



Python Datatypes

Tuesday, December 2022

*“An investment in knowledge pays
the best interest”*

- BENJAMIN FRANKLIN

Today's Agenda

- **Concatenating Strings**
- **The **Slice Operator** In Strings**
- **Operators In Python**

String Concatenation

- **Example:**

```
s1="Good"  
s2="Morning"  
s3=s1+s2  
print(s3)
```

- **Output:**
GoodMorning

- **Example:**

```
s1="Good"  
s2="Morning"  
s3=s1+" "+s2  
print(s3)
```

- **Output:**
Good Morning

The Slicing Operator

- Slicing means pulling out a sequence of characters from a string .
- For example , if we have a string “**Industry**” and we want to extract the word “**dust**” from it , then in **Python** this is done using slicing.
- To slice a string , we use the operator[] as follows:
- **Syntax: s[x:y]**
- **x** denotes the **start index** of slicing and **y** denotes the **end index** . But **Python** ends slicing at **y-1** index.

The Slicing Operator

- Example:

```
s="Industry"  
print(s[2:6])
```

- Example:

```
s="Welcome"  
print(s[3:6])
```

The Slicing Operator

- Example:

```
s="Mumbai"  
print(s[0:3])
```

- Example:

```
s="Mumbai"  
print(s[0:10])
```

The Slicing Operator

- Example:

```
s="Python"  
print(s[2:2])
```

- Example:

```
s="Python"  
print(s[6:10])
```


The Slicing Operator

- Example:

```
s="welcome"  
print(s[1:])
```

- Example:

```
s="welcome"  
print(s[:3])
```

The Slicing Operator

- Example:

```
s="welcome"  
print(s[:])
```

- Example:

```
s="welcome"  
print(s[])
```

The Slicing Operator

- Example:

```
s="welcome"  
print(s[-4:-1])
```

- Example:

```
s="welcome"  
print(s[-1:-4])
```

Using Step Value

- String slicing can accept a **third parameter** also after the two index numbers.
- The **third parameter** is called **step value**.
- So the complete syntax of slicing operator is:

s[begin:end:step]

- Step value indicates *how many characters to move forward after the first character is retrieved* from the string and its default value is **1**, but can be changed as per our choice.

The Slicing Operator

- **For Example:**

```
s="Industry"  
print(s[2:6])
```

- **Can also be written as :**

```
s="Industry"  
print(s[2:6:1])
```

- **Example:**

```
s="Industry"  
print(s[2:6:2])
```

Operators

- **Operators** are special symbols in that carry out different kinds of **computation** on values.
- **For example : $2+3$**
- In the expression $2+3$, **+** is an operator which performs addition of **2** and **3**, which are called **operands**

Types Of Operators In Python

- Arithmetic operators
- Comparison operators or Relational operators
- Logical operators
- Assignment operators
- Identity operators
- Membership operators

Operator in python

- **Operators** are special symbols in **Python** or in any other language which can manipulate the value of **operands**.
- The value that the **operator** operates on is called the operand.
- For example: here $2+3=5$. Here, $+$ is the **operator** that performs addition and 2 and 3 represent the operands.

Arithmetic Operator

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example	
+	Addition	$x + y$	
-	Subtraction	$x - y$	
*	Multiplication	$x * y$	
/	Division	x / y	
%	Modulus	$x \% y$	
**	Exponentiation	$x ** y$	
//	Floor division	$x // y$	

Arithmetic Operator

```
x = 5
```

```
y = 3
```

```
print(x + y)
```

```
print(x - y)
```

```
print(x * y)
```

```
print(x/y)
```

```
Print(x%y)#It's used to get the remainder of a division problem.
```

```
print(x**y)
```

```
print(x//y)#the floor division // rounds the result down to the nearest whole number
```

Relational Operators In Python

- Relational Operators
- Relational Operators With Strings
- Chaining Of Relational Operators
- Special Behavior Of == and !=

Relational Operators In Python

Relational operators are used to **compare** values.

