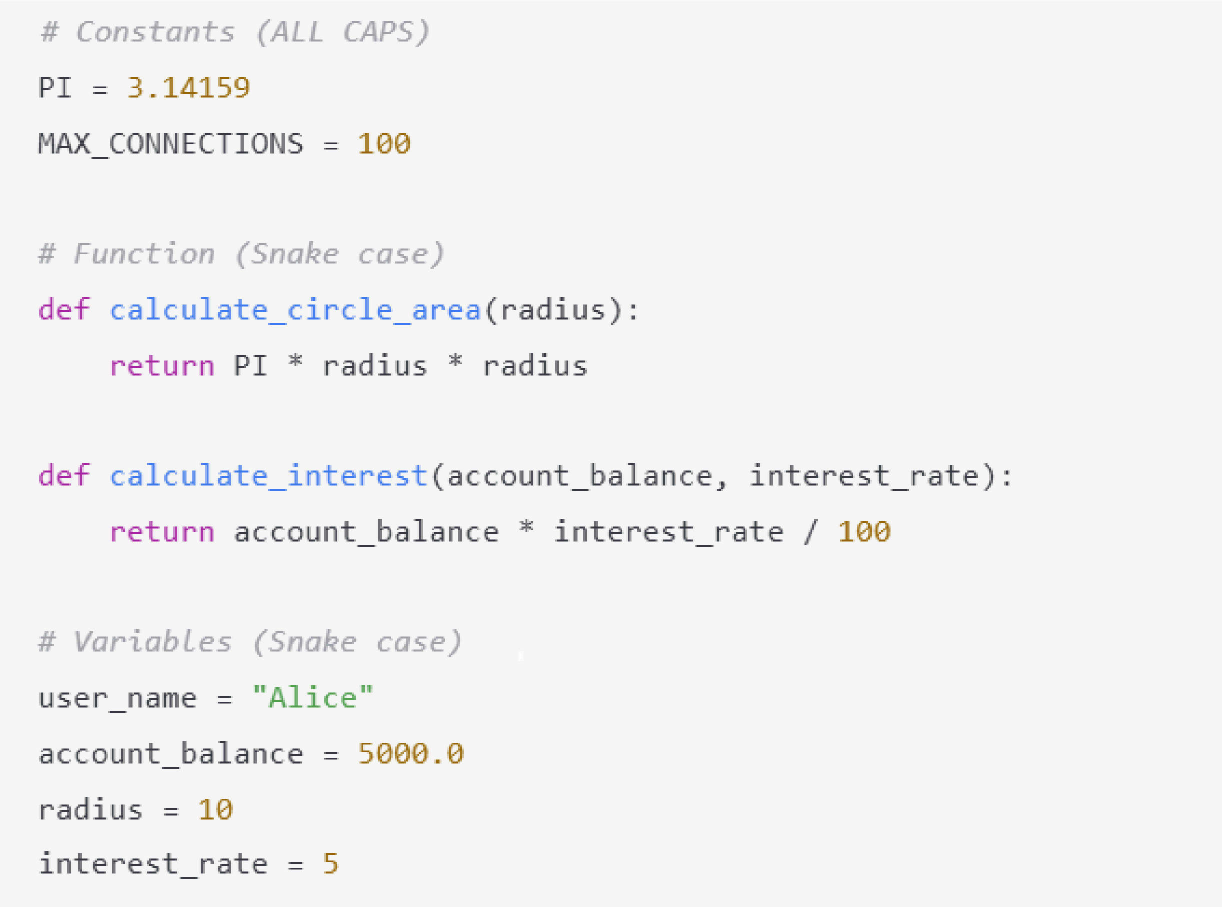
Different Modes of Working: Python provides various modes to write and execute code, each suited for different workflows and purposes.

1. **Interactive Mode:** Python code is executed directly in the Python shell or terminal.
   a. In cmd: type py or python to enter into the python shell.

            i. **exit()** to come out of the python shell.
    b. Search idle in the window's search to directly enter the python shell.
2. **Script Mode:**
    a. Write Python programs in a .py file and execute them as a whole.
    b. Run using the command: python filename.py.
3. **Integrated Development Environments (IDEs):**
    a. Use tools like PyCharm, Jupyter Notebook, or VS Code for an enhanced coding experience.
    b. Offers features like debugging, syntax highlighting, and autocompletion.

Python Fundamentals:

1. **Variables**
    a. A variable is a container. Called a variable because we can change the value of a variable.
    b. Snake Case for variables (using _'s), Pascalcase for class names, All caps for constants
    c. There are no constants in python. You can only show your intention by writing All Capital letters.
    d.



```
# Constants (ALL CAPS)
PI = 3.14159
MAX_CONNECTIONS = 100

# Function (Snake case)
def calculate_circle_area(radius):
    return PI * radius * radius

def calculate_interest(account_balance, interest_rate):
    return account_balance * interest_rate / 100

# Variables (Snake case)
user_name = "Alice"
account_balance = 5000.0
radius = 10
interest_rate = 5
```

2. **DataTypes** (All these are classes)
    a. **Built In data types:**
        i. Numeric – int, float, complex, bool

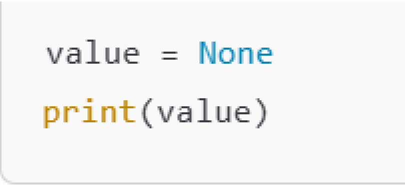
        ii. Sequence Type – string, list, tuple, Range

            1. There are no characters in Python. They are also strings

        iii. Mapping Type – dictionary (Key and Value pairs)

        iv. Set Type – Set

        v. None

            1. 

```
value = None
print(value)
```

    b. Custom data types:

        i. User defined classes: Custom data types are user-defined classes that can define their own attributes and methods.

3. Numeric Data types: Numeric types are used to store numerical values.

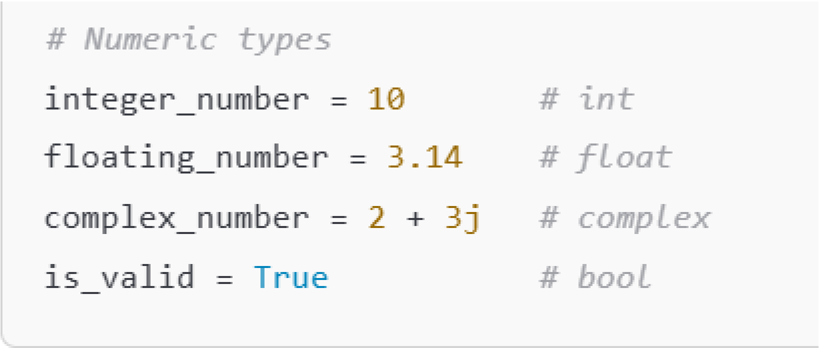
    a. int: Integer values (whole numbers).

    b. float: Floating-point numbers (decimal values).

    c. complex: Numbers with real and imaginary parts.

    d. bool: Boolean values (`True` or `False`).

    e. Examples:

        i. 

```
# Numeric types
integer_number = 10         # int
floating_number = 3.14      # float
complex_number = 2 + 3j     # complex
is_valid = True             # bool
```

4. Sequence Type: Sequence types represent ordered collections of items.

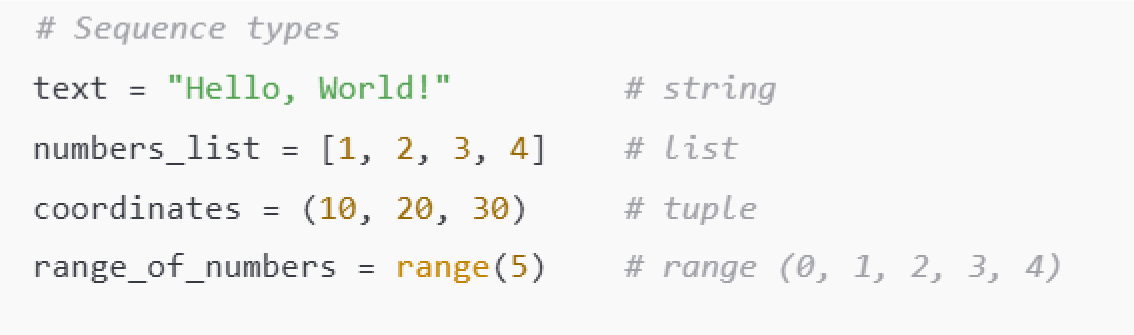
    a. **string**: A sequence of characters. Strings are immutable.

    b. **list**: An ordered, mutable collection of items.

    c. **tuple**: An ordered, immutable collection of items.

