Python

OPERATORS AND DATA STRUCTURES

ROMIL NANDWANA

CONTENTS

INTRODUCTION:

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example, in the 3 + 3 = 6 operation , the numbers on the left hand side of the = sign, 3 and 3, are the operands while + is the operator and 6 is the output of the operation.

Data structures are a way of organizing and storing data so that they can be accessed and worked with efficiently. They define the relationship between the data, and the operations that can be performed on the data. There are many various kinds of data structures defined that make it easier for the data scientists and the computer engineers, alike to concentrate on the main picture of solving larger problems rather than getting lost in the details of data description and access.

Basic Operators in Python

1. **Arithmetic operators:** Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

OPERATOR	DESCRIPTION	SYNTAX
+	Addition: adds two operands	x + y
-	Subtraction: subtracts two operands	x - y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when first operand is divided by the second	x % y

```
# Examples of Arithmetic Operator
a = 9  b = 4
 # Addition of numbers
add = a + b
# Subtraction of numbers
sub = a - b
# Multiplication of number
mul = a * b
# Division(float) of number
div1 = a / b
# Division(floor) of number
div2 = a // b
# Modulo of both number
mod = a % b

 # print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
```

OUTPUT:

```
13
5
36
2.25
2
1
```

2. **Relational Operators:** Relational operators compares the values. It either returns **True** or **False** according to the condition.

OPERATOR	DESCRIPTION	SYNTAX
>	Greater than: True if left operand is greater than the right	x > y
<	Less than: True if left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to: True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to: True if left operand is less than or equal to the right	x <= y

```
# Examples of Relational Operators
a = 13
b = 33

# a > b is False
print(a > b)

# a < b is True
print(a < b)

# a == b is False
print(a == b)

# a != b is True
print(a != b)

# a >= b is False
print(a >= b)

# a <= b is True
print(a <= b)
```

OUTPUT:

```
False

True

False

True

False

True
```

3. **Logical operators:** Logical operators perform **Logical AND**, **Logical OR** and **Logical NOT** operations.

OPERATOR	DESCRIPTION	SYNTAX
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

```
# Examples of Logical Operator
a = True
b = False

# Print a and b is False
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(not a)
```

Output:

```
False

True

False
```

4. **Bitwise operators:** Bitwise operators acts on bits and performs bit by bit operation.

OPERATOR	DESCRIPTION	SYNTAX
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Examples of Bitwise operators

```
a = 10
b = 4
print(a & b)
print(a | b)
print(~a)
print(a ^ b)
print(a >> 2)
print(a << 2)
```

Output:

```
0
14
-11
14
2
40
```

5. **Assignment operators:** Assignment operators are used to assign values to the variables.

OPERATOR	DESCRIPTION	SYNTAX
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right side operand with left side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a*=b a=a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b a=a<<b

6. **Identity operators-**

   is and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

is	True if the operands are identical
is not	True if the operands are not identical

```
# Examples of Identity operators
a1 = 3
b1 = 3
a2 = 'GeeksforGeeks'
b2 = 'GeeksforGeeks'
a3 = [1,2,3]
b3 = [1,2,3]


print(a1 is not b1)


print(a2 is b2)

# Output is False, since lists are mutable.
print(a3 is b3)
```

Output:

```
False

True

False
```

7. **Membership operators-**

   in and **not in** are the membership operators; used to test whether a value or variable is in a sequence.

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

Examples of Membership operator

```
x = 'Geeks for Geeks'
y = {3:'a',4:'b'}

print('G' in x)

print('geeks' not in x)

print('Geeks' not in x)

print(3 in y)

print('b' in y)
```

Output:

```
True

True

False

True

False
```

Data Structures in Python

How to Create a List?

List is a collection of elements that is ordered and mutable (i.e., its values can be changed after creation).

Elements of the list are enclosed in square brackets `[]`. You can create a list in Python using square brackets separated by commas and assigning the same to a variable.

