Week 2

Python Control Statements, String and Lists

I. Introduction to Python Control Statements

Control statements in Python allow you to control the flow of your program. They enable you to execute code based on certain conditions, repeat blocks of code, and branch your code's execution path.

Types of Control Statements:

- Conditional Statements
- Looping Statements

1. Conditional Statements

if Statement

Syntax:

```
if condition:
    # code block
```

Example:

```
if 10 > 5:
    print("10 is greater than 5")
```

Output:

```
10 is greater than 5
```

if-else Statement

Syntax:

```
if condition:
    # code block
else:
    # code block
```

Example:

```
x = 10
if x % 2 == 0:
    print("x is even")
else:
    print("x is odd")
```

Output:

```
x is even
```

if-elif-else Statement

Syntax:

```
if condition1:
    # code block
elif condition2:
    # code block
else:
    # code block
```

Example:

```
x = 15
if x < 10:
    print("x is less than 10")
elif x == 10:
    print("x is 10")
else:
    print("x is greater than 10")
```

Output:

```
x is greater than 10
```

Nested if Statements

Syntax:

```
if condition1:
    if condition2:
        # code block
```

Example:

```
x = 10
y = 20
if x < 15:
    if y > 15:
        print("x is less than 15 and y is greater than 15")
```

Output:

```
x is less than 15 and y is greater than 15
```

2. Looping Statements

for Loop

Syntax:

```
for variable in iterable:
    # code block
```

Example:

```
for i in range(5):
    print(i)
```

Output:

```
0
1
2
3
4
```

while Loop

Syntax:

```
while condition:
    # code block
```

Example:

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

Output:

```
0
1
2
3
4
```

Nested Loops

Example:

```
for i in range(3):
    for j in range(2):
        print(f"i: {i}, j: {j}")
```

Output:

```
i: 0, j: 0
i: 0, j: 1
i: 1, j: 0
i: 1, j: 1
i: 2, j: 0
i: 2, j: 1
```

Loop Control Statements

break Statement

Example:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output:

```
0
1
2
```

continue Statement

Example:

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

Output:

```
0
1
2
4
```

pass Statement

Example:

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

Output:

```
0
1
```

```
2
3
4
```

Scope of Variables

Example:

```
def outer():
    x = "local"
    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner:", x)
    inner()
    print("outer:", x)

outer()
```

Output:

```
inner: nonlocal
outer: nonlocal
```

II. String Manipulation in Python

Strings in Python are used to record text information. Strings are immutable, which means they cannot be changed after they are created. This guide will explore the various functionalities of strings, including basic operations, methods, and how they can be used in Python.

Basic Concepts of Strings

Strings can be defined as a sequence of characters wrapped in single quotes (' ') or double quotes (" "). Python also supports multi-line strings, which can be denoted using triple quotes ('" "' or """ """).

Creating Strings

Example: Creating a simple string

```
my_string = 'Hello, world!'
```

```
print(my_string)
```

Output:

```
Hello, world!
```

Multi-line Strings

Example: Creating a multi-line string

```
multi_line_string = '''
This is
a multi-line
string
'''

print(multi_line_string)
```

Output:

```
This is
a multi-line
string
```

Common String Operations

Strings support a variety of operations, such as indexing, slicing, and concatenation.

1. Indexing

Description: Access individual characters in a string.

Syntax: string[index]

Example:

```
text = "Hello"

print(text[1])  # Outputs 'e'
```

2. Slicing

Description: Extract parts of strings by specifying start and end indices.

Syntax: string[start:end]

Example:

```
text = "Hello"

print(text[1:4])  # Outputs 'ell'
```

3. Concatenation

Description: Combine two or more strings into one.

Syntax: string1 + string2

Example:

```
greeting = "Hello"

name = "Alice"

message = greeting + ", " + name + "!"

print(message)  # Outputs 'Hello, Alice!'
```

4. Repetition

Description: Repeat strings a specified number of times.

Syntax: string * number

Example:

```
text = "Repeat "

print(text * 3)  # Outputs 'Repeat Repeat Repeat '
```

5. Stripping

Description: Remove leading and trailing characters (spaces by default).

Syntax: string.strip(chars)

Example:

```
text = "    Hello World    "

print(text.strip())  # Outputs 'Hello World'
```

6. Upper and Lower Case Conversion

Description: Convert all characters of the string to upper or lower case.

Syntax: string.upper(), string.lower()

Example:

```
 text = "Hello World"

print(text.upper())  # Outputs 'HELLO WORLD'

print(text.lower())  # Outputs 'hello world'
```

7. Finding Substrings

Description: Find the first occurrence of a substring.

Syntax: string.find(substring)

Example:

```
text = "Hello World"

print(text.find("World"))  # Outputs 6
```

8. Replacing Substrings

Description: Replace occurrences of a substring with another.

Syntax: string.replace(old, new)

Example:

```
text = "Hello World"

print(text.replace("World", "Python"))  # Outputs 'Hello Python'
```

9. Splitting Strings

Description: Split a string into a list of substrings based on a delimiter.

Syntax: string.split(delimiter)

Example:

text = "one,two,three"

print(text.split(",")) # Outputs ['one', 'two', 'three']

10. Joining Strings

Description: Join a list of strings into a single string with a separator.

Syntax: `separator.join(list_of_strings)`

Example:

words = ['Hello', 'world']

print(' '.join(words)) # Outputs 'Hello world'

11. Format Method

Description: Insert values into a string with placeholders.