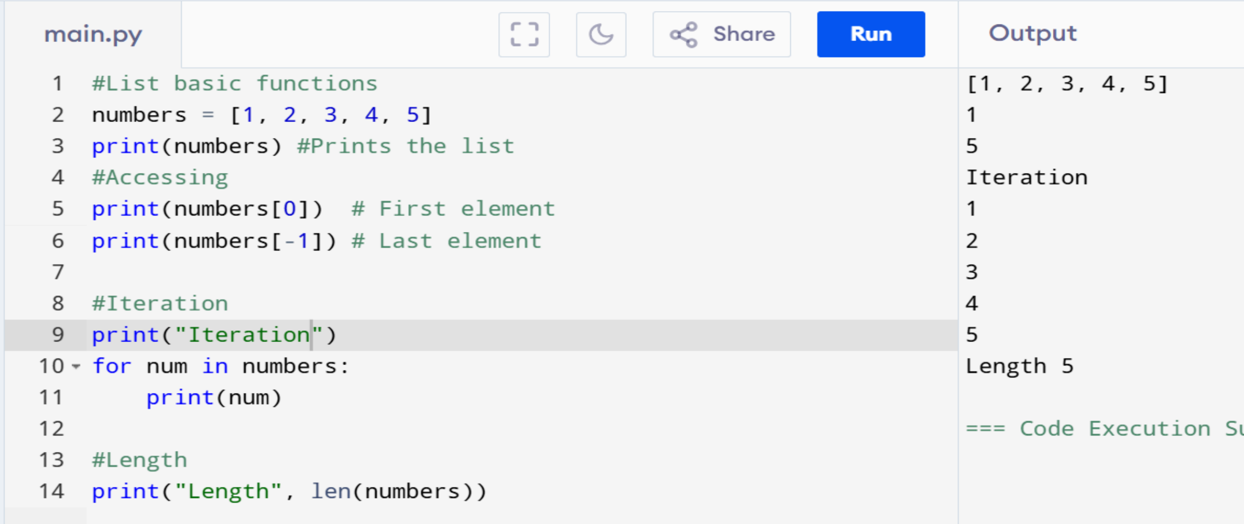
        i. Key difference between list and tuple is - **Lists** are mutable and Tuples are immutable.

    d. **range**: Represents a sequence of numbers, often used in loops.

e. 

```python
# Sequence types
text = "Hello, World!"          # string
numbers_list = [1, 2, 3, 4]     # list
coordinates = (10, 20, 30)      # tuple
range_of_numbers = range(5)     # range (0, 1, 2, 3, 4)
```

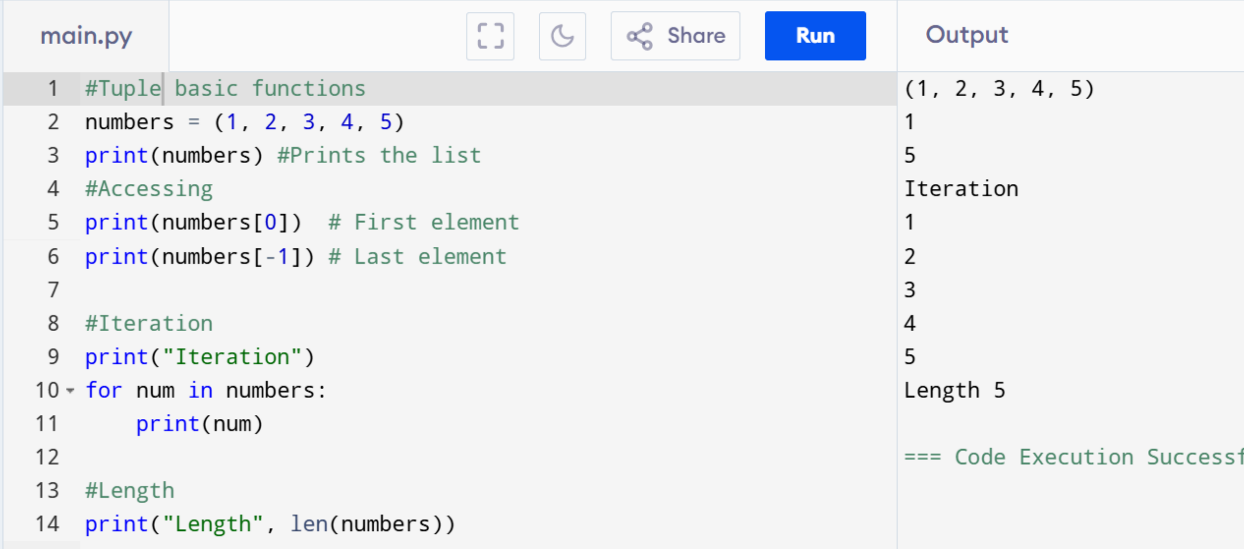
f. List basic functions:

i. 

```python
#List basic functions
numbers = [1, 2, 3, 4, 5]
print(numbers) #Prints the list
#Accessing
print(numbers[0])  # First element
print(numbers[-1]) # Last element

#Iteration
print("Iteration")
for num in numbers:
    print(num)

#Length
print("Length", len(numbers))
```

```
[1, 2, 3, 4, 5]
1
5
Iteration
1
2
3
4
5
Length 5

=== Code Execution Su
```