Example: `scores = [80, 90, 95, 100]`
Let's say you want to change the value at the second index, i.e., 95 to 99; very simple. Use `scores[2] = 99` and the value changes. But the memory location doesn't change, i.e., it is modified in-place.

```
# initial list
scores = [80,90,95,100]

# memory location of initial list
print('Memory loaction initially is ' , id(scores))
print('='*20)
# change value
scores[2] = 99
# display modified list
print(scores)
print('='*20)
# check memory loaction of modified list
print('Memory location of modified list is ' , id(scores))
```

Output

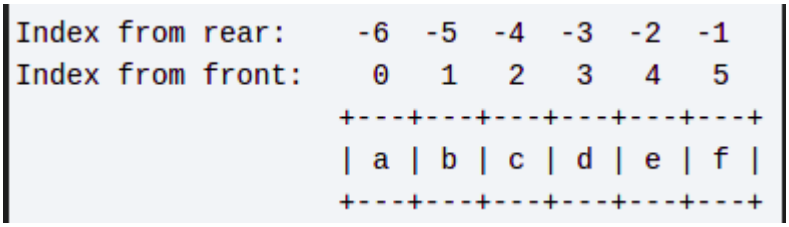
```
Memory loaction initially is  139678449711304
====================
[80, 90, 99, 100]
====================
Memory location of modified list is  139678449711304
```

Indexing

Elements in a list can be accessed using their `index` in the list. `Index` specifies the place of an element in the list. There are two ways to do this which is the same as we saw in strings:

- Forward indexing
- Backward indexing

The following diagram will give you a visual intuition of how a list is in indexed in Python



Indexing and Slicing Lists

The syntax for slicing lists is similar to that of strings:

- `list_object[start:end:step]`

Here, start and end are indices (start inclusive, end exclusive). All slicing values are optional.

Other Common List Operations

There are several operations that can be performed on lists in Python. In this section, we will learn about a few that you will frequentlywould use frequently going forward in this program. Let us take two lists `a = [1,2,3],b = [4,5,6].`

Operation	Description	Example
`len(a)`	Length	3
`a + b`	Concatenation	`[1,2,3,4,5,6]`
`[a]*2`	Repitition	`[1,2,3,1,2,3]`
`3 in a`	Membership	`True`
`max(a)`	Returns item from the list with max value	3
`min(a)`	Returns item from the list with min value	1
`a.append(10)`	Adds `10` to the end of the list	`[1,2,3,10]`
`a.extend(b)`	Appends the contents of `b` to `a`	`[1,2,3,4,5,6]`
`a.index(1)`	Returns the lowest index in the list that `1` appears in	0
`a.reverse()`	Reverses objects of list in place	`[3,2,1]`
`a.pop()`	Removes and returns last object from list	3
`a.remove(2)`	Removes object `2` from list	`[1,3]`

What are Dictonaries?

Dictonaries are similar to hash or maps in other languages. It consists of key value pairs where value can be accessed by unique key in the dictionary.. **But what's the difference between lists and dictionaries?**

Unique Features

- Lists are **ordered** sets of objects, whereas dictionaries are **unordered** sets.
- **Items in dictionaries are accessed via keys and not via their position.**
- Since values of a dictionary can be any Python data type, **so dictionaries are unordered key-value-pairs.**
- Dictionaries don't support the sequence operation of the sequence data types like strings, tuples and lists.
- They belong to the built-in mapping type. They are the sole representative of this kind!

Creating a Dictionary

```
>>> myDictionary = {"key1": "value1", "key2":"value2", "key3": "value3"}
>>> keys = myDictionary.keys()
>>>
>>> print(keys)
dict_keys(['key2', 'key3', 'key1'])
>>>
>>> myDictionary["newKey"] = "newValue"
>>>
>>> print(keys)
dict_keys(['key2', 'key3', 'newKey' 'key1'])
```

In the figure above,

- A dictionary is enclosed in `{}`.
- Every element in the dictionary has two components namely the `key`, and `value`. In the above example "`key1`", "`key2`", "`key3`" are the keys and "`value1`", "`value2`", and "`value3`" are their respective values.

