BASIC OPERATORS IN PYTHON

What are operators?

Operators are used to perform operations on values and variables. Operators can manipulate individual items and returns a result. The data items are referred as operands or arguments. Operators are either represented by keywords or special characters.

Python Language supports these operators:

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Logical Operators
4. Assignment Operators
5. Bitwise Operators
6. Membership Operators (*will be covered later)*
7. Identity Operators (*will be covered later)*

Arithmetic Operators:

Arithmetic operators are used to perform mathematical operations.

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
**	Exponentiation: Raise the first operand to the second	x ** y

Examples:

```
In [1]: 23 + 89
Out[1]: 112
In [2]: 89 + 90.134
Out[2]: 179.13400000000001
In [3]: print(89 + 90.134)
Out[3]: 179.13400000000001
In [4]: 32 * 98
Out[4]: 3136
In [5]: 87 - 321.98
Out[5]: -234.98000000000002
In [6]: 89.78 – 12
Out[6]: 77.78
In [7]: 56 / 7
Out[7]: 8.0
In [8]: 56 / 9
Out[8]: 6.222222222222222
In [9]: 54//7
Out[9]: 7
In [10]: 54//7.0
Out[10]: 7.0
In [11]: 54//7.2
Out[11]: 7.0
In [12]: 16 ** 4
Out[12]: 65536
In [13]: 24**2.0
Out[13]: 576.0
In [14]: 49**0.5
Out[14]: 7.0
```

You would find interesting when we added 89 and 90.134 we should have gotten the result 179.134 but we got something else. Why?? The reason is "**Floating Point Error**" (Follow this link to know more.)

Relational Operators:

These operators are used to compare values. It either returns `True` or `False` according to the condition.

OPERATOR	MEANING	EXAMPLE
>	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:

```
In [24]: 34 > 54
Out[24]: False
In [25]: 78 < 101
Out[25]: True
In [26]: 23 == 23.0
Out[26]: True
In [27]: 34 == 23
Out[27]: False
In [28]: 23 != 23.1
Out[28]: True
```

```
In [29]: 45 != 45
Out[29]: False
In [30]: 34 >= 34
Out[30]: True
In [31]: 23 >= 16
Out[31]: True
In [32]: 67 <= 78
Out[32]: True
```

*Ready for some mindf*ck?*

```
In [34]: 0.1 + 0.1 == 0.2
Out[34]: True
In [34]: 0.1 + 0.1 + 0.1 == 0.3
Out[34]: False
```

^try it out yourself and try to guess why this happens. Btw, this is not python specific; this plagues nearly every language that has floating point arithmetic.

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:

```
In [34]: 0 and 1
Out[34]: False
In [35]: 0 or 4
Out[35]: True
In [36]: not 2
Out[36]: False
```

Bitwise Operators:

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.

For example, 2 is `10` in binary and 7 is `111`.

In the table below: Let `x` = 10 (`0000 1010` in binary) and `y` = 4 (`0000 0100` in binary)

Operator	Meaning	Example
&	Bitwise AND	x& y = 0 (`0000 0000`)
|	Bitwise OR	x | y = 14 (`0000 1110`)
~	Bitwise NOT	~x = -11 (`1111 0101`)
^	Bitwise XOR	x ^ y = 14 (`0000 1110`)
>>	Bitwise right shift	x>> 2 = 2 (`0000 0010`)
<<	Bitwise left shift	x<< 2 = 40 (`0010 1000`)

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

Optional Study Material:

Membership Operators:

`in` and `not in` are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Identity Operators:

`is` and `is not` are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Precedence of Python Operators:

The combination of values, variables, operators and function calls is termed as an expression. Python interpreter can evaluate a valid expression. To evaluate these types of expressions there is a rule of precedence in Python. It guides the order in which operation are carried out. The operator precedence in Python are listed in the following table. It is in descending order; upper group has higher precedence than the lower ones.

Operators	Meaning
()	Parentheses
******	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
***, /, //, %**	Multiplication, Division, Floor division, Modulus
+, -	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

PYTHON DATA TYPES AND VARIABLES

Overview:

Python has five standard Data Types:

- Numbers
- Strings
- Lists
- Tuples
- Dictionary

Python sets the variable type based on the value that is assigned to it. Unlike more rigorous languages, Python will change the variable type if the variable value is set to another value. For example:

```
var = 123 # This will create a number integer assignment
var = 'john' # the `var` variable is now a string type.
```

Numbers

Python numbers variables are created by the standard Python method:

```
var = 382
```

Most of the time using the standard Python number type is fine. Python will automatically convert a number from one type to another if it needs. But, under certain circumstances that a specific number type is needed (i.e. complex, hexadecimal), the format can be forced into a format by using additional syntax in the table below:

Generally, Python will do variable conversion automatically. You can also use Python conversion casting (int(), long(), float(), complex()) to convert data from one type to another. In addition, the type function returns information about how your data is stored within a variable.

TYPE	FORMAT	DESCRIPTION
Int	a = 10	SignedInteger
Long	a =345L	(L) Long integers, they can also be represented in octal and hexadecimal
Float	a = 45.67	(.) Floating point real values
Complex	a = 3.14j	(j) Contains integer in the range 0 to 255.

```
message = "Good morning"
num = 85
pi = 3.14159
i = 0 + 1j
print(type(message))  # This will return string

print(type(n))  # This will return integer

print(type(pi))  # This will return float

print(type(i))  # This will return complex
```

String

Create string variables by enclosing characters in quotes. Python uses single quotes `'` double quotes `"` and triple quotes `"""` to denote literal strings. Only the triple quoted strings `"""` will automatically continue across the end of line statement.

```
firstName = 'py'

lastName = "god"

message = """This is a string that will span across multiple lines. Using newline characters

and no spaces for the next lines. The end of lines within this string also count as a newline
when printed"""
```

Strings can be accessed as a whole string, or a substring of the complete variable using brackets '[]'. Here are a couple examples:

```
var1 = 'Hello World!'

var2 = 'CRUxPython'

print (var1[0]) # this will print the first character in the string an `H`

print (var2[1:5]) # this will print the substring 'RUxP`
```

Python can use a special syntax to format multiple strings and numbers. The string formatter is quickly covered here because it is seen often and it is important to recognize the syntax.

```
print("The item {} is repeated {} times".format(element,count))

print(f"The item{element} is repeated {count} times.")  # new in python3.6
```

The `{}` are placeholders that are substituted by the variables `element` and `count` in the final string. This compact syntax is meant to keep the code more readable and compact.

Python is currently transitioning to the format syntax given above, python can use an older syntax, which is being phased out, but is still seen in some example code:

```
print ("The item %i is repeated %i times"% (element, count))
```

List

Lists are a very useful variable type in Python. A list can contain a series of values. List variables are declared by using brackets `[ ]` following the variable name.

```
A = [ ] # This is a blank list variable

B = [1, 23, 45, 67] # this list creates an initial list of 4 numbers.

C = [2, 4, 'pranjal'] # lists can contain different variable types.
```

All lists in Python are zero-based indexed. When referencing a member or the length of a list the number of list elements is always the number shown plus one.

```
mylist = ['Rhino', 'Grasshopper', 'DoFlamingo', 'Bongo']

B = len(mylist) # This will return the length of the list which is 3. The index is 0, 1, 2, 3.

print (mylist[1]) # This will return the value at index 1, which is 'Grasshopper'

print (mylist[0:2]) # This will return the first 3 elements in the list.
```

You can assign data to a specific element of the list using an index into the list. The list index starts at zero. Data can be assigned to the elements of an array as follows:

```
mylist = [0, 1, 2, 3]

mylist[0] = 'Rhino'

mylist[1] = 'Grasshopper'

mylist[2] = 'DoFlamingo'

mylist[3] = 'Bongo'

print (mylist[1])  # Grasshopper
```

Lists aren't limited to a single dimension. Although most people can't comprehend more than three or four dimensions. You can declare multiple dimensions by separating a with commas. In the following example, the mytable variable is a two-dimensional array:

```
mytable = [[], []]
```

In a two-dimensional array, the first number is always the number of rows; the second number is the number of columns.

For a detailed look at managing lists, take a look at the article

Python Lists - Google developer
TutorialsPoint Python Lists

Tuple

Tuples are a group of values like a list and are manipulated in similar ways. But, tuples are fixed in size once they are assigned. In Python the fixed size is considered immutable as compared to a list that is dynamic and mutable. Tuples are defined by parentheses ().

```
mytuple = ('Rhino', 'Grasshopper', 'DoFlamingo', 'Bongo')
```

Here are some advantages of tuples over lists:

1. Elements cannot be added to or removed from a tuple. They are immutable.
2. Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.
3. It makes your code safer if you "write-protect" data that does not need to be changed.

Dictionary

Dictionaries in Python are lists of `Key` : `Value` pairs. This is a very powerful data type to hold a lot of related information that can be associated through `keys` . The main operation of a dictionary is to extract a value based on the `key` name. Unlike lists, where index to numbers are used, dictionaries allow the use of a `key` access its members.

Dictionaries can also be used to sort, iterate and compare data.

Dictionaries are created by using braces ({}) with pairs separated by a comma (,) and the key values associated with a colon(:). In Dictionaries the `key` must be unique.
Here is a quick example on how dictionaries might be used:

```
room_num = {'john': 425, 'tom': 212}

room_num['john'] = 645  # set the value associated with the 'john' key to 645

print (room_num['tom']) # print the value of the 'tom' key.

room_num['isaac'] = 345 # Add a new key 'isaac' with the associated value

print (room_num.keys()) # print out a list of keys in the dictionary

print ('isaac' in room_num) # test to see if 'isaac' is in the dictionary. This returns True.
```

PYTHON VARIABLES

Overview

A variable is a convenient placeholder that refers to a computer memory location where you can store program information that may change during the time your script is running
When a variable is stored in memory, the interpreter will allocate a certain amount of space for each variable type.

In Python, variables are always one of the five fundamental data types:

- Numbers
- String
- List
- Tuple
- Dictionary

While each variable has its own properties and methods, there are common methods we use to deal with all variables in Python.

Declaration

In Python, variables are created the first time a value is assigned to them. For example:

```
number = 10

string = "This is a string"
```

You declare multiple variables by separating each variable name with a comma. For example:

```
a, b = True, False
```

This is the same the multiple line declaration of:

```
a = True
b = False
```

Naming Restrictions

Variable names follow the standard rules for naming anything in Python. A variable name:

- Must begin with an alphabetic character (A -Z) or an underscore (_).
- Cannot contain a period(.), @, $, or %.
- Must be unique in the scope in which it is declared.
- Python is case sensitive. So "selection" and "Selection" are two different variables.

Best practices for all Python naming can be found in the (Style Guide for Python Naming Conventions)

Scope & Lifetime

Scope of a variable defines where that variable can be accessed in your code. For instance, a `global` variable can be accessed from anywhere in your code. A `local` variable can only be accessed within the function it was declared in. Generally, a variable's scope is determined by where you declare it.

When you declare a variable within a procedure, only code within that procedure can access or change the value of that variable. It has local scope and is a procedure-level variable. If you declare a variable outside a procedure, you make it recognizable to all the procedures in your script. This is a global variable, and it has global scope.

Here are few examples:

```
global_var = True


def function1():
    local_var = False
    print (global_var)
    print (local_var)

function1() # this runs the function
print (global_var) # this works because global_var is accessible
print (local_var)  # this gives an error because we are outside function1
```

It is important to be careful when declaring variables. It is easy to create duplicate variable names that do not reference the correct values. For instance, do not declare a global variable this way:

```
g_var = 'True'
def function2():
    g_var = 'False'
    print ('inside the function var is ', g_var)

ex2()
print ('outside the function var is ', g_var)
```

The example above will create a `Global` variable named `g_var`. When dropping in the `function2` function, there will be a second `local` variable created named `g_var` with a different value. The proper way to work with a global variable is to be very explicit with the `global` statement in the `local` scope:

```
g_var = "Global"
def function2():
    g_var = "Local"
    print ('inside the function var is ', g_var)
    return;

function2()
```

```
print ('outside the function var is ', g_var)
```

The *lifetime* of a variable depends on how long it exists. The lifetime of a `global` variable extends from the time it is declared until the time the script is finished running. Local Variables stay as long as the functional unit (such as a *for loop or a function*) is working.

Assigning Values

Values are assigned to variables creating an expression as follows: the variable is on the left side of the expression and the value you want to assign to the variable is on the right. For example:

```
B = 200
```

The same value can be assigned to multiple variables at the same time:

```
a = b = c = 1
```

And multiple variables can be assigned different values on a single line:

```
a, b, c = 1, 2, "john"
```

This is the same as:

```
a = 1
b = 2
c = "john"
```

Scalar Variables & Lists

Much of the time, you only want to assign a single value to a variable you have declared. A variable containing a single value is a scalar variable. Other times, it is convenient to assign more than one related value to a single variable. Then you can create a variable that can contain a series of values. This is called a list variable. List variables and scalar variables are declared in the same way, except that the declaration of an array variable uses brackets `[ ]` following the variable name.

```
A = [ ] # This is a blank list variable
B = [1, 23, 45, 67] # this list creates an initial list of 4 numbers.
C = [2, 4, 'john'] # lists can contain different variable types.
```