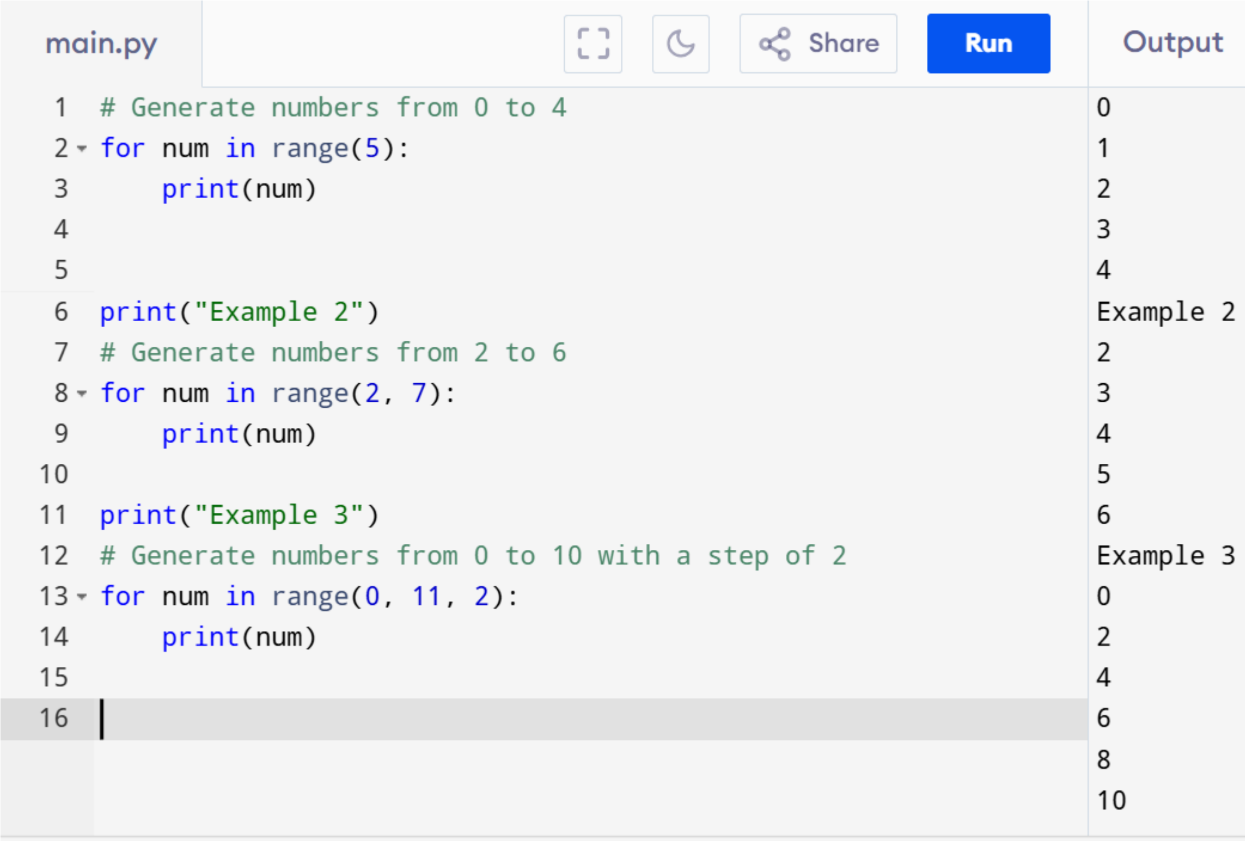
g. Tuple basic functions:

i. 

```python
#Tuple basic functions
numbers = (1, 2, 3, 4, 5)
print(numbers) #Prints the list
#Accessing
print(numbers[0])  # First element
print(numbers[-1]) # Last element

#Iteration
print("Iteration")
for num in numbers:
    print(num)

#Length
print("Length", len(numbers))
```

```
(1, 2, 3, 4, 5)
1
5
Iteration
1
2
3
4
5
Length 5

=== Code Execution Successf
```

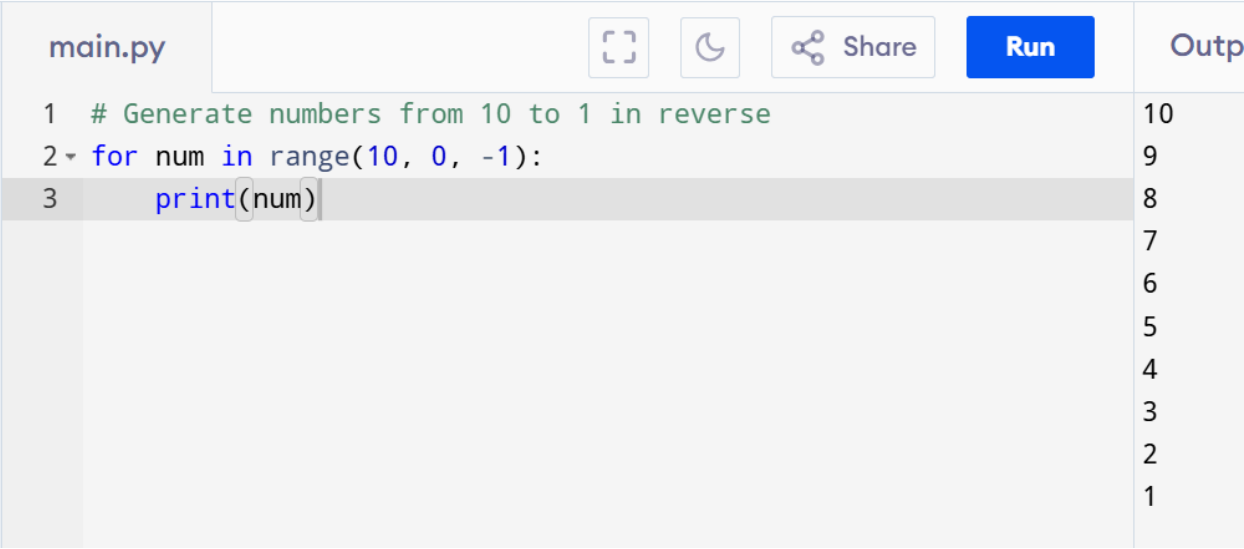
h. Range examples:



```python
# Generate numbers from 0 to 4
for num in range(5):
    print(num)


print("Example 2")
# Generate numbers from 2 to 6
for num in range(2, 7):
    print(num)

print("Example 3")
# Generate numbers from 0 to 10 with a step of 2
for num in range(0, 11, 2):
    print(num)

```

```
0
1
2
3
4
Example 2
2
3
4
5
6
Example 3
0
2
4
6
8
10
```

i.

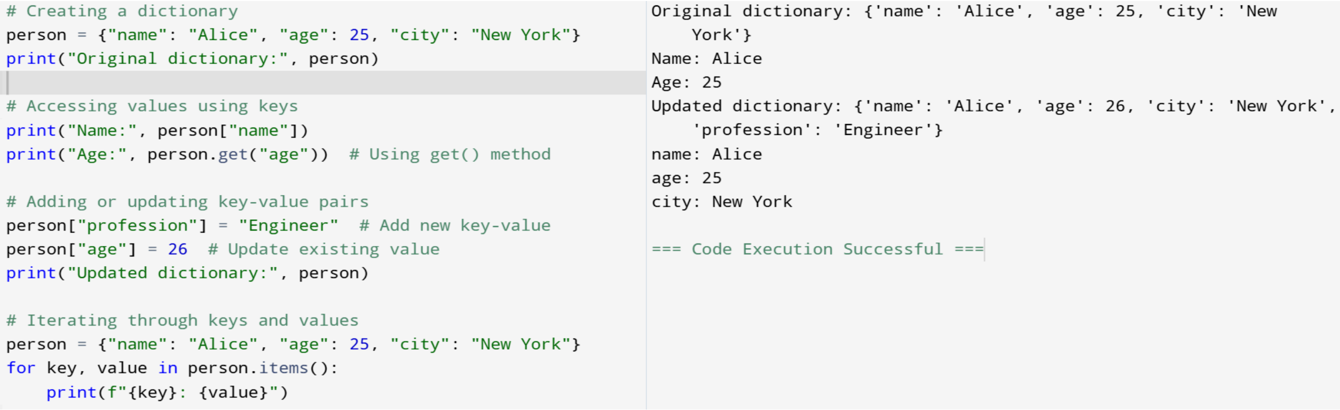


```python
# Generate numbers from 10 to 1 in reverse
for num in range(10, 0, -1):
    print(num)
```

```
10
9
8
7
6
5
4
3
2
1
```

ii.