- We can access the values by making use of these keys; for example, typing `myDictionary["key1"]` will give us "`value1`".
- We can have different data types for different values in a dictionary.
- Use the `.keys()` method to look at all the keys as shown.
- To add a new key-value pair, just look at the third line in the image. A new key named "`newKey`" and its value "`newValue`" is created by `myDictionary["newKey"] = "newValue"`.

Dictionary keys are mutable (can't be lists), and all the keys are unique (no duplicates allowed).

Dictionary Operations

To understand dictionary operations we will use the following dictionary:

`area = { 'living' : [400, 450], 'bedroom' : [650, 800], 'kitchen' : [300, 250], 'garage' : [250, 0]}.`

- **Accessing Values in a Dictionary**

  We can access the values within a dictionary using keys and perform various operations on it.

  Example: If you need to access areas of bedrooms, and you have already have seen that `bedroom` is a key in the dictionary `area`, you can access it using `area['bedroom']`.
  Another method by which you can accomplish the same thing is `dict.get(key, default_value)`. Here, you will search for `key` in the keys of `dict` and, if present, it returns the value associated with that key. And if the `key` is absent, then the `default_value` gets displayed. Show below is an example carried out on `area`:

```
area = { 'living' : [400,450], 'bedroom' : [650, 800], 'kitchen' : [300, 250],
'garage': [250, 0]}
print(area.get('living'))
print('='*50)
print (area.get('color','This key is absent' ))
```

Output

```
[400, 450]
==================================================
This key is absent
```

- **Updating Values in a Dictionary**

  We can update a dictionary by adding a new entry or a key-value pair, or modifying an existing entry

  Example:

  - To update the value of `living` to `[400,500]`, we can simply use `area['living'] = [400, 500]`
  - Suppose now we also have area of `garden`, which needs to be updated in the dictionary, we can use `area['garden'] = [200, 250]`.
- **Deleting Values in a Dictionary**

  We can delete values for a particular `key` of a dictionary using the `del`command
  Example: To delete the key value pair for `'garage'`, we can use `del area['garage']`
- **Updating a Dictionary**

  Now you want the dictionary `area` to have another key `color` and the `['red', 'blue']` list. This process is called updating the dictionary and can be achieved
  by **`dict1.update(dict2)`** where `dict1` and `dict2` are two dictionaries.

```
area = { 'living' : [400,450], 'bedroom' : [650, 800], 'kitchen' : [300, 250],
'garage': [250, 0]}
d2 = {'color' :['red','blue']}
area.update(d2)
print(area)
```

Output

```
{'living': [400, 450], 'bedroom': [650, 800], 'kitchen': [300, 250], 'garage
```

Creating a Tuple

Tuples are another type similar to lists. They store values of any type separated by commas. But how do they differ from lists?

- Firstly, they start and end with parenthesis `()`. For example, (1,2,3,4).
- But the main difference that **unlike lists, tuples are immutable i.e. they cannot be updated/changed.** You can think of them as **read-only** lists.

As discussed above, it should begin and end with parenthesis.

Taking the same example of weights from our discussion on `lists`,

```
scores = (80,90,95,100)
```

The only difference is that now we cannot change the values inside it.

Common Tuple Operations

Let us take two tuples `a = (1,2)` and `b = (3,4)` for this purpose.

Expression	Description	Example
`len(a)`	Length of tuple	2
`a + b`	Concatenation	`(1,2,3,4)`
`a*4`	Repitition	`(1,2,1,2,1,2,1,2)`
`2 in a`	Membership	`True`
`max(a)`	Maximum value in tuple	2
`min(a)`	Minimum value in tuple	1
`a[0]`	Indexing	1
`a[:2]`	Slicing	`(1,2)`

You cannot delete an element from tuple (as they are immutable), but you can delete the entire tuple.

CONCLUSION:

You Did It!

Hurray! You reached the end of this tutorial! This gets you one topic closer to understand the fundamentals of python.