3/100 Day of Data Science

Eternaltek

Data Types in python

1- List

- In Python, a list is a built-in data type that represents a collection of elements.
- Lists are mutable, meaning you can change their content by adding or removing elements.
- Lists are defined using square brackets [], and elements are separated by commas.

In [1]:

```python
# Creating a list
my_list = [1, 2, 3, 4, 5]

# Accessing elements in a list
print(my_list[0])  # Output: 1
print(my_list[2])  # Output: 3

# Modifying elements in a list
my_list[1] = 10
print(my_list)  # Output: [1, 10, 3, 4, 5]

# Adding elements to a list
my_list.append(6)
print(my_list)  # Output: [1, 10, 3, 4, 5, 6]

# Removing elements from a list
my_list.remove(3)
print(my_list)  # Output: [1, 10, 4, 5, 6]

# Length of a list
print(len(my_list))  # Output: 5

# Iterating through a list
for element in my_list:
    print(element)
```

```
1
3
[1, 10, 3, 4, 5]
[1, 10, 3, 4, 5, 6]
[1, 10, 4, 5, 6]
5
1
10
4
5
6
```

Here are some common list methods:

- append(x): Adds element x to the end of the list.
- extend(iterable): Appends elements of the iterable to the end of the list.
- insert(i, x): Inserts element x at position i in the list.
- remove(x): Removes the first occurrence of element x from the list.
- pop([i]): Removes and returns the element at position i. If i is not specified, it removes and returns the last element.
- index(x): Returns the index of the first occurrence of element x in the list.
- count(x): Returns the number of occurrences of element x in the list.
- sort(): Sorts the elements of the list in ascending order.
- reverse(): Reverses the elements of the list in place.

2 - String

- In Python, a string is a built-in data type used to represent text.
- Strings are sequences of characters, and they are defined using either single quotes (') or double quotes (").
- Strings in Python are immutable, meaning once a string is created, you cannot change its content.

In [2]:

```python
# Creating strings
single_quoted_string = 'Hello, World!'
double_quoted_string = "Hello, World!"

# Printing strings
print(single_quoted_string)  # Output: Hello, World!
print(double_quoted_string)  # Output: Hello, World!

# Accessing characters in a string
first_char = single_quoted_string[0]  # The first character
print(first_char)  # Output: H

# String length
length = len(single_quoted_string)
print(length)  # Output: 13
```

```
# Concatenating strings
concatenated_string = single_quoted_string + ' Welcome!'
print(concatenated_string)  # Output: Hello, World! Welcome!

# String slicing
substring = single_quoted_string[7:12]
print(substring)  # Output: World

# String methods
uppercase_string = single_quoted_string.upper()
print(uppercase_string)  # Output: HELLO, WORLD!

lowercase_string = double_quoted_string.lower()
print(lowercase_string)  # Output: hello, world!

# String formatting
name = "Alice"
age = 30
formatted_string = f"My name is {name} and I'm {age} years old."
print(formatted_string)  # Output: My name is Alice and I'm 30 years old.
```

```
Hello, World!
Hello, World!
H
13
Hello, World! Welcome!
World
HELLO, WORLD!
hello, world!
My name is Alice and I'm 30 years old.
```

In [ ]:

```

```

- Python also supports triple-quoted strings (''' or """) that can span multiple lines, which is useful for multiline strings, docstrings, or formatting longer text.

In [3]:

```
multiline_string = '''This is a
multiline
string.'''
print(multiline_string)
```

```
This is a
multiline
string.
```

- Strings in Python come with a variety of built-in methods that allow you to perform various operations on strings.
- Here are some commonly used string methods:

len(): Returns the length of the string.

In [4]:

```
my_string = "Hello, World!"
length = len(my_string)
print(length)  # Output: 13
```

```
13
```

upper() and lower(): Converts all characters to uppercase or lowercase.

In [5]:

```
my_string = "Hello, World!"
uppercase_string = my_string.upper()
lowercase_string = my_string.lower()
print(uppercase_string)  # Output: HELLO, WORLD!
print(lowercase_string)  # Output: hello, world!
```

```
HELLO, WORLD!
hello, world!
```

strip(), lstrip(), and rstrip(): Removes leading and trailing whitespaces from the string.