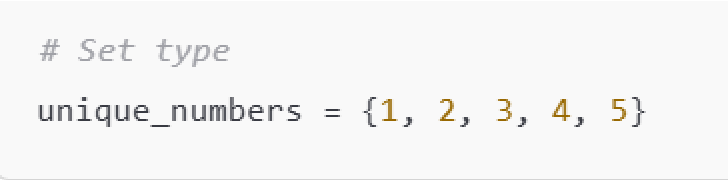
5. **Mapping Type:** Mapping types store key-value pairs.
    a. **Dictionary**: A collection of key-value pairs where keys must be unique.
    b. Basic operations on Dictionary:

c. 

```
# Creating a dictionary
person = {"name": "Alice", "age": 25, "city": "New York"}
print("Original dictionary:", person)

# Accessing values using keys
print("Name:", person["name"])
print("Age:", person.get("age"))  # Using get() method

# Adding or updating key-value pairs
person["profession"] = "Engineer"  # Add new key-value
person["age"] = 26  # Update existing value
print("Updated dictionary:", person)

# Iterating through keys and values
person = {"name": "Alice", "age": 25, "city": "New York"}
for key, value in person.items():
    print(f"{key}: {value}")
```

```
Original dictionary: {'name': 'Alice', 'age': 25, 'city': 'New
    York'}
Name: Alice
Age: 25
Updated dictionary: {'name': 'Alice', 'age': 26, 'city': 'New York',
    'profession': 'Engineer'}
name: Alice
age: 25
city: New York

=== Code Execution Successful ===
```

6. **Set Type:** A **set** is an unordered collection of unique items.

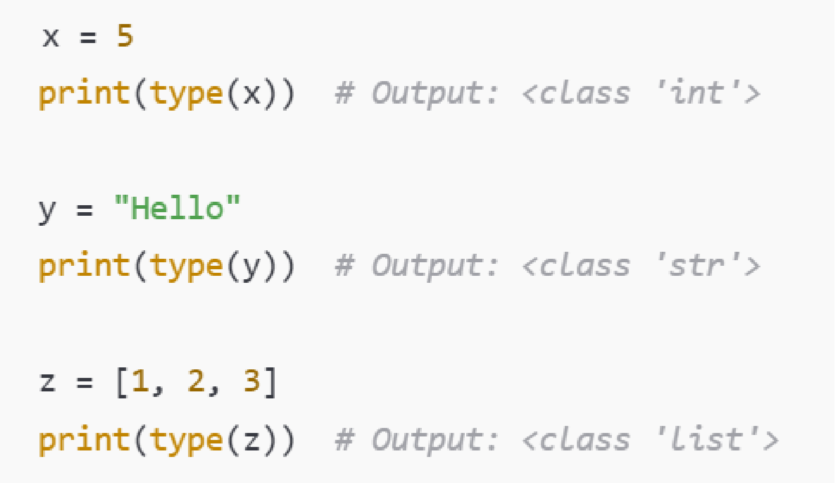
   a. 

   ```
   # Set type
   unique_numbers = {1, 2, 3, 4, 5}
   ```

   b. Order of the elements might not be same as the order of elements as in declaration

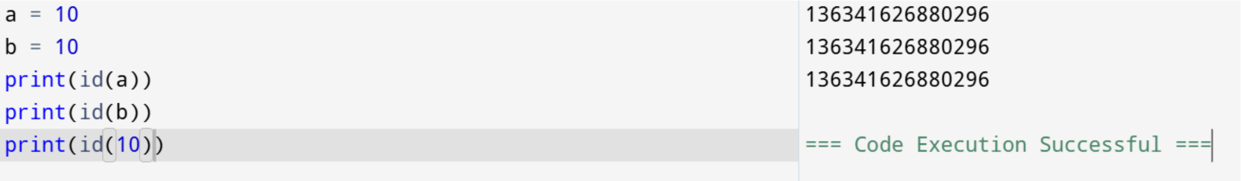
   c. Even if you give duplicates in declaration of set, it will store only one value.

7. **Introduce Type function**

   a. 

   ```
   x = 5
   print(type(x))  # Output: <class 'int'>

   y = "Hello"
   print(type(y))  # Output: <class 'str'>

   z = [1, 2, 3]
   print(type(z))  # Output: <class 'list'>
   ```

8. **Id function**

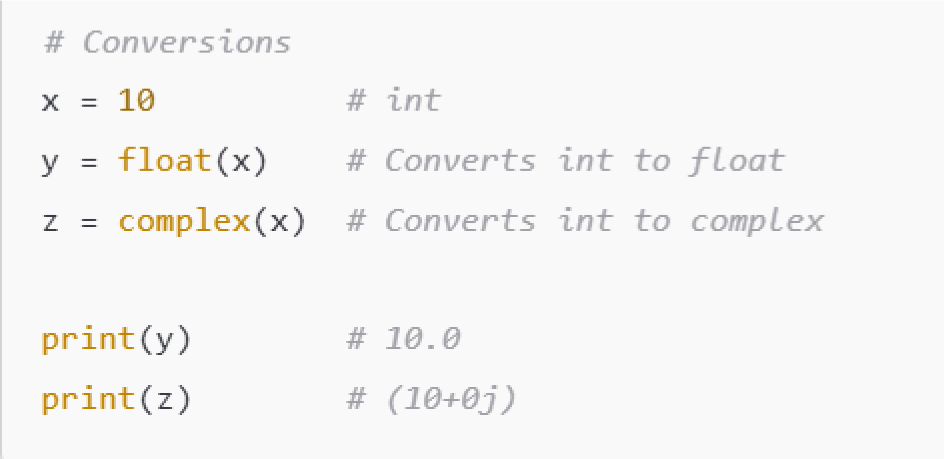
   a. 

   ```
   a = 10
   b = 10
   print(id(a))
   print(id(b))
   print(id(10))
   ```

   ```
   136341626880296
   136341626880296
   136341626880296

   === Code Execution Successful ===
   ```

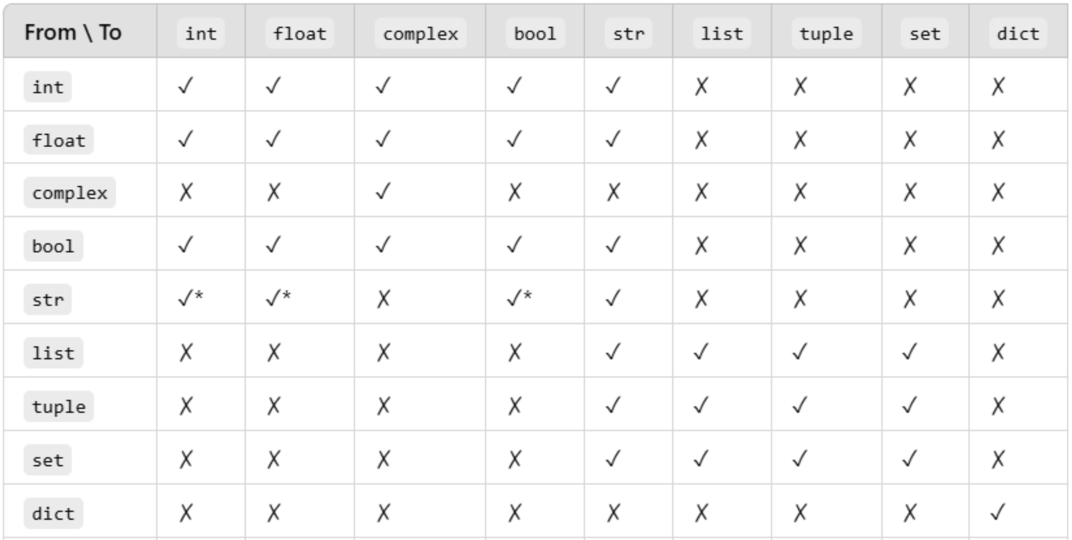
Numeric Types:

1. **Conversion from one type to another**

a. 

```
# Conversions
x = 10         # int
y = float(x)   # Converts int to float
z = complex(x) # Converts int to complex

print(y)       # 10.0
print(z)       # (10+0j)
```

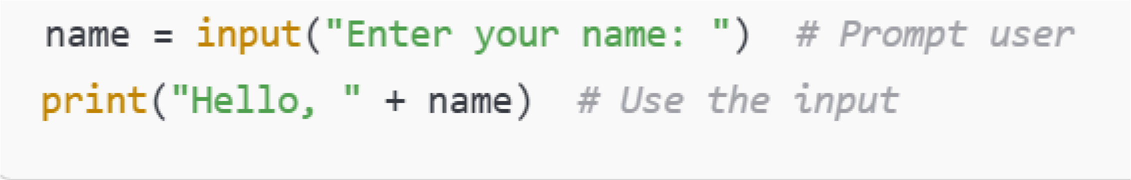
2. Similarly conversions can be done for other data types also.

3. 

From \ To	int	float	complex	bool	str	list	tuple	set	dict
int	✓	✓	✓	✓	✓	✗	✗	✗	✗
float	✓	✓	✓	✓	✓	✗	✗	✗	✗
complex	✗	✗	✓	✗	✗	✗	✗	✗	✗
bool	✓	✓	✓	✓	✓	✗	✗	✗	✗
str	✓*	✓*	✗	✓*	✓	✗	✗	✗	✗
list	✗	✗	✗	✗	✓	✓	✓	✓	✗
tuple	✗	✗	✗	✗	✓	✓	✓	✓	✗
set	✗	✗	✗	✗	✓	✓	✓	✓	✗
dict	✗	✗	✗	✗	✗	✗	✗	✗	✓

   a. ✓: Conversion is supported directly using type casting (e.g., `int(x)`, `float(x)`).
   b. ✓*: Supported if the string is formatted correctly (e.g., `int("123")`, `float("3.14")`).
   c. ✗: Conversion is not supported directly and may raise a `TypeError` or `ValueError`.

Basic Input and Output

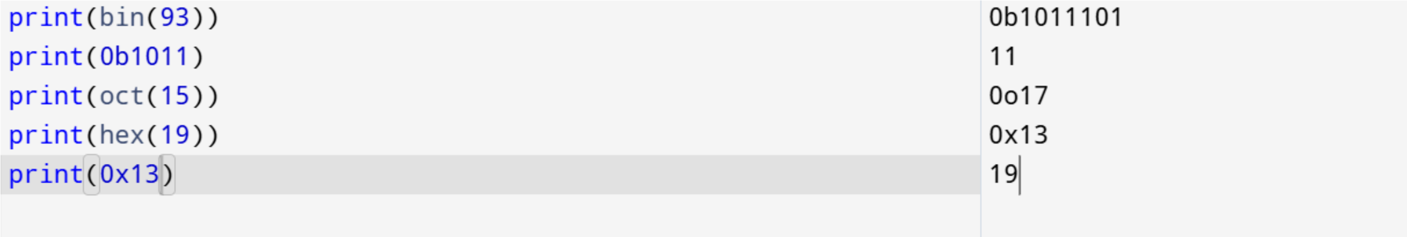
1. 

```
name = input("Enter your name: ")  # Prompt user
print("Hello, " + name)  # Use the input
```

2. Type conversion - int(), float()

Different Number Systems in Python:

1. Decimal, Binary, Octagonal, Hexadecimal system



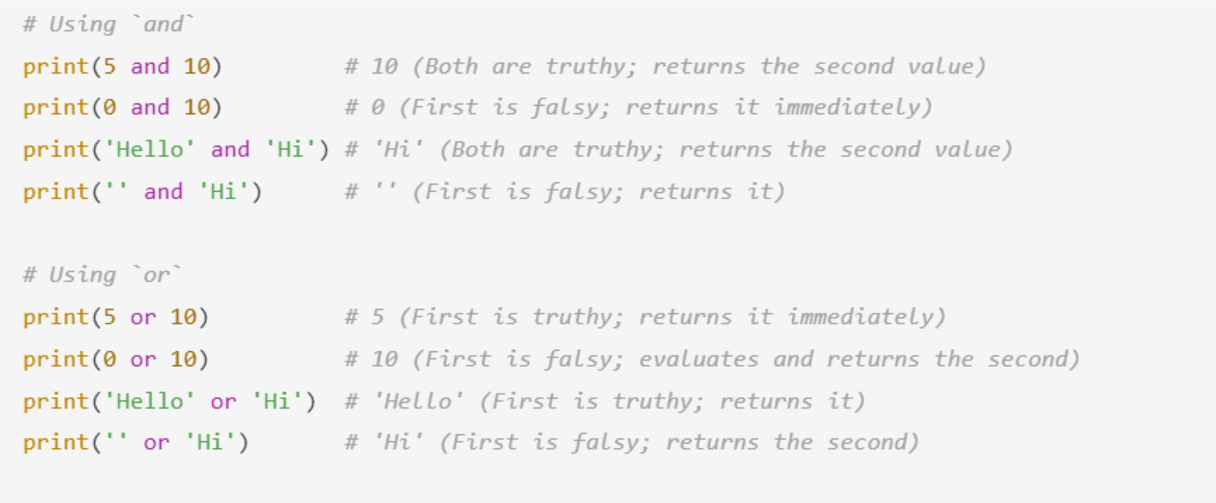
```
print(bin(93))
print(0b1011)
print(oct(15))
print(hex(19))
print(0x13)
```

```
0b1011101
11
0o17
0x13
19
```

2.

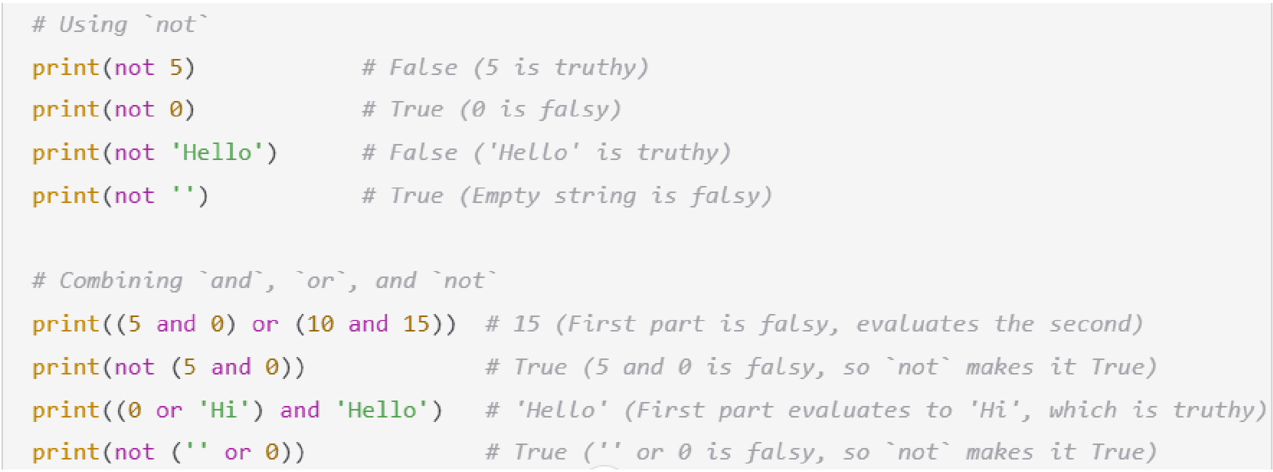
Operators

1. Arithmetic operators - (+, -, *, /, //, ** etc)
   a. / is float division
   b. // is integer division
2. Assignment operators - (=, +=, -=, *= etc)
   a. a,b = 2, 3
   b. n = -n
3. Relational operators - (<, >, <=, >=, ==, !=)
4. Logical operators - (and, or, not)
   a. Truthy and Falsy values in Python # Falsy: 0, 0.0, '', None, False, [], {}, (), set()
   b. Truthy: Non-zero numbers, non-empty strings, non-empty collections, True



```
# Using `and`
print(5 and 10)         # 10 (Both are truthy; returns the second value)
print(0 and 10)         # 0 (First is falsy; returns it immediately)
print('Hello' and 'Hi') # 'Hi' (Both are truthy; returns the second value)
print('' and 'Hi')      # '' (First is falsy; returns it)

# Using `or`
print(5 or 10)          # 5 (First is truthy; returns it immediately)
print(0 or 10)          # 10 (First is falsy; evaluates and returns the second)
print('Hello' or 'Hi')  # 'Hello' (First is truthy; returns it)
print('' or 'Hi')       # 'Hi' (First is falsy; returns the second)
```

   c.