Syntax: `'My name is {name}'.format(name="Alice")`

Example:

print('My name is {name}'.format(name="Alice")) # Outputs 'My name is Alice'

12. F-strings (Formatted String Literals)

Description: Embed expressions inside string literals for formatting.

Syntax: `f"Hello {name}!"`

Example:

name = "Alice"

print(f"Hello {name}!") # Outputs 'Hello Alice!'

III. Lists in Python

A list in Python is a mutable, ordered collection of items. Lists are one of the most versatile data structures in Python, allowing you to store a sequence of items in a single variable. Items in a list can be of any data type and can even include other lists.

1. **Creating Lists**

   Lists are created using square brackets [] with items separated by commas.

   Syntax:

   ```
   my_list = [item1, item2, item3, ...]
   ```

   Example:

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

   numbers = [1, 2, 3, 4, 5]

   mixed = [1, 'apple', 3.5, True]
   ```

2. **Accessing List Elements**

   You can access individual items in a list by using their index, which starts from 0.

   Syntax:

   ```
   list_name[index]
   ```

   Example:

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

   print(fruits[0])  # Output: apple

   print(fruits[1])  # Output: banana

   print(fruits[-1])  # Output: cherry (negative index starts from the end)
   ```

3. **Slicing Lists**

Slicing allows you to get a subset of a list by specifying a range of indices.

Syntax:

```
ame[start:end:step]
```

Example:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numbers[2:5])  # Output: [2, 3, 4]

print(numbers[:3])   # Output: [0, 1, 2]

print(numbers[4:])   # Output: [4, 5, 6, 7, 8, 9]

print(numbers[::2])  # Output: [0, 2, 4, 6, 8]
```

4. Modifying Lists

Lists are mutable, meaning you can change their contents.

1. Changing an Item:

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

fruits[1] = 'blueberry'

print(fruits)  # Output: ['apple', 'blueberry', 'cherry']
```

2. Adding Items:

Append: Adds an item to the end of the list.

```
fruits.append('orange')

print(fruits)  # Output: ['apple', 'blueberry', 'cherry', 'orange']
```

Insert: Adds an item at a specified index.

```
fruits.insert(1, 'mango')

print(fruits)  # Output: ['apple', 'mango', 'blueberry', 'cherry', 'orange']
```

3. Removing Items:

Remove: Removes the first occurrence of a specified item.

```
fruits.remove('blueberry')

print(fruits)  # Output: ['apple', 'mango', 'cherry', 'orange']
```

Pop: Removes and returns an item at a specified index. If no index is specified, removes the last item.

```
fruits.pop(2)

print(fruits)  # Output: ['apple', 'mango', 'orange']

last_item = fruits.pop()

print(last_item)  # Output: orange

print(fruits)  # Output: ['apple', 'mango']
```

Del: Deletes an item at a specified index.

```
del fruits[1]

print(fruits)  # Output: ['apple']
```

5. List Methods

Python lists come with several built-in methods to perform common operations.

1. Append: Adds an element to the end of the list.

```
fruits.append('kiwi')
```

2. Extend: Adds all elements from another list to the end of the list.

```
fruits.extend(['grape', 'pineapple'])
```

3. Insert: Inserts an element at a specified index.

```
fruits.insert(1, 'peach')
```

4. Remove: Removes the first occurrence of an element.

```
fruits.remove('apple')
```

5. Pop: Removes and returns an element at a specified index.

```
last_fruit = fruits.pop()
```

6. Clear: Removes all elements from the list.

```
fruits.clear()
```

7. Index: Returns the index of the first occurrence of an element.

```
index = fruits.index('banana')
```

8. Count: Returns the number of occurrences of an element.

```
count = fruits.count('apple')
```

9. Sort: Sorts the list in ascending order.

```
fruits.sort()
```

10. Reverse: Reverses the elements of the list.

```
fruits.reverse()
```

11. Copy: Returns a shallow copy of the list.

```
fruits_copy = fruits.copy()
```

6. List Comprehensions

List comprehensions provide a concise way to create lists. They consist of brackets containing an expression followed by a for clause.

Syntax:

```
new_list = [expression for item in iterable if condition]
```

Example:

```
squares = [x**2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Nested Lists

Lists can contain other lists, creating a nested structure.

Example:

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(nested_list[0])  # Output: [1, 2, 3]

print(nested_list[0][1])  # Output: 2
```

7. Common List Operations

1. Length:

```
len(fruits)
```

2. Membership:

```
'apple' in fruits  # Output: True
```

3. Iteration:

```
for fruit in fruits:
    print(fruit)
```

4. Concatenation:

```
combined = fruits + ['kiwi', 'grape']
```

5. Repetition:

```
repeated = fruits * 2
```

6. List to String Conversion:

```
fruit_string = ', '.join(fruits)
```

Examples

1. Basic List Operations:

```
numbers = [1, 2, 3, 4, 5]

numbers.append(6)

numbers.insert(0, 0)

numbers.remove(3)

print(numbers)  # Output: [0, 1, 2, 4, 5, 6]
```

2. Using List Comprehensions:

```
even_numbers = [x for x in range(20) if x % 2 == 0]

print(even_numbers)  # Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

3. Nested Lists:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in matrix:

    print(row)

# Output:

# [1, 2, 3]

# [4, 5, 6]

# [7, 8, 9]
```