They either return **True** or **False** according to the condition.

These operators are:

Operator	Meaning
>	Greater Than
<	Less Than
>=	Greater Than Equal To
<=	Less Than Equal To
==	Equal To
!=	Not Equal To

The 6 Basic Relational Operators

```
a=10
```

```
b=4
```

```
print("a=",a,"b=",b)
```

```
print("a > b",a>b)
```

```
print("a < b",a<b)
```

```
print("a==b",a==b)
```

```
print("a!=b",a!=b)
```

```
print("a>=b",a>=b)
```

```
print("a<=b",a<=b)
```

Relational Operators With Strings

Relational Operators can also work with **strings**.

When applied on **string operands**, they compare the **unicode** of corresponding characters and return **True** or **False** based on that comparison.

As discussed previously, this type of comparison is called **lexicographical comparison**

Relational Operators With Strings

```
a="Ramesh"  
b="Rajesh"  
print("a=",a,"b=",b)  
print("a > b",a>b)  
print("a < b",a<b)  
print("a==b",a==b)  
print("a!=b",a!=b)  
print("a>=b",a>=b)  
print("a<=b",a<=b)
```

Relational Operators With Strings

If we want to check the **UNICODE** value for a particular letter , then we can call the function **ord()**.

It is a built in function which accepts **only one character** as argument and it returns the **UNICODE** number of the argument passed

Example:

`ord('A')`

`ord('m')`

`ord('j')`

Relational Operators With Strings

```
a= "BHOPAL"  
b= "bhopal"  
print("a=",a,"b=",b)  
print("a > b",a>b)  
print("a < b",a<b)  
print("a==b",a==b)  
print("a!=b",a!=b)  
print("a>=b",a>=b)  
print("a<=b",a<=b)
```

Will This Code Run ?

```
a=True
b=False
print("a=",a,"b=",b)
print("a > b",a>b)
print("a < b",a<b)
print("a==b",a==b)
print("a!=b",a!=b)
print("a>=b",a>=b)
print("a<=b",a<=b)
```

**Yes , the code will successfully
Run because True is 1 and False is 0**

What about this code?

```
a='True'  
b='False'  
print("a=",a,"b=",b)  
print("a > b",a>b)  
print("a < b",a<b)  
print("a==b",a==b)  
print("a!=b",a!=b)  
print("a>=b",a>=b)  
print("a<=b",a<=b)
```

Yes , this code will also successfully
Run but 'True' and 'False' will be handled as strings

Special Behavior Of Relational Operators

Python allows us to **chain** multiple **relational operators** in one **single statement**.

For example the expression **1<2<3** is perfectly valid in **Python**

However when **Python** evaluates the expression , it returns **True** if **all individual conditions are true** , otherwise it returns **False**

Cascading Of Relational Operators

- **Example:**
`print(7>6>5)`
- **Example:**
`print(5<6>7)`
- **Example:**
`print(5>6>7)`
- **Example:**
`print(5<6<7)`

Special Behavior Of == And !=

== compares its **operands** for **equality** and if they are of **compatible types** and **have same value** then it returns **True** otherwise it returns **False**

Similarly **!=** compares its **operands** for **inequality** and if they are of **incompatible types** or **have different value** then it returns **True** otherwise it returns **False**

Special Behavior Of == And !=

- Example:
`print(10==10)`
- Example:
`print(10==20)`
- Example:
`print(10=="10")`
- Example:
`print(10==True)`

- Example:
`print(1==True)`
- Example:
`print("A"=="A")`
- Example:
`print("A"=="65")`
- Example:
`print("A"==65)`

Special Behavior Of

== And !=

- Example:
`print(15==15.0)`

- Example:
`print(15==15.01)`

- Example:
`print(15!="15")`

- Example:
`print(0 != False)`

- Example:
`print(False!=True)`

- Example:
`print(False != 0.0)`

- Example:
`print(2+5j==2+5j)`

- Example:
`print(2+5j!= 2)`

Logical Operators In Python

Logical operators are used to combine **two or more conditions** and perform the logical operations using **Logical and**, **Logical or** and **Logical not**.