In [6]:

```
my_string = "   Hello, World!   "
stripped_string = my_string.strip()
print(stripped_string)  # Output: Hello, World!
```

```
Hello, World!
```

In [ ]:

3 Tuple

- In Python, a tuple is a built-in data type that represents an ordered, immutable collection of elements.
- Tuples are similar to lists, but the key difference is that once a tuple is created, its elements cannot be changed, added, or removed.
- Tuples are defined using parentheses ().

In [7]:

```
# Creating a tuple
my_tuple = (1, 2, 3, 'a', 'b', 'c')

# Accessing elements in a tuple
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: a

# Tuple length
length = len(my_tuple)
print(length)  # Output: 6

# Iterating through a tuple
for element in my_tuple:
    print(element)

# Tuple concatenation
tuple1 = (1, 2, 3)
tuple2 = ('a', 'b', 'c')
concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple)  # Output: (1, 2, 3, 'a', 'b', 'c')
```

```
1
a
6
1
2
3
a
b
c
(1, 2, 3, 'a', 'b', 'c')
```

- Since tuples are immutable, you cannot modify their elements, but you can create new tuples with modified content.

In [8]:

```python
# Creating a new tuple with modified content
modified_tuple = my_tuple[:3] + (4, 5, 6) + my_tuple[3:]
print(modified_tuple)
```

```
(1, 2, 3, 4, 5, 6, 'a', 'b', 'c')
```

4 Dictionary

- In Python, a dictionary is a built-in data type that represents an unordered collection of key-value pairs.
- Dictionaries are sometimes also known as associative arrays or hash maps in other programming languages.
- They are defined using curly braces {} and consist of key-value pairs separated by colons.

In [9]:

```python
# Creating a dictionary
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

# Accessing values using keys
print(my_dict['name'])  # Output: John
print(my_dict['age'])   # Output: 25

# Modifying values
my_dict['age'] = 26
print(my_dict['age'])   # Output: 26

# Adding a new key-value pair
my_dict['occupation'] = 'Engineer'
print(my_dict)          # Output: {'name': 'John', 'age': 26, 'city': 'New York', 'occ

# Removing a key-value pair
del my_dict['city']
print(my_dict)          # Output: {'name': 'John', 'age': 26, 'occupation': 'Engineer'
```

```
John
25
26
{'name': 'John', 'age': 26, 'city': 'New York', 'occupation': 'Engineer'}
{'name': 'John', 'age': 26, 'occupation': 'Engineer'}
```

In [10]:

```
# Dictionary with mixed data types
mixed_dict = {'name': 'Alice', 'age': 30, 'grades': [90, 85, 92], 'contact': {'email':

# Accessing nested values
print(mixed_dict['grades'][0])  # Output: 90
print(mixed_dict['contact']['email'])  # Output: alice@example.com
```

```
90
alice@example.com
```

Common dictionary methods include:

- keys(): Returns a list of all keys in the dictionary.
- values(): Returns a list of all values in the dictionary.
- items(): Returns a list of key-value pairs as tuples.
- get(key, default): Returns the value associated with the given key, or a default value if the key is not found.
- update(other_dict): Updates the dictionary with key-value pairs from another dictionary.
- pop(key, default): Removes and returns the value associated with the given key, or a default value if the key is not found.

5 Set

- In Python, a set is a built-in data type that represents an unordered collection of unique elements.
- Sets are defined using curly braces {}, similar to dictionaries, but without key-value pairs.

In [11]:

```
# Creating a set
my_set = {1, 2, 3, 4, 5}

# Printing the set
print(my_set)  # Output: {1, 2, 3, 4, 5}
```

```
{1, 2, 3, 4, 5}
```

In [12]:

```
# Creating a set from a list
my_list = [1, 2, 2, 3, 4, 4, 5]
my_set_from_list = set(my_list)

# Printing the set
print(my_set_from_list)  # Output: {1, 2, 3, 4, 5}
```

```
{1, 2, 3, 4, 5}
```

