CodroidHub summer training

Title :

For and While #LOOPS

1. Simple while loop
2. Simple for loop
3. Nested loops
4. String
5. List
6. Tuple
7. Set
8. Dictionary

Submitted by: DRISHTI GUPTA

ROLL NO: (2322825)

BRANCH: AI&ML

SUBMITTED TO: Mr. DEVASHISH SIR

(FOUNDER, CodroidHub Private Limited)

Simple while loop

A while loop in Python is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The loop continues to execute as long as the specified condition is True. When the condition becomes False, the loop stops running.

Syntax:

```
while condition:
    # code block to be executed
```

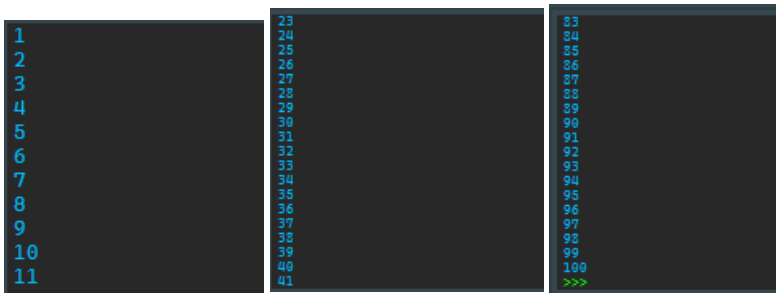
Key Points:

1. **Condition:** The loop checks the condition before executing the code block. If the condition is True, the code block is executed. If False, the loop terminates.
2. **Code Block:** The indented code block under the while statement is the code that gets executed on each iteration.
3. **Update:** Typically, there is an update within the loop to ensure that the condition eventually becomes False, preventing an infinite loop.

Example:

```
c=1 #starting point
while(c<=100):
    print(c) #infinite loop
    c=c+1 #To stop the loop
```

Output :



Simple for loop

A for loop in Python is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each item in the sequence. Unlike the while loop, which relies on a condition to continue running, a for loop automatically iterates over each element in the sequence.

Syntax:

for variable in sequence:

    # code block to be executed

Key Points:

1. **Variable:** This is the loop variable that takes the value of the current element of the sequence in each iteration.
2. **Sequence:** This is the collection of items over which the loop iterates.
3. **Code Block:** The indented block of code inside the loop that gets executed once for each item in the sequence.

Example :

```
for x in range (1,101):
    print (x)

nums=["apple","orange"]
for y in nums:
    print(y)
```

Output :

```
95
96
97
98
99
100
apple
orange
>>>
```

Nested loops

A nested loop in Python refers to a loop inside another loop. The outer loop runs first, and then the inner loop runs for each iteration of the outer loop. This structure is useful for iterating over multi-dimensional data structures like matrices or grids.

Syntax:

for outer_variable in outer_sequence:

for inner_variable in inner_sequence:

 # code block to be executed

Example:

```
n=6
for i in range(1,n):
    for j in range(1,i+1):
        print(j,end="")
    print()
```

Output :

```
1
12
123
1234
12345
>>>
```

In this code:

1. n is set to 6.
2. The outer loop runs from 1 to n-1 (i.e., 1 to 5).
3. The inner loop runs from 1 to i (inclusive) in each iteration of the outer loop.
4. print(j, end="") prints the current value of j without a newline, so the numbers in each row are printed on the same line.
5. print() outside the inner loop prints a newline after each row.

String

In Python, a string is a sequence of characters enclosed within single quotes ('), double quotes ("), or triple quotes (''' or """). Strings are used to represent text data and are immutable, meaning that once a string is created, it cannot be changed.

Creating Strings :

Strings can be created in several ways:

```python
# Using single quotes
str1 = 'Hello, World!'

# Using double quotes
str2 = "Hello, World!"

# Using triple quotes for multi-line strings
str3 = '''This is a
multi-line string.'''

str4 = """This is another
multi-line string."""
```

Accessing Characters :

You can access individual characters in a string using indexing, starting from 0 for the first character.

```python
s = "Python"
print(s[0])  # Output: P
print(s[1])  # Output: y
print(s[-1]) # Output: n (last character)
```

Slicing Strings :

You can extract a substring using slicing.

```python
s = "Python"
print(s[0:2])  # Output: Py (characters from index 0 to 1)
print(s[2:])   # Output: thon (characters from index 2 to the end)
print(s[:4])   # Output: Pyth (characters from the beginning to index 3)
print(s[-3:])  # Output: hon (last three characters)
```