Operator	Meaning
and	It will return true when both conditions are true
or	It will returns true when at-least one of the condition is true
not	If the condition is true, logical NOT operator makes it false

Behavior Of Logical **and** Operator

```
>>> a=40
>>> b=20
>>> c=50
>>> a>b and a>c
False
```

```
>>> a=40
>>> b=20
>>> c=50
>>> a>b or a>c
True
```

```
>>> a=40
>>> b=20
>>> c=50
>>> a>b and c>a
True
```

```
>>> a=40
>>> b=20
>>> c=50
>>> b>a or b>c
False
```

Behavior Of Logical Operators With Non Boolean



Python allows us to apply logical operators with **non boolean types** also

But before we understand how these operators work with **non boolean** types, we must understand some very important points

Behavior Of Logical Operators With Non Boolean

1. **None**, **0**, **0.0**, **""** are all **False** values
2. The return value of **Logical and** & **Logical or operators** is never **True** or **False** when they are applied on **non boolean** types.

Behavior Of Logical Operators With Non Boolean

3. If the **first value** is **False** , then **Logical and** returns **first value** , otherwise it returns the **second value**
4. If the **first value** is **True** , then **Logical or** returns **first value** , otherwise it returns the **second value**
5. When we use **not operator** on **non boolean** types , it returns **True** if it's operand is **False**(in any form) and **False** if it's operand is **True** (in any form)

Logical Operators On Non Boolean Types

- Example:
5 and 6
- Example:
5 and 0
- Example:
0 and 10
- Example:
6 and 0

- Example:
'Sachin' and 10
- Example:
'Sachin' and 0
- Example:
'Indore' and 'Bhopal'
- Example:
'Bhopal' and 'Indore'

Logical Operators On Non Boolean Types

- Example:
0 and 10/0
- Example:
10/0 and 0
- Example:
5 or 6
- Example:
5 or 0
- Example:
0 or 10
- Example:
6 or 0
- Example:
'Sunny' or 10
- Example:
'Sunny' or 0
- Example:
'Indore' or 'Bhopal'

Logical Operators On Non Boolean Types

- Example:
0 or 10/0
- Example:
10/0 or 0
- Example:
not 5
- Example:
not 0
- Example:
not 'Sunny'
- Example:
not "

Assignment Operators In Python

The **Python Assignment Operators** are used to assign the values to the declared variables.

Equals (=) operator is the most commonly used assignment operator in Python.

For example:

- **a=10**

Assignment Operators In Python

Shortcut for assigning same value to all the variables

- `x=y=z=10`

Shortcut for assigning different value to all the variables

- `x,y,z=10,20,30`

Guess The Output

```
a,b,c=10,20
```

```
print(a,b,c)
```

Output:

ValueError : Not enough values to unpack

```
a,b,c=10,20,30,40
```

```
print(a,b,c)
```

Output:

ValueError : Too many values to unpack

Compound Assignment Operators

Python allows us to combine **arithmetic operators** as well as **bitwise operators** with assignment operator.

For example: The statement

- **`x=x+5`**

Can also be written as

- **`x+=5`**

Compound Assignment Operators

Operator	Example	Meaning
+=	$x+=5$	$x=x+5$
-=	$x-=5$	$x=x-5$
=	$x=5$	$x=x*5$
/=	$x/=5$	$x=x/5$
%=	$x\%=5$	$x=x\%5$
//=	$x//=5$	$x=x//5$
=	$x=5$	$x=x**5$
&=	$x\&=5$	$x=x\&5$
!=	$x!=5$	$x=x!5$
^=	$x^=5$	$x=x^5$
>>=	$x>>=5$	$x=x>>5$
<<=	$x<<=5$	$x=x<<5$

Guess The Output

```
a=10
```

```
print(++a)
```

Output:

10

```
a=10
```

```
print(a++)
```

Output:

SyntaxError : Invalid Syntax

Conclusion:

Python does not has any **increment operator** like **++**.