In [13]:

```
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

# Union
union_set = set1.union(set2)
print(union_set)  # Output: {1, 2, 3, 4, 5, 6, 7}

# Intersection
```

```
intersection_set = set1.intersection(set2)
print(intersection_set)  # Output: {3, 4, 5}

# Difference
difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2}

# Symmetric Difference
symmetric_difference_set = set1.symmetric_difference(set2)
print(symmetric_difference_set)  # Output: {1, 2, 6, 7}
```

```
{1, 2, 3, 4, 5, 6, 7}
{3, 4, 5}
{1, 2}
{1, 2, 6, 7}
```

In [14]:

```
# Adding elements to a set
my_set.add(6)
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}

# Removing elements from a set
my_set.remove(3)
print(my_set)  # Output: {1, 2, 4, 5, 6}
```

```
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6}
```

In [ ]:

Conditional statements

- Conditional statements in Python allow you to control the flow of your program based on certain conditions.
- The most common conditional statements are if, elif (else if), and else.

if statement:

The if statement is used to execute a block of code if a particular condition is true.

In [15]:

```
# Example 1
x = 10

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

```
x is greater than 5
```

if-else statement:

The if-else statement allows you to specify two blocks of code: one to be executed if the condition is true and another if the condition is false.

In [16]:

```
# Example 2
y = 3

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

```
y is odd
```

if-elif-else statement:

The if-elif-else statement allows you to test multiple conditions sequentially. The first true condition encountered will execute its corresponding block of code, and subsequent conditions will be skipped.

In [17]:

```
# Example 3
z = 0

if z > 0:
    print("z is positive")
elif z < 0:
    print("z is negative")
else:
    print("z is zero")
```

```
z is zero
```

Nested if statements:

You can also nest if statements inside other if, elif, or else blocks to create more complex conditions.

In [18]:

```
# Example 4
a = 15

if a > 10:
    print("a is greater than 10")

    if a % 2 == 0:
        print("a is also even")
    else:
        print("a is odd")
else:
    print("a is not greater than 10")
```

```
a is greater than 10
a is odd
```

In [ ]:

Loop Statements

In Python, both while and for are loop structures that allow you to execute a block of code repeatedly.

while loop:

The while loop repeatedly executes a block of code as long as a specified condition is true.

In [20]:

```
# Example 1: Simple while loop
count = 0
while count < 5:
    print(f"Count is {count}")
    count += 1
```

```
Count is 0
Count is 1
Count is 2
Count is 3
Count is 4
```

In [ ]:

for loop:

The for loop is used for iterating over a sequence (such as a list, tuple, string, or range) or other iterable objects.

In [21]:

```
# Example 3: Iterating over a list
fruits = ["apple", "orange", "banana"]
for fruit in fruits:
    print(f"I like {fruit}s")
```

```
I like apples
I like oranges
I like bananas
```

In [ ]:

Control Statements

In Python, pass, break, and continue are control flow statements that can be used within loops or conditional statements to control the program's execution.

1. pass statement:

The pass statement is a no-operation statement. It is used when a statement is syntactically required, but you want to do nothing. It serves as a placeholder.

In [22]:

```python
# Example with pass
for i in range(5):
    if i == 2:
        pass  # Do nothing when i is 2
    else:
        print(i)
```

```
0
1
3
4
```

2. break statement:

The break statement is used to exit a loop prematurely. When encountered, the loop is immediately terminated, and the program continues with the next statement after the loop.

In [23]:

```python
# Example with break
for i in range(5):
    if i == 3:
        print("Breaking the loop at", i)
        break
    else:
        print(i)
```

```
0
1
2
Breaking the loop at 3
```

3. continue statement:

The continue statement is used to skip the rest of the code inside a loop for the current iteration and move to the next iteration.

In [24]:

```python
# Example with continue
for i in range(5):
    if i == 2:
        print("Skipping iteration at", i)
        continue
    else:
        print(i)
```

```
0
1
Skipping iteration at 2
3
4
```

In [ ]:

In [ ]:

Follow us on Social Media

Linkedin: https://www.linkedin.com/company/eternaltek/about/?viewAsMember=true

Medium: https://medium.com/@eternaltek.info

WhatsApp Channel: https://whatsapp.com/channel/0029Va5onCbDjiOTi6D1vU36

In [ ]: