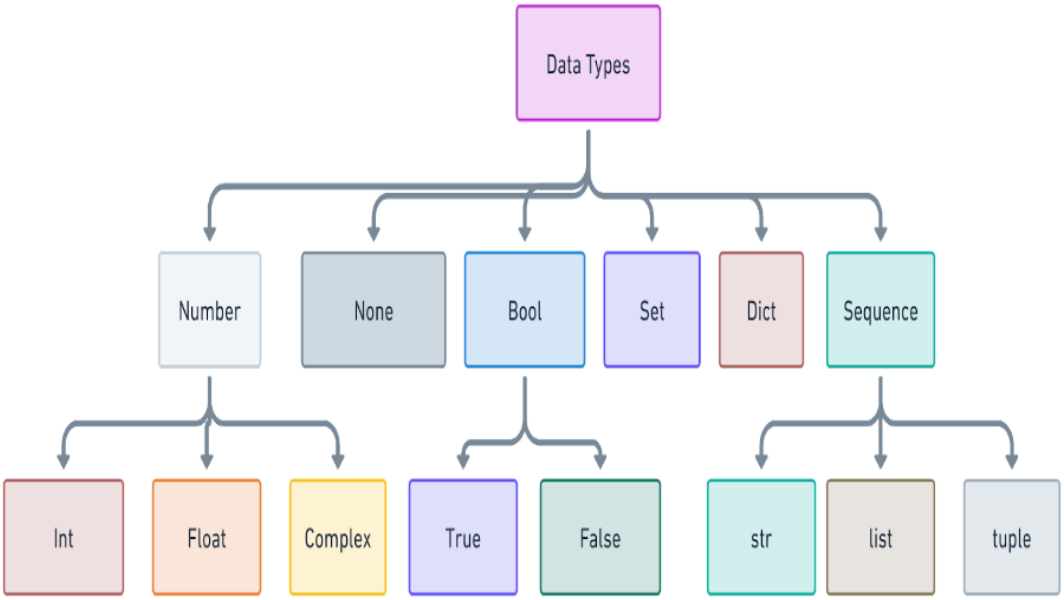
Data types in python

Lesson Plan

PW SKILLS

Data Types in Python

Data Types in Python :



1. Number :

- int
- float
- complex

Integers (int): Integers are whole numbers, both positive and negative, without any decimal point. For example: 5, -10, 1000.

Floating-point numbers (float): Floating-point numbers represent real numbers with a decimal point or an exponent. They can represent both integer and fractional parts of a number. For example: 3.14, 2.71828, -0.5.

Complex numbers (complex): Complex numbers consist of a real part and an imaginary part, represented as a + bj, where a is the real part and b is the imaginary part. In Python, j is used to represent the imaginary unit. For example: 3 + 2j, -1 - 4j, 5j.

Inbuilt Functions in Python

1. abs(x) :

- The `abs()` function returns the absolute value of a number.
- It takes a single argument `x`, which can be an integer, floating-point number, or complex number.
- For real numbers, `abs(x)` returns the distance of `x` from zero on the number line, always returning a non-negative value.
- For complex numbers, `abs(x)` returns the magnitude (or modulus) of the complex number, which is the distance from the origin in the complex plane.

Example:

```
abs_value = abs(-5)
print(abs_value)  # Output: 5
```

2. type(obj) :

- The `type()` function returns the type of an object.
- It takes a single argument `obj`, which can be any Python object, such as a variable, data structure, or instance of a class.
- The return value is a type object that represents the data type of the object.

Example:

```
type_int = type(5)
type_float = type(3.14)
type_str = type('Hello')
print(type_int)   # Output: <class 'int'>
print(type_float)  # Output: <class 'float'>
print(type_str)   # Output: <class 'str'>
```

3. pow(x, y, z=None) :

- The `pow()` function returns the value of `x` raised to the power of `y`.
- It optionally takes a third argument `z`, representing the modulus (i.e., the result is computed as `(x y) % z`)
- If `z` is not specified, the standard behavior is to return `x` raised to the power of `y`.

