# Python Operator

## Python Arithmetic Operator

These Python arithmetic operators include Python operators for basic mathematical operations.

**Addition(+)**

|  |
| --- |
| >>> 3+4 # 7 |

**Subtraction(-)**

|  |
| --- |
| >>> 3-4 # -1 |

**Multiplication(\*)**

|  |
| --- |
| >>> 3\*4  # 12 |

**Division(/)**

|  |
| --- |
| >>> 3/4  # 0.75 |

**Exponentiation(\*\*)**

|  |
| --- |
| >>> 3\*\*4  # 81 |

**Floor Division(//)**

|  |
| --- |
| >>> 10//3  # 3 |

**Modulus(%)**

|  |
| --- |
| >>> 4%3  # 1 |

## Python Relational Operator

Relational Python Operator carries out the comparison between operands. They tell us whether an operand is greater than the other, lesser, equal, or a combination of those.

**Less than(<)**

|  |
| --- |
| >>> 3<4  # True |

**Greater than(>)**

|  |
| --- |
| >>> 3>4  # False |

**Less than or equal to(<=)**

|  |
| --- |
| >>> 7<=7  # True |

****Greater than or equal to(>=)****

|  |
| --- |
| >>> 0>=0  # True |

**Equal to(= =)**

|  |
| --- |
| >>> 3==3.0  >>> 1==True  >>> 0==False  # True |

**f. Not equal to(!=)**

|  |
| --- |
| >>> 1!=1.0  # False |

## Python Assignment Operator

An assignment operator assigns a value to a variable. It may manipulate the value by a factor before assigning it. We have 8 assignment operators- one plain, and seven for the 7 arithmetic python operators.

**Assign(=)**

|  |
| --- |
| >>> a=7  >>> print(a)  # 7 |

**Add and Assign(+=)**

|  |
| --- |
| >>> a+=2  >>> print(a)  # 9 |

**Subtract and Assign(-=)**

|  |
| --- |
| >>> a-=2  >>> print(a)  # 7 |

**Divide and Assign(/=)**

|  |
| --- |
| >>> a/=7  >>> print(a)  # 1.0 |

**Multiply and Assign(\*=)**

|  |
| --- |
| >>> a\*=8  >>> print(a)  # 8.0 |

**Modulus and Assign(%=)**

|  |
| --- |
| >>> a%=3  >>> print(a)  # 2.0 |

## Python Logical Operator

These are conjunctions that you can use to combine more than one condition. We have three Python logical operator – and, or, and not that come under python operators.

**and**

|  |
| --- |
| >>> a=7>7 and 2>-1  >>> print(a)  # False |

**or**

|  |
| --- |
| >>> a=7>7 or 2>-1  >>> print(a)  # True |

**not**

|  |
| --- |
| >>> a=not(0)  >>> print(a)  # True |

## Membership Python Operator

These operators test whether a value is a member of a sequence. The sequence may be a list, a string, or a tuple. We have two membership python operators- ‘in’ and ‘not in’.

**in**

|  |
| --- |
| >>> pets=[‘dog’,’cat’,’ferret’]  >>> ‘fox’ in pets  # False  >>> ‘cat’ in pets  # True |

**not in**

|  |
| --- |
| >>> ‘pot’ not in ‘disappointment’  # True |

## Python Identity Operator

Let us proceed towards identity Python Operator.

These operators test if the two operands share an identity. We have two identity operators- ‘is’ and ‘is not’.

**is**

|  |
| --- |
| >>> 2 is 20  # False |

**is not**

|  |
| --- |
| >>> 2 is not ‘2’  # True |

# Variables

Variables and data types in python as the name suggests are the values that vary. In a programming language, a variable is a memory location where you store a value. The value that you have stored may change in the future according to the specifications.

## Variable Definition & Declaration

Python has no additional commands to declare a variable. As soon as the value is assigned to it, the variable is declared.

|  |
| --- |
| >>> x = 10  # variable is declared as the value 10 is assigned to it. |

1. The variable name cannot start with a number. It can only start with a character or an underscore.
2. Variables in python are case sensitive.
3. They can only contain alpha-numeric characters and underscores.
4. No special characters are allowed.

# Data Types In Python

## Numerical Data Types

Numerical data type holds numerical value. In numerical data there are 4 sub types as well. Following are the sub-types of numerical data type:

* Integers
* Float
* Complex Numbers
* Boolean

## Integers

Integers are used to represent whole number values.

|  |
| --- |
| >>> x = 10  # it will be the integer as long as the value is a whole number. |

## Float

Float data type is used to represent decimal point values.

|  |
| --- |
| >>> x = 10.25  >>> y = 12.30 |

## Strings

Strings in python are used to represent unicode character values. Python does not have a character data type, a single character is also considered as a string.

We denote or declare the string values inside single quotes or double quotes. To access the values in a string, we use the indexes and square brackets.

|  |
| --- |
| >>> name = 'edureka'  >>> name[2]  # this will give you the output as 'u' |

****Operations using strings****

|  |
| --- |
| >>> name = 'edureka'  >>> name.upper()  # this will make the letters to uppercase  >>> name.lower()  # this will make the letters to lowercase  >>> name.replace('e') = 'E'  # this will replace the letter 'e' with 'E'  >>> name[1: 4]  # this will return the strings starting at index 1 until the index 4. |

## Lists

List is one of the four collection data type that we have in python. When we are choosing a collection type, it is important to understand the functionality and limitations of the collection. Tuple, set and dictionary are the other collection data type is python.

A list is ordered and changeable, unlike strings. We can add duplicate values as well. To declare a list we use the square brackets.

|  |
| --- |
| >>> mylist = [10,20,30,40,20,30, 'edureka']  # this will give you the output as 'u' |

****Accessing values from a list****

|  |
| --- |
| >>> mylist[2:6]  # this will get the values from index 2 until index 6. |

****Adding/Replacing values in a list****

|  |
| --- |
| >>> mylist[6] = 'python'    # this will replace the value at the index 6.    >>> mylist.append('edureka')    # this will add the value at the end of the list.    >>> mylist.insert(5, 'data science')    # this will add the value at the index 5. |

|  |  |
| --- | --- |
| **Method Name** | **Property** |
| clear() | removes all the elements from the list |
| copy() | returns a copy of the list |
| extend() | add the elements of the list to the end of the current list |
| count() | returns the number of elements of the specified value |
| index() | returns the index of the element |
| pop() | removes the element from the specified position |
| remove() | removes the item with the specified value |
| sort() | sorts the list |
| reverse() | returns the reversed list |

## **Tuples**

Tuple is a collection which is unchangeable or immutable. It is ordered and the values can be accessed using the index values. A tuple can have duplicate values as well. To declare a tuple we use the round brackets.

|  |
| --- |
| >>> mytuple = (10,10,20,30,40,50)  # to count the number of elements    >>> mytuple.count(10)  # the output will be 2  # to find the index    >>> mytuple.index(50)  # the output will be 5. since the index number at 50 is 5. |

## Sets

A set is a collection which is unordered, it does not have any indexes as well. To declare a set in python we use the curly brackets.

|  |
| --- |
| >>> myset = {10, 20 , 30 ,40, 50, 50} |

A set does not have any duplicate values, even though it will not show any errors while declaring the set, the output will only have the distinct values.

To access the values in a set we can either loop through the set, or use a membership operator to find a particular value.

|  |
| --- |
| >>> for x in myset:  >>> print(x)  #this will get all the values.  >>> 20 in myset  #this will return true if the value is in the set.  #to add a value in a set  >>> myset.add('edureka')  #to add multiple values in a list  >>> myset.update([ 10, 20, 30, 40, 50])  #to remove an item from a set  >>> myset.remove('edureka')  #we can use the discard or pop method to remove an item from a set as well.  >>> myset = {10, 20, 30}  >>> myset1 = {10,30,50}  >>> myset.issubset(myset1)  #this will return false  >>> myset.union(myset1)  #this will return a set with the union of the two sets. |

Other operations in a dictionary include the following.