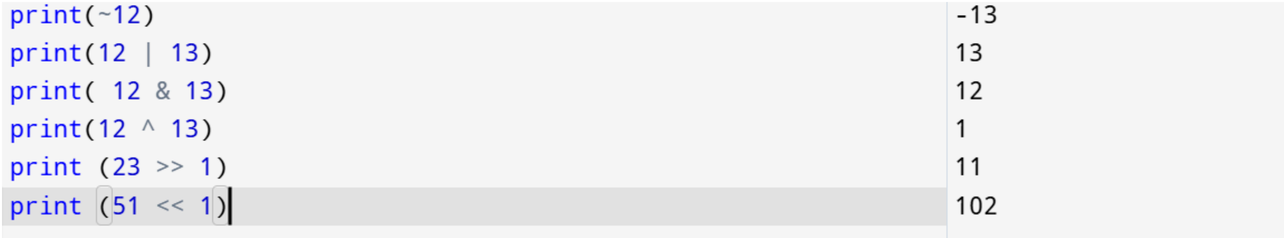
d. 

```
# Using `not`
print(not 5)            # False (5 is truthy)
print(not 0)            # True (0 is falsy)
print(not 'Hello')      # False ('Hello' is truthy)
print(not '')           # True (Empty string is falsy)

# Combining `and`, `or`, and `not`
print((5 and 0) or (10 and 15))  # 15 (First part is falsy, evaluates the second)
print(not (5 and 0))             # True (5 and 0 is falsy, so `not` makes it True)
print((0 or 'Hi') and 'Hello')   # 'Hello' (First part evaluates to 'Hi', which is truthy)
print(not ('' or 0))             # True ('' or 0 is falsy, so `not` makes it True)
```

Explanation of Results:

1. **and**: Returns the first falsy value if encountered; otherwise, the last truthy value.
2. **or**: Returns the first truthy value if encountered; otherwise, the last falsy value.
3. **not**: Negates the truth value of the operand.

5. Bitwise operator - Total 6 operators - Complement(~), &, |, ^, << (left shift), >> (right shift)

a. 

```
print(~12)                -13
print(12 | 13)            13
print( 12 & 13)           12
print(12 ^ 13)            1
print (23 >> 1)           11
print (51 << 1)           102
```

i. **Bitwise NOT (~):** Inverts all the bits of a number (flips `0` to `1` and vice versa).

ii. **Bitwise AND (&)**: Performs an AND operation on each bit of two numbers. The result is `1` if both bits are `1`; otherwise, `0`.

iii. **Bitwise OR (|) :** Performs an OR operation on each bit of two numbers. The result is `1` if either bit is `1`; otherwise, `0`.

iv. **XOR (^)**: Performs an XOR operation on each bit of two numbers. The result is `1` if the bits are different; otherwise, `0`

v. **Left Shift (<<):** Shifts the bits of a number to the left by the specified number of positions, effectively multiplying the number by `2` for each shift.

vi. **Right Shift (>>):** Shifts the bits of a number to the right by the specified number of positions, effectively dividing the number by `2` for each shift.

9. **Basic Troubleshooting (Syntax errors, Indentation)**

   a. **Syntax error: Common Causes**

      i. Missing colons (`:`) in `if`, `for`, `while`, etc.

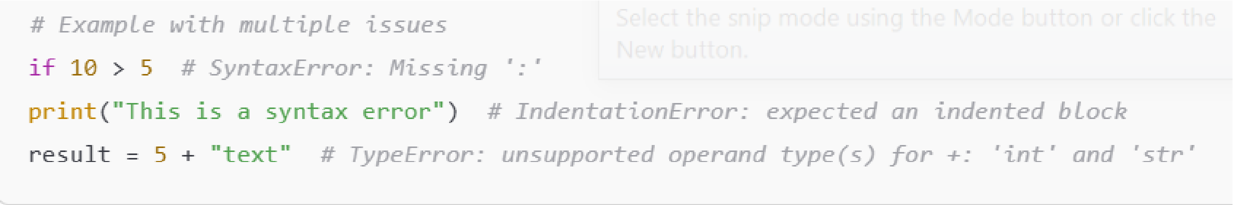
      ii. Using invalid characters. ?

      iii. Incorrect use of parentheses, brackets, or quotes.

b. **Indentation error:** Generally 4 spaces i.e one tab

c. **Type errors:**

  i. Example adding a number to a string

d. 

```
# Example with multiple issues
if 10 > 5  # SyntaxError: Missing ':'
print("This is a syntax error")  # IndentationError: expected an indented block
result = 5 + "text"  # TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Select the snip mode using the Mode button or click the New button.

Class 3 and Class 4:

A control statement is a programming construct that dictates the flow of execution in a program. It determines the order in which statements are executed based on certain conditions or repetitions. By using control statements, **you can make decisions, repeat actions, or jump to specific parts** of your program.

Categories of Control Statements:

Control statements are broadly classified into three categories:

1. Conditional Statements
   Used to execute specific blocks of code based on a condition.
   - Example: if, else, elif (or else if), switch (in some languages).
2. Looping Statements
   Allow a block of code to be executed repeatedly, either for a fixed number of times or until a condition is met.
   - Example: for, while, do-while.
3. Jump Statements
   Control the flow of loops or exit a block of code prematurely.
   - Example: break, continue, return, pass.

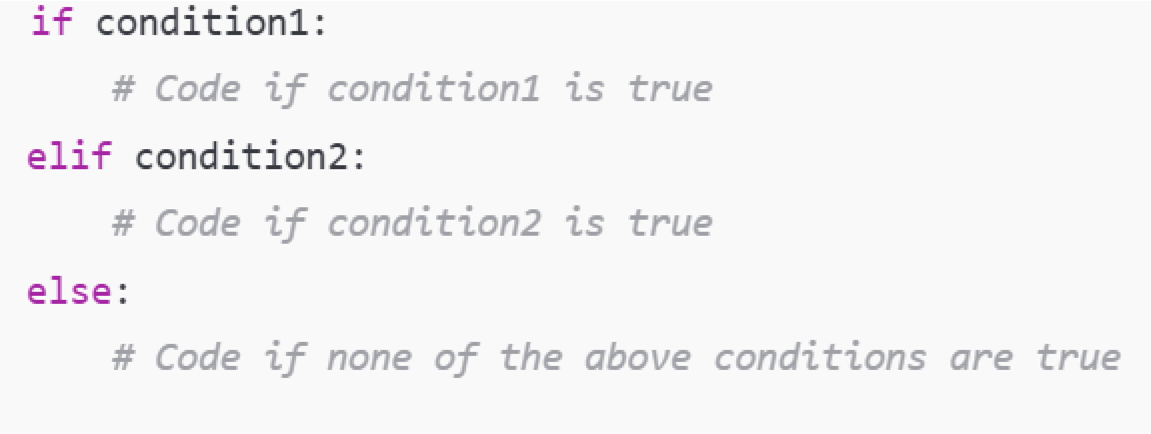
Purpose of Control Statements

1. Decision-making: Decide which path of code to execute (e.g., if-else).
2. Repetition: Repeat tasks (e.g., using loops).
3. Branching: Exit or skip specific parts of code when needed (break, continue).

1. Conditional Statements

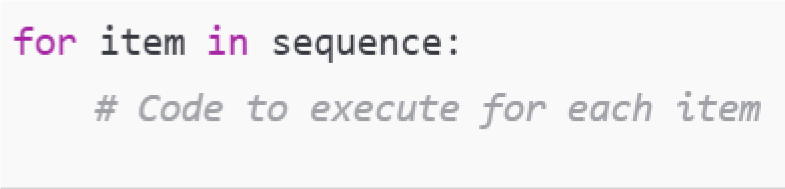
Conditional statements allow decision-making in your code, where specific blocks of code execute based on conditions.

1. If, else and elif:

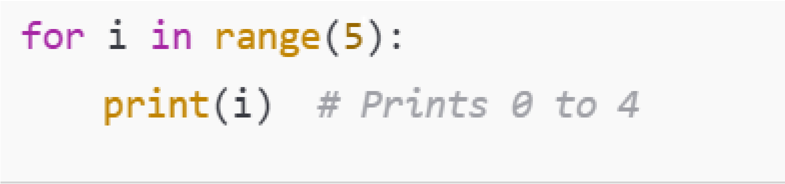
    a. 