Example :

```python
name="codroiodhub "
print(name)
print(name[0])
print(name[1:3])
print(name[-1])
print(name[1:-3])
print(name[0:])
print(name[:3])
print(name+name)
print(name *2)
a=5
b=str(a)
print(type(b))
```

Output :

```
codroiodhub
c
od

odroiodh
codroiodhub
cod
codroiodhub codroiodhub
codroiodhub codroiodhub
<class 'str'>
>>>
```

List

A list in Python is a built-in data structure that allows you to store an ordered collection of items. Lists are mutable, meaning you can modify their contents (add, remove, or change items) after they are created. Lists can contain items of different types, including integers, floats, strings, and even other lists.

Creating Lists

You can create lists using square brackets [] or the list() function.

```
# Creating an empty list
empty_list = []

# Creating a list with integers
numbers = [1, 2, 3, 4, 5]

# Creating a list with mixed data types
mixed_list = [1, "hello", 3.14, True]

# Creating a list using the list() function
another_list = list((1, 2, 3, 4, 5))
```

List Methods

Python provides many built-in methods to work with lists.

`len()`: Returns the length of the list.

```
numbers = [1, 2, 3, 4, 5]
print(len(numbers))  # Output: 5
```

`sort()`: Sorts the list in ascending order.

```
numbers = [5, 2, 3, 1, 4]
numbers.sort()
print(numbers)  # Output: [1, 2, 3, 4, 5]
```

`reverse()`: Reverses the order of the list.

```
numbers = [1, 2, 3, 4, 5]
numbers.reverse()
print(numbers)  # Output: [5, 4, 3, 2, 1]
```

Slicing Lists

You can extract a subset of a list using slicing.

```
numbers = [1, 2, 3, 4, 5]
print(numbers[1:4])  # Output: [2, 3, 4] (elements from index 1 to 3)
print(numbers[:3])   # Output: [1, 2, 3] (elements from the beginning to i
print(numbers[2:])   # Output: [3, 4, 5] (elements from index 2 to the end
print(numbers[-3:])  # Output: [3, 4, 5] (last three elements)
```

Nested Lists

Lists can contain other lists, creating nested or multi-dimensional lists.

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(nested_list[0])  # Output: [1, 2, 3]
print(nested_list[1][2])  # Output: 6 (second row, third element)
```

Example :

```
list=[1,2,3,4,5]
list1=["Drishti","Srishti","Kirti","Chirag"]
print(list1)
for data in list:
    print(data)
for data in list1:
    print(data)
    print(list1[0])
    print(list1[1:])
    print(list1[:3])
list1.sort()
print(list1)
list1.sort(reverse=True) #decending
```

Output :

```
['Drishti', 'Srishti', 'Kirti']
Kirti
Drishti
['Srishti', 'Kirti', 'Chirag']
['Drishti', 'Srishti', 'Kirti']
Chirag
Drishti
['Srishti', 'Kirti', 'Chirag']
['Drishti', 'Srishti', 'Kirti']
['Chirag', 'Drishti', 'Kirti', 'Srishti']
>>>
```

Tuple

A tuple in Python is an immutable, ordered collection of items. Once created, the items in a tuple cannot be changed, added, or removed. Tuples can contain items of different types, including integers, floats, strings, and even other tuples.

Creating Tuples

You can create tuples using parentheses () or the tuple() function.

```
# Creating an empty tuple
empty_tuple = ()

# Creating a tuple with integers
numbers = (1, 2, 3, 4, 5)

# Creating a tuple with mixed data types
mixed_tuple = (1, "hello", 3.14, True)

# Creating a tuple using the tuple() function
another_tuple = tuple((1, 2, 3, 4, 5))
```

Slicing Tuples

You can extract a subset of a tuple using slicing.

```
numbers = (1, 2, 3, 4, 5)
print(numbers[1:4])  # Output: (2, 3, 4)
print(numbers[:3])   # Output: (1, 2, 3)
print(numbers[2:])   # Output: (3, 4, 5)
print(numbers[-3:])  # Output: (3, 4, 5)
```

Example :

```
#tuple
mytuple=("Amit","Pooja","Raj","Rohit","Priya")
print(mytuple)
print(mytuple[0])
print(mytuple[1:])
print(mytuple[:3])
```

Output :

```
*** Remote Interpreter Reinitialized ***
('Amit', 'Pooja', 'Raj', 'Rohit', 'Priya')
Amit
('Pooja', 'Raj', 'Rohit', 'Priya')
('Amit', 'Pooja', 'Raj')
```

Tuple Methods

Tuples have only two built-in methods: count() and index().

count(): Returns the number of occurrences of a specified item in the tuple.