|  |  |
| --- | --- |
| **Method Name** | **Property** |
| clear() | clears the items from a set |
| copy() | returns the copy of the set |
| difference() | returns a set with the difference of the two sets |
| isdisjoint() | returns if the sets have intersection |
| issubset() | returns if the set is a subset |
| symmetricdifference() | returns a set with the symmetric difference |
| update() | update the sets with union of the set |

## Dictionary

A dictionary is just like any other collection array in python. But they have key value pairs. A dictionary is unordered and changeable. We use the keys to access the items from a dictionary. To declare a dictionary, we use the curly brackets.

|  |
| --- |
| >>> mydictionary = { 'python': 'data science', 'machine learning' : 'tensorflow' , 'artificial intelligence': 'keras'}    >>> mydictionary['machine learning']    #this will give the output as 'tensorflow'    >>> mydictionary.get('python')    #this serves the same purpose to access the value. |

Since we are using the keys to access the items, they cannot be duplicate.The values can have duplicate items.

|  |
| --- |
| #adding a new value    >>> mydictionary['analysis'] = 'matplotlib'    #replacing a value    >>> mydictionary['analysis'] = 'pandas'    #deleting a value    >>> mydictionary.pop('analysis')    #remove() , del also serves the same purpose for deleting a value. |

Other operations in a dictionary include the following.

|  |  |
| --- | --- |
| **Method Name** | **Property** |
| copy() | returns a copy of the dictionary |
| clear() | clears the dictionary |
| items() | returns a list containing tuple of key value pairs |
| keys() | returns a list containing all the keys |
| update() | updates the dictionary with all the key-value pairs |
| values() | returns a list of all the values in a dictionary |
| setdefault() | returns the value of a specified key |

## **Range**

**Range is a data type which is mainly used when we are using a loop. Lets take an example to understand this.**

|  |
| --- |
| >>> for x in range(10):  >>> print(x)    #this will print the numbers from 0-10. Range will have the numbers from 0-10 |

# **Escape Sequences**

List of escape sequences available in Python 3.

****\newline****

|  |
| --- |
| >>> print("line1 \  >>> line2 \  >>> line3")  # line1 line2 line3 |

****\\****

|  |
| --- |
| >>> print("\\")  # \ |

****\'****

|  |
| --- |
| >>> print('\'')  # ' |

****\"****

|  |
| --- |
| >>> print("\"")  # " |

****\n****

|  |
| --- |
| >>> print("Hello \n World!")  # Hello  World! |

****\t****

|  |
| --- |
| >>> print("Hello \t World!")  # Hello World! |

# **String Interpolation**

**String Interpolation is the process of substituting values of variables into placeholders in a string, sounds like string concatenation right! But without using + or concatenation methods.**

**Let’s see how many ways string interpolation works in Python.**

* **%-formatting**
* **Str.format()**
* **f-strings**
* **Template Strings**

****%-formatting****

**It’s a feature provided by Python which can be accessed with a % operator. This is similar to printf style function in C.**

|  |
| --- |
| # Python program to demonstrate  # string interpolation    >>> n1 = 'Hello'  >>> n2 ='GeeksforGeeks'    # for single substitution  >>> print("Welcome to % s"% n2)    # for single and multiple substitutions () mandatory  >>> print("% s ! This is % s."%(n1, n2))  # Welcome to GeeksforGeeks  # Hello! This is GeeksforGeeks. |

****Str.format()****

str.format() is one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

|  |
| --- |
| # Python program to demonstrate  # string interpolation    >>> n1 = 'Hello'  >>> n2 ='GeeksforGeeks'    # for single substitution  >>> print('Hello, {}'.format(n1))    # for single or multiple substitutions  # let's say b1 and b2 are formal parameters  # and n1 and n2 are actual parameters  >>> print("{b1}! This is {b2}.".format(b1 = n1, b2 = n2))    # else both can be same too  >>> print("{n1}! This is {n2}.".format(n2 = n2, n1 = n1))  # Hello, Hello  # Hello! This is GeeksforGeeks.  # Hello! This is GeeksforGeeks. |

****f-strings****

**PEP 498 introduced a new string formatting mechanism known as Literal String Interpolation or more commonly as F-strings (because of the leading f character preceding the string literal). The idea behind f-strings is to make string interpolation simpler.**