```
if condition1:
    # Code if condition1 is true
elif condition2:
    # Code if condition2 is true
else:
    # Code if none of the above conditions are true
```

1. **Loops: For and While**
    a. Loops are used to execute a block of code repeatedly until a specific condition is met.
    b. For Loop:

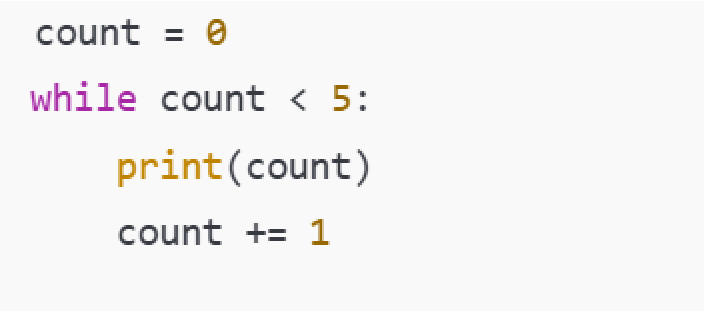
        i. 

```
for item in sequence:
    # Code to execute for each item
```

        ii. 

```
for i in range(5):
    print(i)  # Prints 0 to 4
```

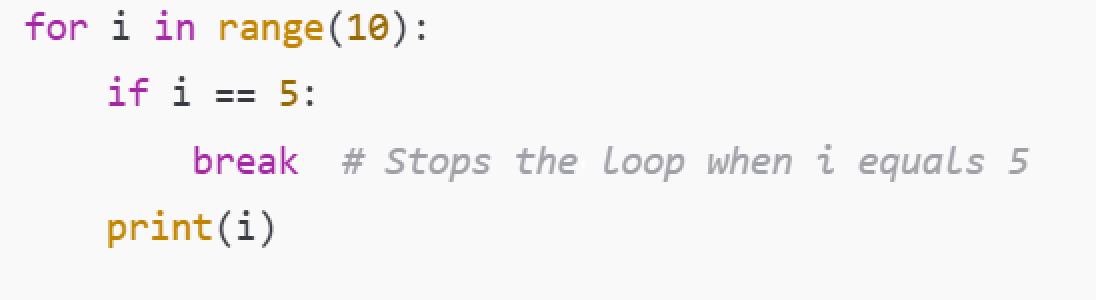
    c. **While:**

        i. 

```
count = 0
while count < 5:
    print(count)
    count += 1
```

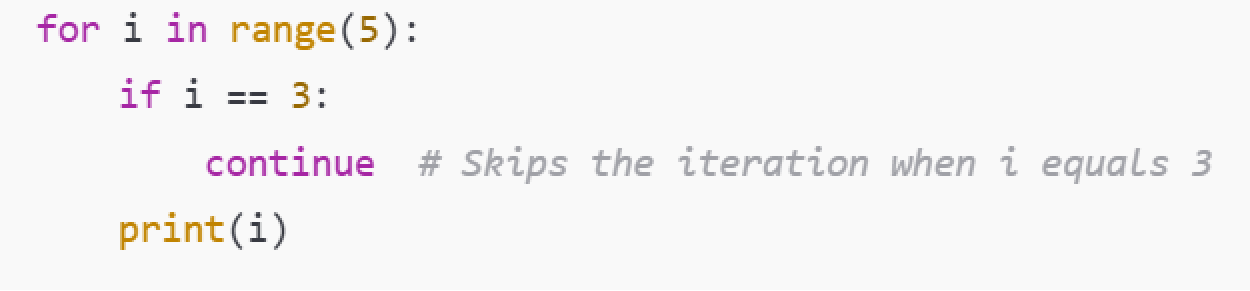
Break, continue, pass

1. **Break: Exits the loop prematurely when a specific condition is met.**

a. 

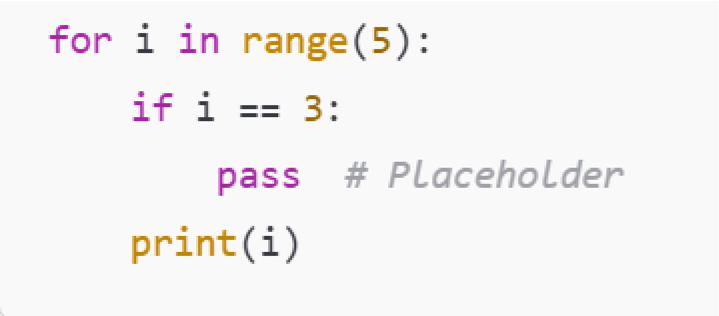
```
for i in range(10):
    if i == 5:
        break  # Stops the loop when i equals 5
    print(i)
```

2. **Continue: Skips the rest of the current loop iteration and moves to the next iteration.**

a. 

```
for i in range(5):
    if i == 3:
        continue  # Skips the iteration when i equals 3
    print(i)
```

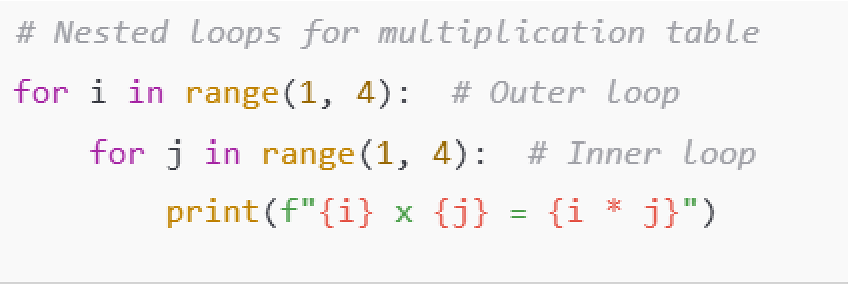
3. **Pass:** A placeholder that does nothing and allows for syntactically correct empty code blocks.

a. 

```
for i in range(5):
    if i == 3:
        pass  # Placeholder
    print(i)
```

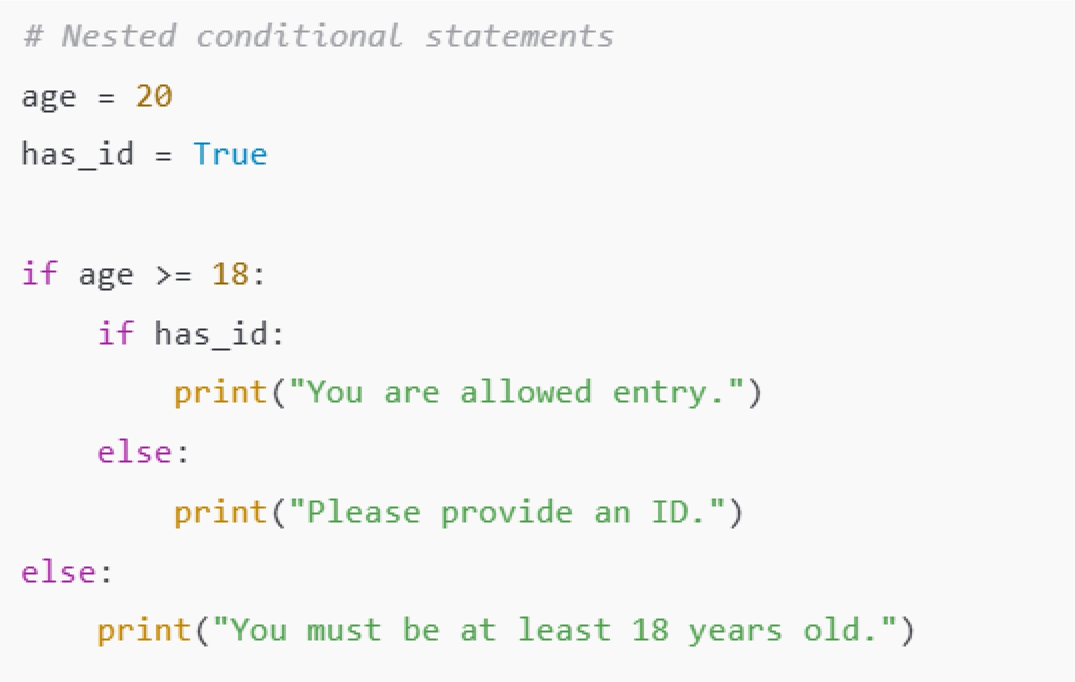
Nested loops and Nested conditionals:

1. **Nested Loops:** A **nested loop** is a loop inside another loop. The inner loop runs completely for every single iteration of the outer loop.

a. 

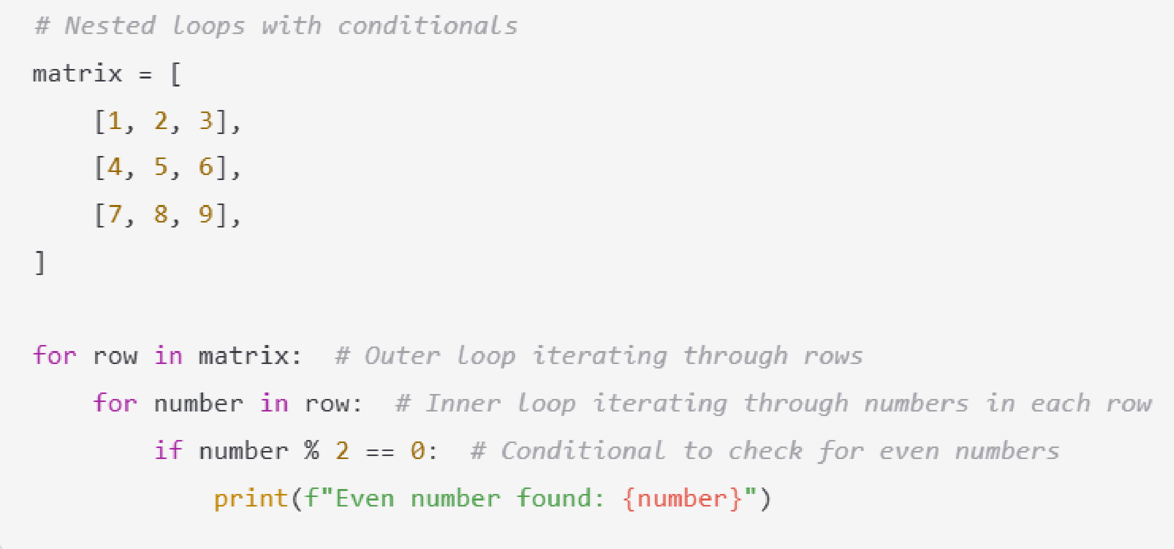
```
# Nested loops for multiplication table
for i in range(1, 4):  # Outer loop
    for j in range(1, 4):  # Inner loop
        print(f"{i} x {j} = {i * j}")
```

2. Nested Conditions:

a. 

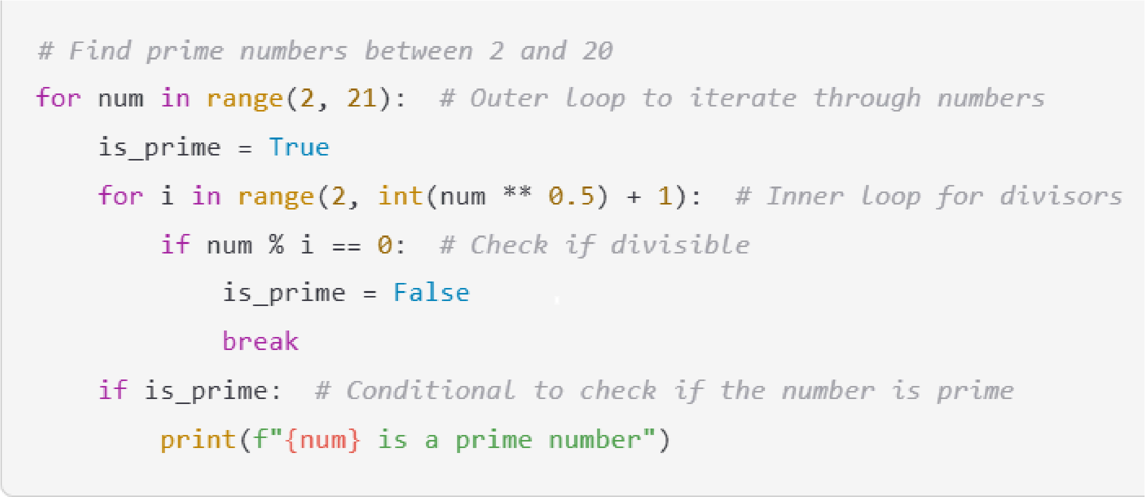
```python
# Nested conditional statements
age = 20
has_id = True

if age >= 18:
    if has_id:
        print("You are allowed entry.")
    else:
        print("Please provide an ID.")
else:
    print("You must be at least 18 years old.")
```

3. Nested Loops with Conditions:

   a. 

```python
# Nested loops with conditionals
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]

for row in matrix:  # Outer loop iterating through rows
    for number in row:  # Inner loop iterating through numbers in each row
        if number % 2 == 0:  # Conditional to check for even numbers
            print(f"Even number found: {number}")
```

   b. Example: Finding Prime Numbers in a given range:

      i. 

```python
# Find prime numbers between 2 and 20
for num in range(2, 21):  # Outer loop to iterate through numbers
    is_prime = True
    for i in range(2, int(num ** 0.5) + 1):  # Inner loop for divisors
        if num % i == 0:  # Check if divisible
            is_prime = False
            break
    if is_prime:  # Conditional to check if the number is prime
        print(f"{num} is a prime number")
```