```
numbers = (1, 2, 3, 3, 4, 5)
print(numbers.count(3))  # Output: 2
```

index(): Returns the index of the first occurrence of a specified item in the tuple.

```
numbers = (1, 2, 3, 4, 5)
print(numbers.index(3))  # Output: 2
```

Nesting Tuples

Tuples can contain other tuples, creating nested or multi-dimensional tuples.

```
nested_tuple = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
print(nested_tuple[0])  # Output: (1, 2, 3)
print(nested_tuple[1][2])  # Output: 6 (second tuple, third element)
```

Set

A set in Python is an unordered collection of unique items. Sets are mutable, meaning you can add or remove items after a set is created, but they do not allow duplicate elements. Sets are useful for membership tests, eliminating duplicate entries, and performing mathematical set operations like union, intersection, difference, and symmetric difference.

Creating Sets

You can create sets using curly braces {} or the set() function.

```
# Creating an empty set
empty_set = set()

# Creating a set with integers
numbers = {1, 2, 3, 4, 5}

# Creating a set with mixed data types
mixed_set = {1, "hello", 3.14, True}

# Creating a set using the set() function
another_set = set([1, 2, 3, 4, 5])
```

Set Methods

Sets provide several built-in methods for common operations.

`len()`: Returns the number of elements in the set.

```
numbers = {1, 2, 3, 4, 5}
print(len(numbers))  # Output: 5
```

`in`: Checks if an element is in the set.

```
numbers = {1, 2, 3, 4, 5}
print(3 in numbers)  # Output: True
print(6 in numbers)  # Output: False
```

Example :

```
myset={"Amit","Pooja","Raj","Rohit","Priya"}
print(myset)

names={"Amit","Pooja","Raj","Rohit","Priya","Amit"}
print(names)
```

Output :

```
('Amit', 'Pooja', 'Raj')
{'Pooja', 'Priya', 'Amit', 'Raj', 'Rohit'}
{'Pooja', 'Priya', 'Amit', 'Raj', 'Rohit'}
>>>
```

Dictionary

A dictionary in Python is a mutable, unordered collection of key-value pairs. Each key-value pair maps the key to its associated value. Dictionaries are indexed by keys, which can be any immutable type, such as strings, numbers, or tuples. Values can be of any type, including other dictionaries.

Creating Dictionaries

You can create dictionaries using curly braces {} with key-value pairs separated by colons, or by using the dict() function.

```
# Creating an empty dictionary
empty_dict = {}

# Creating a dictionary with key-value pairs
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

# Creating a dictionary using the dict() function
another_dict = dict(name="Bob", age=25, city="Los Angeles")
```

Accessing and Modifying Dictionary Elements

You can access and modify values in a dictionary by using the keys.

Accessing Values :s

```
person = {"name": "Alice", "age": 30, "city": "New York"}
print(person["name"])  # Output: Alice
print(person["age"])   # Output: 30
```

Modifying Values :

```
person["age"] = 31
print(person)  # Output: {'name': 'Alice', 'age': 31, 'city': 'New York'}
```

Dictionary Methods

Dictionaries come with several built-in methods for common operations.

keys(), values(), and items()

keys(): Returns a view object containing the keys of the dictionary.

```
person = {"name": "Alice", "age": 30, "city": "New York"}
print(person.keys())  # Output: dict_keys(['name', 'age', 'city'])
```

values(): Returns a view object containing the values of the dictionary.

```
print(person.values())  # Output: dict_values(['Alice', 30, 'New York
```

items(): Returns a view object containing the key-value pairs of the dictionary.

```
print(person.items())  # Output: dict_items([('name', 'Alice'),
('age', 30), ('city', 'New York')])
```

Example :

```
#dictionary
mydic={"name":"Amit","Address":"Ambala","Course":"BIT"}
print(mydic)
print(mydic["name"])
print(mydic["Address"])
```

Output :

```
*** Remote Interpreter Reinitialized ***
{'name': 'Amit', 'Address': 'Ambala', 'Course': 'BIT'}
Amit
Ambala
>>>
```