Rather it is solved as **+(+x)** i.e **+(+10)** which is **10**

However the expression **a++** is an error as it doesn't make any sense

Guess The Output

```
a=10
```

```
print(--a)
```

Output:

10

```
a=10
```

```
print(a--)
```

Output:

SyntaxError : Invalid Syntax

Conclusion:

Python does not have any **decrement operator** like **--**.

Rather it is solved as **-(-x)** i.e **-(-10)** which is **10**

However the expression **a--** is an error as it doesn't make any sense

Guess The Output

```
a=10
```

```
print(++++a)
```

Output:

10

```
a=10
```

```
print(-----a)
```

Output:

-10

Identity Operators

Identity operators in Python are **is** and **is not**

They serve 2 purposes:

- To verify if two **references** point to the **same memory location** or not

AND

- To determine whether a **value** is of a **certain class** or **type**

Behavior Of **is** and **is not**

The operator **is** returns **True** if the operands are **identical**, otherwise it returns **False**.

The operator **is not** returns **True** if the operands are **not identical**, otherwise it returns **False**.

Guess The Output

a=2

b=3

c=a is b

print(c)

Explanation:

Since **a** and **b** are pointing
to **2 different objects**, so
the operator **is** returns **False**

a=2

b=2

c=a is b

print(c)

Explanation:

Since **a** and **b** are pointing
to **same objects**, so
the operator **is** returns **True**

Examples Of **is** Operator

```
a=2
```

```
b=type(a) is int
```

```
print(b)
```

Output:

True

Explanation:

type(a) is int evaluates to **True** because

2 is indeed an **integer** number.

```
a=2
```

```
b=type(a) is float
```

```
print(b)
```

Output:

False

Explanation:

type(a) is float evaluates to **False**

because 2 is not a **float** number.

Examples Of **is not** Operator

a="Delhi"

b="Delhi"

c=a **is not** b

print(c)

Output:

False

Explanation:

Since **a** and **b** are pointing

to the **same object**, so

the operator **is not** returns **False**

a="Delhi"

b="delhi"

c=a **is not** b

print(c)

Output:

True

Explanation:

Since **a** and **b** are pointing

to **2 different objects**, so

the operator **is not** returns **True**

Membership Operators

Membership operators are used to test whether a value or variable is found in a sequence (**string**, **list**, **tuple**, **set** and **dictionary**).

There are 2 **Membership operators**

- **in**
- **not in**

Behavior Of **in** and **not in**

in: The '**in**' operator is used to check if a value exists in a sequence or not

not in: The '**not in**' operator is the opposite of '**in**' operator. So, if a value does not exists in the sequence then it will return a **True** else it will return a **False**.

Examples Of **in** Operator

```
a="Welcome"
```

```
b="om"
```

```
print(b in a)
```

Output:

True

```
a="Welcome"
```

```
b="mom"
```

```
print(b in a)
```

Output:

False

Examples Of **not in** Operator

```
primes=[2,3,5,7,11]
```

```
x=4
```

```
print(x not in primes)
```

Output:

True

```
primes=[2,3,5,7,11]
```

```
x=5
```

```
print(x not in primes)
```

Output:

False

Precedence Of Operators

There can be more than one operator in an expression.

To evaluate these type of expressions there is a rule called **precedence** in all programming languages .

It guides the order in which operation are carried out.

Precedence And Associativity

Operator	Name
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor div, Mod
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Guess The Output

```
a=6/2+3**4
```

```
print(a)
```

Output:

84.0

```
a=20-12//3**2
```

```
print(a)
```

Output:

19

```
a=25/(2+3)**2
```

```
print(a)
```

Output:

1.0

Associativity Of Operators

When two operators have the same precedence, **Python** follows **associativity**

Associativity is the order in which an expression is evaluated and almost all the operators have **left-to-right associativity**.

For example, **multiplication** and **division** have the same precedence. Hence, if both of them are present in an expression, **left one is evaluates first**.

Guess The Output

```
a=5*2//3
```

```
print(a)
```

Output:

3

```
a=5*(2//3)
```

```
print(a)
```

Output:

0

Guess The Output

```
a=2**3**2
```

```
print(a)
```

Output:

512

```
a=(2**3)**2
```

```
print(a)
```

Output:

64



Remember, **
has Right to left
associativity

Exercise

- **WAP to accept two numbers from the user and display their sum**

Code:

```
a=int(input("Enter first num:"))
```

```
b=int(input("Enter second num:"))
```

```
c=a+b
```

```
print("Nos are",a,"and",b)
```

```
print("Their sum is",c)
```


Exercise

- **WAP to accept radius of a Circle from the user and calculate area and circumference**

Code:

```
radius=float(input("Enter radius:"))
```

```
area=3.14*radius**2
```

```
circum=2*3.14*radius
```

```
print("Area is",area)
```

```
print("Circumference is",circum)
```

Accepting Different Values

- **WAP to accept roll number , grade and percentage as input from the user and display it back**

Code

```
roll=int(input("Enter roll no:"))  
name=input("Enter name:");  
per=float(input("Enter per:"))  
print("Roll no is",roll)  
print("Name is",name)  
print("Per is",per)
```

Exercise

- Write a program that asks the user to enter his/her name and age. Print out a message , displaying the user's name along with the year in which they will turn 100 years old.

```
What is your name ? Sachin  
How old are you ? 36  
Hello Sachin  
You will be 100 years old in the year 2082
```

Accepting Multiple Values In One Line

In **Python** , the **input()** function can read and return a complete line of input as a string.

However , we can split this input string into individual values by using the function **split()** available in the class **str**

The function **split()** , breaks a string into multiple strings by using **space** as a separator

Accepting Multiple Values In One Line

To understand , working of **split()** , consider the following example:

```
text="I Love Python"
```

```
word1,word2,word3=text.split()
```

```
print(word1)
```

```
print(word2)
```

```
print(word3)
```

Accepting Multiple Values In One Line

```
text=input("Type a 3 word message")
```

```
word1,word2,word3=text.split()
```

```
print("First word",word1)
```

```
print("Second word",word2)
```

```
print("Third word",word3)
```

An Important Point!

The number of variables on left of assignment operator and number of values generated by **split()** must be the same

Exercise

Write a program that asks the user to input 2 integers and adds them . Accept both the numbers in a single line only

```
Enter 2 numbers:10 20  
First number is 10  
Second number is 20  
Their sum is 30
```


Solution

Code:

```
s=input("Enter 2 numbers:")
```

```
a,b=s.split()
```

```
print("First number is",a);
```

```
print("Second number is",b)
```

```
c=int(a)+int(b)
```

```
print("Their sum is",c)
```

Accepting Multiple Values Separated With ,

By default **split()** function considers , space as a separator

However , we can use any other symbol also as a separator if we pass that symbol as argument to **split()** function

For example , if we use comma , as a separator then we can provide comma separated input

Example

Code:

```
s=input("Enter 2 numbers separated with comma:")
```

```
a,b=s.split(",")
```

```
print("First number is",a);
```

```
print("Second number is",b)
```

```
c=int(a)+int(b)
```

```
print("Their sum is",c)
```

```
Enter 2 numbers separated with comma:2,4
First number is 2
Second number is 4
Their sum is 6
```

Accepting Different Values In One Line

Code:

```
s=input("Enter roll no,name and per:")  
  
roll,name,per=s.split()  
  
print("Roll no is",roll)  
  
print("Name is",name)  
  
print("Per is",per)
```

```
Enter roll no,name and per:10 Sachin 78.9  
Roll no is 10  
Name is Sachin  
Per is 78.9
```

Thank you