Example:

```
result = pow(2, 3)
print(result)  # Output: 8

result_mod = pow(2, 3, 5)
print(result_mod)  # Output: 3 (since 2^3 = 8, and 8 % 5 = 3)
```

4. round(number[, ndigits]) :

- The `round()` function rounds a number to a specified precision.
- It takes two arguments: `number` is the number to be rounded, and `ndigits` (optional) is the number of decimal places to round to. If `ndigits` is not provided, the number is rounded to the nearest integer.
- When `ndigits` is positive, it specifies the number of decimal places to round to. When `ndigits` is negative, it specifies the number of digits to the left of the decimal point to round to.

Example:

```
rounded_num = round(3.14159, 2)
print(rounded_num)  # Output: 3.14

rounded_num_negative = round(1234, -2)
print(rounded_num_negative)  # Output: 1200
```

5. min(iterable, *iterables[, key]) :

- The `min()` function returns the smallest item from an iterable or multiple iterables.
- It can take one or more iterables (separated by commas) as arguments and returns the smallest item among all the items in those iterables.
- Optionally, you can provide a `key` function to customize the comparison.

Example:

```
smallest = min([5, 3, 8, 1, 9])
print(smallest)  # Output: 1

smallest_letter = min('hello')
print(smallest_letter)  # Output: 'e'
```

6. max(iterable, *iterables[, key]) :

- The `max()` function returns the largest item from an iterable or multiple iterables.
- It can take one or more iterables (separated by commas) as arguments and returns the largest item among all the items in those iterables.
- Optionally, you can provide a `key` function to customize the comparison.

Example:

```
largest = max([5, 3, 8, 1, 9])
print(largest)  # Output: 9

largest_letter = max('hello')
print(largest_letter)  # Output: 'o'
```

```
# Creating int, float, complex variale
int_var = 25
float_var = 32.5
complex_var = 6+9j

#checking the varibale types
print(type(int_var))
print(type(float_var))
print(type(complex_var))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

2. Bool :

The bool type represents Boolean values, which can either be True or False. Boolean values are commonly used in control statements like if, while, and for loops to control the flow of the program based on certain conditions.

- True

```
a = True
b = False
print(type(a))
print(type(b))
```

```
<class 'bool'>
<class 'bool'>
```

- False

3. Set

In Python, a set is an unordered collection of unique elements. Sets are mutable, meaning you can add or remove items from them, but they do not support indexing or slicing like lists or tuples. Sets are commonly used for tasks like removing duplicates from a list or performing set operations like union, intersection, difference, etc.

1. Definition :

- A set in Python is an unordered collection of unique elements.
- It is a mutable data structure that allows for efficient membership testing and eliminating duplicate values.
- Sets are enclosed within curly braces `{ }` and consist of comma-separated elements.

Example:

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

Explanation:

- In this example, `my_set` is a set containing five elements: `1`, `2`, `3`, `4`, and `5`.
- Sets automatically eliminate duplicate values, so if duplicates are provided, only unique elements are retained in the set.

2. Uniqueness :

- Sets contain only unique elements. Duplicate values are automatically removed when creating a set.

Example:

```
# Set with duplicate elements
my_set = {1, 2, 2, 3, 3, 3}
```

Explanation:

- Despite the duplicate values (`2` and `3`) in the provided elements, the resulting set `my_set` will only contain unique elements (`1`, `2`, and `3`), eliminating duplicates automatically.

3. Mutable :

- Sets in Python are mutable, meaning you can add or remove elements after creation.
- However, the elements themselves must be immutable.

Example:

```
# Adding elements to a set
my_set.add(6)

# Removing elements from a set
my_set.remove(3)
```

Explanation:

- In this example, we demonstrate the mutability of sets by adding an element (`6`) using the `add()` method and removing an element (`3`) using the `remove()` method.

4. Unordered :

- Sets are unordered collections, meaning they do not maintain any specific order of elements.
- The order of elements in a set is undefined and can change between iterations.

Example:

```
# Set with elements in a different order
my_set = {5, 4, 3, 2, 1}
```

Explanation:

- Although elements are defined in a specific order (`5`, `4`, `3`, `2`, `1`), sets do not preserve this order when iterating or printing.
- Elements in a set are arranged for optimized retrieval and membership testing, rather than maintaining a specific sequence.

5. Membership Testing :

- Sets are efficient for membership testing, allowing you to quickly check whether a value is present in the set.

Example:

```
# Membership testing
if 5 in my_set:
    print("5 is present in the set")
```

Explanation:

- In this example, we use membership testing with the `in` keyword to check if `5` is present in the set `my_set`.
- Since `5` is indeed present in the set, the condition evaluates to `True`, and the corresponding message is printed.

Overall, sets in Python provide a flexible and efficient way to work with collections of unique elements, offering capabilities for membership testing, eliminating duplicates, and performing set operations such as union, intersection, and difference.

```
set_var = {6,7,8}
print(type(set_var))
```

```
<class 'set'>
```

4. Dict

A dictionary is an unordered collection of key-value pairs. Each key in a dictionary must be unique, and it maps to a corresponding value. Dictionaries are mutable, meaning you can add, modify, or delete key-value pairs.

1. Definition :

- A dictionary in Python is an unordered collection of key-value pairs.
- It is a versatile and powerful data structure used to store and retrieve data efficiently.
- Dictionaries are enclosed within curly braces `{ }` and consist of comma-separated key-value pairs.

Example:

```
# Creating a dictionary
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
```

Explanation:

- In this example, `my_dict` is a dictionary containing three key-value pairs.
- Each key-value pair is separated by a colon (`:`), with the key on the left and the corresponding value on the right.
- Keys (`'name'`, `'age'`, `'city'`) are unique identifiers for accessing the associated values (`'John'`, `30`, `'New York'`).

2. Key-Value Pairs :

- Each element in a dictionary is a key-value pair, where the key is a unique identifier and the value is the associated data.
- Keys must be immutable objects, such as strings, numbers, or tuples (containing immutable elements).
- Values can be of any data type.

Example:

```
# Dictionary with different data types as values
person_info = {'name': 'Alice', 'age': 25, 'is_student': False, 'grades': [85, 90, 95]}
```

Explanation:

- In this example, `person_info` is a dictionary containing various types of data as values.
- The keys (`'name'`, `'age'`, `'is_student'`, `'grades'`) are strings, serving as unique identifiers.
- The values associated with each key can be strings (`'Alice'`), integers (`25`), booleans (`False`), or even lists (`[85, 90, 95]`).

3. Mutable :

- Dictionaries in Python are mutable, allowing for modification after creation.
- You can add, update, or remove key-value pairs as needed.

Example:

```
# Adding a new key-value pair
person_info['email'] = 'alice@example.com'

# Updating an existing value
person_info['age'] = 26

# Removing a key-value pair
del person_info['is_student']
```

Explanation:

- In this example, we demonstrate the mutability of dictionaries by adding, updating, and removing key-value pairs.
- The `['email'] = 'alice@example.com` statement adds a new key-value pair to the `person_info` dictionary.
- The `['age'] = 26` statement updates the value associated with the key `age`.
- The `del person_info['is_student']` statement removes the key-value pair with the key `is_student` from the dictionary.

4. Unordered :

- Dictionaries are unordered collections, meaning they do not maintain any specific order of insertion.
- However, starting from Python 3.7, dictionaries maintain the order of insertion.

Example:

```
# Dictionary with keys in a different order
my_dict = {'b': 2, 'a': 1, 'c': 3}
```

Explanation:

- Although keys are defined in a specific order (`b`, `a`, `c`), dictionaries do not guarantee that order when iterating or printing.

5. Sequence

In Python, a sequence is an ordered collection of elements. Lists, tuples, and strings are all examples of sequences. Sequences support various operations like indexing, slicing, concatenation, and repetition.

- String

1. Indexing

Indexing in Python can be done from both sides of a string. Let's delve into how indexing works from both the left and right sides:

Indexing (left to right):

- In Python, indexing starts from 0 for the first character from the left side of the string.
- You can access individual characters of a string using positive indices.
- **Example:**

```
my_string = "Hello"
print(my_string[0])   # Output: 'H'
print(my_string[1])   # Output: 'e'
```

Here, `my_string[0]` accesses the first character 'H', and `my_string[1]` accesses the second character 'e'.

Indexing (right to left):

- Strings in Python can be thought of as sequences of characters. Each character in a string has an index associated with it, starting from 0 for the first character from the left.
- Negative indexing allows accessing characters from the end of the string, with -1 representing the last character, -2 representing the second last character, and so on.
- **Example:**

```
my_string = "Hello"
print(my_string[0])   # Output: 'H'
print(my_string[-1])  # Output: 'o'
```

Here, `my_string[0]` accesses the first character 'H', and `my_string[-1]` accesses the last character 'o'.

2. Immutable:

- In Python, strings are immutable, meaning once a string is created, its content cannot be changed.
- While you can access individual characters or slices of a string, you cannot directly modify the characters in the string.
- **Example:**

```
my_string = "Hello"
my_string[0] = 'J'  # Raises TypeError: 'str' object does not support item assignment
```

Attempting to change a character in the string, as shown in the example, raises a TypeError.

3. Saves space:

- Python optimizes memory usage for strings internally. Strings are stored efficiently, and memory allocation and deallocation are managed dynamically by Python's memory management system.
- Example:

Memory-saving techniques are implemented internally by Python and are not directly observable in code.

4. Concatenation:

- Concatenation is the process of combining strings together.
- In Python, strings can be concatenated using the `+` operator.
- Example:

```
str1 = "Hello"
str2 = "World"
concatenated_string = str1 + " " + str2
print(concatenated_string)  # Output: 'Hello World'
```

Here, `str1 + " " + str2` joins the two strings with a space in between.

5. Length:

- The length of a string can be obtained using the `len()` function, which returns the number of characters in the string.
- Example:

```
my_string = "Hello"
print(len(my_string))  # Output: 5
```

The length of the string "Hello" is 5.

6. Slice:

- Slicing allows you to extract a portion of a string by specifying start and stop indices.
- The syntax for slicing is `str[start:stop:step]`.
- **Example:**

```
my_string = "Hello World"
print(my_string[0:5])  # Output: 'Hello'
```

Here, `my_string[0:5]` extracts the substring from index 0 to index 4 (5 is exclusive), resulting in 'Hello'.

7. str[:]:

- Using `str[:]` creates a copy of the entire string.
- It's equivalent to slicing with no start or stop indices specified.
- Example:

```
my_string = "Hello"
copy_of_string = my_string[:]
print(copy_of_string)  # Output: 'Hello'
```

Here, `my_string[:]` creates a copy of the string "Hello".

8. Upper:

- The `upper()` method converts all characters in a string to uppercase.
- Example:

```
my_string = "hello"
print(my_string.upper())  # Output: 'HELLO'
```

The string "hello" is converted to uppercase.

9. Lower:

- The `lower()` method converts all characters in a string to lowercase.
- Example:

```
my_string = "HELLO"
print(my_string.lower())  # Output: 'hello'
```

The string "HELLO" is converted to lowercase.

10. Capitalize:

- The `capitalize()` method capitalizes the first character of a string and converts the rest to lowercase.
- Example:

```
my_string = "hello world"
print(my_string.capitalize())  # Output: 'Hello world'
```

The first character 'h' is capitalized, resulting in 'Hello world'.

11. Strip:

- The `strip()` method removes leading and trailing whitespace (spaces, tabs, newlines) from a string.
- Example:

```
my_string = "  Hello  "
print(my_string.strip())  # Output: 'Hello'
```

The leading and trailing spaces are removed from the string.

12. Escape character:

- Escape characters in Python are used to represent characters that are difficult or impossible to type directly.
- Examples include `\n` for newline, `\t` for tab, `\'` for single quote, `\"` for double quote, etc.
- Example:

```
my_string = "This is a\nmultiline\nstring."
print(my_string)  # Output:
# This is a
# multiline
# string.
```

The escape sequence `\n` is used to represent newlines in the string.

• List :-

In Python, a list is a built-in data structure used to store collections of items. It is one of the most versatile and commonly used data structures in Python. Here's a concise definition:

- **List :** A list is an ordered, mutable collection of items, where each item is separated by a comma and enclosed within square brackets `[ ]`.

In more detail:

1. Ordered: Lists maintain the order of elements as they are inserted. The order of elements in a list is preserved, meaning you can access elements by their index, and they will always appear in the same sequence.

2. Mutable: Lists are mutable, which means you can modify them after creation. You can change, add, or remove elements from a list as needed, making them very flexible for dynamic data manipulation.

3. Heterogeneous : Lists can contain elements of different data types, including integers, floats, strings, booleans, and even other lists or complex objects. This allows you to create lists with diverse sets of data.

4. Dynamic : Lists in Python are dynamic arrays, which means they automatically resize to accommodate new elements. You can append new elements to the end of a list, insert elements at specific positions, or remove elements from the list.

5. Iterable : Lists are iterable, meaning you can loop over them using loops like `for` loops or list comprehensions. This makes it easy to iterate through all the elements in a list for processing or manipulation.

6. Negative indexing:
- Negative indexing allows accessing elements from the end of the list.
- Index -1 refers to the last element, -2 refers to the second last element, and so on.

Lists:

1. Append :
- The `append()` method adds an element to the end of a list.
- Example:

```
my_list = [1, 2, 3]
my_list.append(4)
```

Explanation: In this example, we have a list `[1, 2, 3]`. The `append()` method is called on this list with the argument `4`. After the operation, `my_list` becomes `[1, 2, 3, 4]`, with `4` added at the end of the list.

2. Insert :
- The `insert()` method inserts an element at a specified position in a list.
- Example:

```
my_list = [1, 2, 3]
my_list.insert(1, 5)
```

Explanation: Here, we have a list `[1, 2, 3]`. The `insert()` method is called with arguments `(1, 5)`, meaning we want to insert `5` at index `1`. After the operation, `my_list` becomes `[1, 5, 2, 3]`, with `5` inserted at index `1`.

3. Remove :
- The `remove()` method removes the first occurrence of a specified value from a list.
- Example:

```
my_list = [1, 2, 3, 2]
my_list.remove(2)
```

Explanation: In this example, we have a list `[1, 2, 3, 2]`. The `remove()` method is called with the argument `2`. It removes the first occurrence of `2` from the list. After the operation, `my_list` becomes `[1, 3, 2]`.

4. Pop :

- The `pop()` method removes and returns the element at a specified index in a list.
- Example:

```
my_list = [1, 2, 3]
popped_element = my_list.pop(1)
```

Explanation: Here, we have a list `[1, 2, 3]`. The `pop()` method is called with the argument `1`, meaning we want to remove and return the element at index `1`. After the operation, `my_list` becomes `[1, 3]`, and `popped_element` contains the value `2`.

5. Reverse :

- The `reverse()` method reverses the elements of a list in place.
- Example:

```
my_list = [1, 2, 3]
my_list.reverse()
```

Explanation: In this example, we have a list `[1, 2, 3]`. The `reverse()` method is called, which reverses the order of elements in the list in place. After the operation, `my_list` becomes `[3, 2, 1]`.

Now, let's cover the characteristics and operations for tuples.

Tuple Characteristics:-

1. Ordered :

- Tuples, like lists, are ordered collections.
- The order of elements in a tuple is maintained.

2. Immutable :

- Tuples are immutable, meaning once they are created, their content cannot be changed.
- You cannot add, remove, or modify elements in a tuple.

Operations on Tuples:

1. Type :

- Tuples are defined using parentheses `( )`.
- Example:

```
my_tuple = (1, 2, 3)
```

Explanation: Here, we define a tuple `my_tuple` containing the elements `1`, `2`, and `3` enclosed within parentheses.

2. Count :

- The `count()` method returns the number of occurrences of a specified value in a tuple.
- Example:

```
my_tuple = (1, 2, 2, 3)
count = my_tuple.count(2)
```

Explanation: In this example, we have a tuple `(1, 2, 2, 3)`. The `count()` method is called with the argument `2`, which returns the number of occurrences of `2` in the tuple. In this case, `count` would be `2`.

3. Length :

- The `len()` function returns the number of elements in a tuple.
- Example:

```
my_tuple = (1, 2, 3)
length = len(my_tuple)
```

Explanation: Here, we have a tuple `(1, 2, 3)`. The `len()` function is called with `my_tuple` as an argument, which returns the number of elements in the tuple. In this case, `length` would be `3`.

4. Accessing Elements :

- Elements in a tuple are accessed using indexing.
- Example:

```
my_tuple = (1, 2, 3)
element = my_tuple[1]
```

Explanation: In this example, we have a tuple `(1, 2, 3)`. We access the element at index `1` using square brackets, which returns the value `2` and assigns it to the variable `element`.

5. Slice :

- Slicing allows you to extract a portion of a tuple by specifying start and stop indices.
- Example:

```
my_tuple = (1, 2, 3, 4, 5)
slice_tuple = my_tuple[1:4]
```

Explanation: Here, we have a tuple `(1, 2, 3, 4, 5)`. We use slicing to extract a portion of the tuple from index `1` to index `3` (excluding index `4`). This creates a new tuple `slice_tuple` containing elements `(2, 3, 4)`.

6. None

None is a special constant in Python that represents the absence of a value or a null value. It is often used to indicate that a variable or result has no value assigned or returned.

None:

1. Definition :

- `None` is a special constant in Python that represents the absence of a value or a null value.
- It is often used to denote that a variable or a function returns nothing, or that a value is missing or undefined.

2. Purpose :

- `None` serves as a placeholder or default value when you need to initialize a variable or return a value that has no meaningful content.
- It can be used to indicate that a function doesn't return anything explicitly, or that a variable has not been assigned a value yet.

3. Type :

- `None` is a unique singleton object in Python and is of the type `NoneType`.
- It is the only instance of the `NoneType` class.

4. Comparison :

- You can use the `is` operator to check if a variable or expression is `None`.
- Unlike other values like `0`, `False`, or empty strings, `None` is its own distinct object and cannot be equated with other values using comparison operators like `==`.

5. Common Usages :

- Initializing variables: It is common to initialize variables to `None` if their values will be assigned later.

```
x = None
```

- Return value: Functions that do not return any meaningful value often return `None` by default.

```
def do_something():
    # code that does something
    return None
```

- Placeholder for missing values: `None` can be used to represent missing or undefined values in data structures or function parameters.

```
def process_data(data):
    if data is None:
        print("No data provided")
    else:
        # process data
        pass
```

6. Behavior :

- When used in boolean contexts, `None` evaluates to `False`.
- However, `None` is not equivalent to `False` in terms of identity. That is, `None` is not the same object as `False`.

7. Idiomatic Usage :

- In Python, it is idiomatic to use `None` to signify the absence of a value or to indicate that a function does not return anything explicitly.
- Using `None` in such cases makes the code more readable and expressive, as it clearly communicates the intent to other developers.

```
var = None
print(type(var))
```

```
<class 'NoneType'>
```