**To create an f-string, prefix the string with the letter “ f ”. The string itself can be formatted in much the same way that you would with str.format(). F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting.**

|  |
| --- |
| # Python program to demonstrate  # string interpolation    >>> n1 = 'Hello'  >>> n2 ='GeeksforGeeks'    # f tells Python to restore the value of two  # string variable name and program inside braces {}  >>> print(f"{n1}! This is {n2}")    # inline arithmetic  >>> print(f"(2 \* 3)-10 = {(2 \* 3)-10}") |

# **Type Conversion**

Python defines type conversion functions to directly convert one data type to another which is useful in day to day and competitive programming. This article is aimed at providing the information about certain conversion functions.

## **int()**

This function converts any data type to integer

## **float()**

This function is used to convert any data type to a floating point number

|  |
| --- |
| >>> s = "26"  >>> c = int(s)  # string converting to int  >>> print(d)  # 26 #type int  >>> d = float(c)  # string converting to int  >>> print(d)  # 26.0 #type float |

## **ord()**

This function is used to convert a character to integer.

## **hex()**

This function is to convert integer to hexadecimal string.

## **oct()**

This function is to convert integer to octal string.

## tuple()

This function is used to convert to a tuple.

## set()

This function returns the type after converting to set.

## list()

This function is used to convert any data type to a list type.

## dict()

This function is used to convert a tuple of order (key,value) into a dictionary.

## str()

Used to convert integer into a string.

## complex(real,imag)

This function converts real numbers to complex(real,imag) number.

|  |
| --- |
| >>> a = 1  >>> b = 2  # initializing tuple  >>> tup = (('a', 1) ,('f', 2), ('g', 3))  # printing integer converting to complex number  >>> c = complex(1,2)  >>> print (c) # (1+2j)  # printing integer converting to string  >>> c = str(a)  >>> print (c) # 1  # printing tuple converting to expression dictionary  >>> c = dict(tup)  >>> print (c) # {'a': 1, 'f': 2, 'g': 3} |

# Control Flow Tools

## if Statements

Perhaps the most well-known statement type is the if statement.

|  |
| --- |
| >>> x = int(input("Please enter an integer: "))  # Please enter an integer: 42  >>> if x < 0:  ... x = 0  ... print('Negative changed to zero')  ... elif x == 0:  ... print('Zero')  ... elif x == 1:  ... print('Single')  ... else:  ... print('More')  ...  # More |

## for Statements

The for statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

|  |
| --- |
| # Measure some strings:  ... words = ['cat', 'window', 'defenestrate']  >>> for w in words:  ... print(w, len(w))  ...  # cat 3  # window 6  # defenestrate 12 |

## The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions:

|  |
| --- |
| >>> for i in range(5):  ... print(i)  # 0  # 1  # 2  # 3  # 4 |

|  |
| --- |
| >>> for i in range(5, 10):  ... print(i)  # 5, 6, 7, 8, 9 |

|  |
| --- |
| >>> a = ['Mary', 'had', 'a', 'little', 'lamb']  >>> for i in range(len(a)):  ... print(i, a[i])  # 0 Mary  # 1 had  # 2 a  # 3 little  # 4 lamb |

## break and continue Statements, and else Clauses on Loops

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the iterable (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:

|  |
| --- |
| >>> for n in range(2, 10):  ... for x in range(2, n):  ... if n % x == 0:  ... print(n, 'equals', x, '\*', n//x)  ... break  ... else:  ... # loop fell through without finding a factor  ... print(n, 'is a prime number')  ...  # 2 is a prime number  # 3 is a prime number  # 4 equals 2 \* 2  # 5 is a prime number  # 6 equals 2 \* 3  # 7 is a prime number  # 8 equals 2 \* 4  # 9 equals 3 \* 3 |

|  |
| --- |
| >>> for n in range(1, 8):  ... if n % 2 == 0 :  ... continue  ... print(n)  # 1  # 3  # 5  # 7 |

## while Loop Statements

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

|  |
| --- |
| >>> count = 0  >>> while (count < 9):  ... print ('The count is:', count)  ... count += 1  >>> else:  ... print ("Good bye!")# The count is: 0  # The count is: 1  # The count is: 2  # The count is: 3  # The count is: 4  # The count is: 5  # The count is: 6  # The count is: 7  # The count is: 8  # Good bye! |

# Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

|  |
| --- |
| >>> def my\_function():  ... print("Hello from a function") |

To call a function, use the function name followed by parenthesis:

|  |
| --- |
| >>> def my\_function():  ... print("Hello from a function")  >>> my\_function() |

## Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

|  |
| --- |
| >>> def my\_function(fname):  ... print(fname + " Refsnes")  >>> my\_function("Emil")  >>> my\_function("Tobias")  >>> my\_function("Linus") |

## Arbitrary Arguments, \*args

If you do not know how many arguments that will be passed into your function, add a \* before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

|  |
| --- |
| >>> def my\_function(\*kids):  ... print("The youngest child is " + kids[2])  >>> my\_function("Emil", "Tobias", "Linus") |

## Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

|  |
| --- |
| >>> def my\_function(child3, child2, child1):  ... print("The youngest child is " + child3)  >>> my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus") |

## Arbitrary Keyword Arguments, \*\*kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: \*\* before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

|  |
| --- |
| >>> def my\_function(\*\*kid):  ... print("His last name is " + kid["lname"])  >>> my\_function(fname = "Tobias", lname = "Refsnes") |

## Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

|  |
| --- |
| >>> def my\_function(country = "Norway"):  ... print("I am from " + country)  >>> my\_function("Sweden")  >>> my\_function("India")  >>> my\_function()  >>> my\_function("Brazil") |

## Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

|  |
| --- |
| >>> def my\_function(food):  ... for x in food:  ... print(x)  >>> fruits = ["apple", "banana", "cherry"]  >>> my\_function(fruits) |

## Return Values

To let a function return a value, use the return statement:

|  |
| --- |
| >>> def my\_function(x):  ... return 5 \* x  >>> print(my\_function(3))  >>> print(my\_function(5))  >>> print(my\_function(9)) |

# One-Liner

Swap Two Variables Python One-Liner

|  |
| --- |
| >>> a = 1  >>> b = 2  >>> a, b = b, a  >>> print(a, b)  # 2 1  >>> c = 1  >>> d = 2  >>> c = d  >>> d = c  >>> print(c, d)  # 2 2 |

# Modules

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

## Create a Module

To create a module just save the code you want in a file with the file extension .py:

Save this code in a file named mymodule.py

|  |
| --- |
| >>> def greeting(name):  ... print("Hello, " + name) |

## Use a Module

Now we can use the module we just created, by using the import statement:

Import the module named mymodule, and call the greeting function:

|  |
| --- |
| >>> import mymodule  >>> mymodule.greeting("Jonathan") |

## Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Save this code in the file mymodule.py

|  |
| --- |
| >>> person1 = { ...  "name": "John", ...  "age": 36, ...  "country": "Norway" ...} |

Import the module named mymodule, and access the person1 dictionary:

|  |
| --- |
| >>> import mymodule  >>> a = mymodule.person1["age"]  >>> print(a) |

## Naming a Module

You can name the module file whatever you like, but it must have the file extension .py

## Re-naming a Module

You can create an alias when you import a module, by using the as keyword:

Create an alias for mymodule called mx:

|  |
| --- |
| >>> import packages.mymodule as mx # packages is directory  >>> a = mx.person1["age"]  >>> print(a) |

## Import From Module

You can choose to import only parts from a module, by using the from keyword.

The module named mymodule has one function and one dictionary:

|  |
| --- |
| >>> def greeting(name):  ... print("Hello, " + name)  >>> person1 = {  ... "name": "John",  ... "age": 36,  ... "country": "Norway"  } |

Import only the person1 dictionary from the module:

|  |
| --- |
| >>> from mymodule import person1  >>> print (person1["age"]) |

# List Comprehensions

With this elegant approach, you could rewrite the for loop from the first example in just a single line of code:

|  |
| --- |
| >>> squares = [i \* i for i in range(10)]  >>> squares  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81] |

|  |
| --- |
| >>> list = [i for i in range(10) if i % 2 == 0]  >>> print(list)  # [0, 2, 4, 6, 8] |