Basic Operations in Python (Part 2)

In this tutorial, we'll cover basic operations in Python, including operations on `str` (strings), `int` (integers), `float` (floating-point numbers), `bool` (Booleans), `tuple`, `list`, and `dict` (dictionaries).

Operations on Strings

Strings in Python are sequences of characters and can be defined using either single quotes (`'`) or double quotes (`"`). You can concatenate strings using the `+` operator, repeat strings using the `*` operator, and access individual characters using indexing (e.g., `string[0]` returns the first character in the string).

```
# Concatenating strings
name = "John" + " " + "Doe"
print(name) # Output: "John Doe"


# Repeating strings
greeting = "Hello" * 3
print(greeting) # Output: "HelloHelloHello"

# Indexing strings
name = "John Doe"
print(name[0]) # Output: "J"
```

You can also use various string methods to manipulate strings, such as `len()` to get the length of a string, `lower()` to convert a string to lowercase, and `upper()` to convert a string to uppercase.

```
name = "John Doe"

# Getting the length of a string
print(len(name)) # Output: 8

# Converting to lowercase
print(name.lower()) # Output: "john doe"
```

```
# Converting to uppercase
print(name.upper()) # Output: "JOHN DOE"
```

Operations on Integers and Floating-Point Numbers

Integers and floating-point numbers are basic numerical data types in Python. You can perform various arithmetic operations on numbers, including addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulo (`%`).

```
# Addition
sum = 2 + 3
print(sum) # Output: 5

# Subtraction
difference = 5 - 2
print(difference) # Output: 3

# Multiplication
product = 2 * 3
print(product) # Output: 6

# Division
quotient = 6 / 3
print(quotient) # Output: 2.0

# Modulo
remainder = 5 % 2
print(remainder) # Output: 1
```

Operations on Booleans

Booleans are data types that can have one of two values: `True` or `False`. You can perform various logical operations on Booleans, including `and`, `or`, and `not`.

```
# Logical AND
print(True and False) # Output: False

# Logical OR
print(True or False) # Output: True

# Logical NOT
print(not True) # Output: False
```

Operations on Tuples

Tuples are ordered, immutable sequences of elements, which can be of different data types. You can access elements of a tuple using indexing (e.g., `tuple[0]`), and you can perform various operations on tuples, such as concatenation (`+`) and repetition (`*`).

```
# Creating a tuple
person = ("John", "Doe", 30)
```

```
# Accessing elements of a tuple
print(person[0]) # Output: "John"

# Concatenating tuples
names = ("Jane", "Doe") + ("John", "Doe")
print(names) # Output: ("Jane", "Doe", "John", "Doe")

# Repeating tuples
greeting = ("Hello",) * 3
print(greeting) # Output: ("Hello", "Hello", "Hello")
```

Operations on Lists

Lists are ordered, mutable sequences of elements, which can be of different data types. You can perform various operations on lists, such as indexing (e.g., `list[0]`), concatenation (`+`), repetition (`*`), and membership testing (`in`).

```
# Creating a list
names = ["Jane", "Doe", "John", "Doe"]

# Accessing elements of a list
print(names[0]) # Output: "Jane"

# Concatenating lists
names = ["Jane", "Doe"] + ["John", "Doe"]
print(names) # Output: ["Jane", "Doe", "John", "Doe"]

# Repeating lists
greeting = ["Hello"] * 3
print(greeting) # Output: ["Hello", "Hello", "Hello"]

# Membership testing
print("John" in names) # Output: True
```

You can also use various list methods to manipulate lists, such as `len()` to get the length of a list, `append()` to add an element to the end of a list, and `remove()` to remove an element from a list.

```
# Getting the length of a list
print(len(names)) # Output: 4

# Adding an element to the end of a list
names.append("Jane")
print(names) # Output: ["Jane", "Doe", "John", "Doe", "Jane"]

# Removing an element from a list
names.remove("Jane")
print(names) # Output: ["Doe", "John", "Doe"]
```

Operations on Dictionaries

Dictionaries are unordered, mutable collections of key-value pairs. You can access the value associated with a key in a dictionary using indexing (e.g., `dict[key]`), and you can perform various operations on dictionaries, such as adding a new key-value pair (`dict[key] = value`) and testing for membership (`in`).

```
# Creating a dictionary
person = {"first_name": "John", "last_name": "Doe", "age": 30}

# Accessing values in a dictionary
print(person["first_name"]) # Output: "John"

# Adding a new key-value pair
person["gender"] = "Male"
print(person) # Output: {'first_name': 'John', 'last_name': 'Doe', 'age': 30, 'gender': 'Male

# Membership testing
print("first_name" in person) # Output: True
```

Comparison, Identity, and Membership Operators

Python provides several operators for comparing and testing the type and contents of variables. These operators include comparison, identity, and membership operators.

Comparison Operators

Comparison operators are used to compare the values of two variables. The following table lists the comparison operators in Python:

- `==` Equal
- `!=` Not equal
- `>` Greater than
- `<` Less than
- `>=` Greater than or equal to
- `<=` Less than or equal to

For example, we can use the `==` operator to compare two values:

```
x = 3
y = 4
print(x == y) # Output: False
```

Identity Operators

Identity operators are used to compare the memory addresses of two variables. The following table lists the identity operators in Python:

- `is` True if the variables are the same object
- `is not` True if the variables are not the same object

For example, we can use the `is` operator to compare the memory addresses of two values:

```
x = [1, 2, 3]
y = [1, 2, 3]
```

```
print(y is x) # Output: False

z = x
print(z is x) # Output: True
```

In this example, `x` and `y` are two separate lists with the same values, so their memory addresses are different.

Membership Operators

Membership operators are used to test if a value is a member of a sequence, such as a list, tuple, or string. The following table lists the membership operators in Python:

- `in` True if the value is found in the sequence
- `not in` True if the value is not found in the sequence

For example, we can use the `in` operator to check if a value is a member of a list:

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

In this example, the value `3` is a member of the list `numbers`, so the expression `3 in numbers` returns `True`.