**Variable Names, Types and Basic Operators in Python**

In Python, variables are created when you assign a value to it. Variables are reserved memory locations to store values. That means, when you create a variable, you reserve a space in memory for its storage temporarily.

Based on the datatype of a variable, the Python interpreter allocates memory and decides what can be stored in the reserved memory. That means, by reserving different data types to variables, you can store integers, decimals or characters in these variables.

**Variable types in Python**

In many other programming languages, you need to declare the type of variable depending on what type of data you’ll store in it. For example, int first = 10; float second = 20.5; . These are called statically typed languages, and you can’t store a different type of data with these variables.

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables in Python.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

counting = 10 # An integer assignment

meters = 100.0 # A floating point

name = "Uhuru" # A string

print (counting)

print (meters)

print (name)

Here, 10, 100.0 and "Uhuru" are the values assigned to ***counting*, *meters*,** and ***name*** variables, respectively. This produces the following result −

**10**

**100.0**

**Uhuru**

### Example

**Variables in Python:**

number = 10

greetings = "Hello, World!"

Python on the other hand is a dynamically typed language. So you don’t need to define the type of variables upfront. You can even assign one type of data to a variable and later reassign another type of data to the same variable! Here’s what I mean:

### Example

1 # here number1 is an integer

2 number1 = 4

3 print(number1)

4 # here number1 is a float

5 number1 = 5.5

6 print(number1)

7 # now here I am a string

8 number1 = ‘ ‘ ‘Now I’m a String’ ‘ ‘

9

10 number1 = [1,2,3,4,5] # here number1 is a list

11 print(number1)

**Output**

4

5.5

Now I’m a String

[1,2,3,4,5]

The above illustration states that a variable type can be changed, and notice that you’re not limited to assigning only numbers to a variable. But rather any data type is supported.

**Multiple Assignments**

Python allows us to assign a single value to several variables simultaneously. For example:-

a = b = c = g = f =3

Here, an integer object is created with the value 3, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

a,b,c = 7,8,"George"

Here, two integer objects with values 7 and 8 are assigned to variables **a** and **b** respectively, and one string object with the value of "**George**" is assigned to the variable **c**.

Python is completely object oriented, and not "statically typed". You do not need to declare variables before using them, or declare their type. Every variable in Python is an object.

### **Numbers**

Python supports two types of numbers - **integers** and **floating point numbers**. (It also supports complex numbers).

### Example

**Integer variable declaration in Python:**

1. **myint = 7**
2. **print(myint)**

### **Output:**

7

### **Example**

**Float variable declaration in Python:**

1. myfloat = 7.0
2. print(myfloat)
3. myfloat = float(7)
4. print(myfloat)

**Output:**

**7.0**

**7.0**

A **complex number** is represented by “ x + yi “. Python converts the real **numbers** x and y into **complex** using the function **complex**(x,y). The real part can be accessed using the function real() and the imaginary part can be represented by imag().

Python can handle complex numbers and its associated functions using the file “cmath”. Complex numbers have their uses in many applications related to mathematics, and python provides useful tools to handle and manipulate them.

**Note:** Complex numbers in Python is a topic currently beyond the scope of this introductory course. So we shall not cover it here.

### **Strings in Python**

Strings variables in Python are defined either with a single quote or a double quote.

### **Example**

**String variable declaration in Python:**

1 mystring = 'Hello World!'

2 print(mystring)

3 mystring = "hello there"

4 print(mystring)

**Output:**

**Hello World!**

**hello there**

The difference between the two is that using double quotes makes it easy to include apostrophes (whereas these would terminate the string if using single quotes)

See example below

mystring = "Don't worry about apostrophes"

print(mystring)

OR

mystring = ‘Don\'t worry about apostrophes’

print(mystring)

In the above example, if you were to declare a string variable with a single quote then you will need to escape the \’ in the “**Don’t** with a backslash, otherwise without the escape character we shall receive an error message in the code.

There are additional variations on defining strings that make it easier to include things such as carriage returns, backslashes and Unicode characters. These are beyond the scope of this tutorial.

For further reading and reference, please check the official website here <https://docs.python.org/3/tutorial/introduction.html#strings>

Simple operators in Python can be executed on numbers and strings:

**See example below:**

1. moja = 1
2. mbili = 2
3. tatu = moja + mbili
4. print(tatu) # result ⇒ 3
5. habari = "habari"
6. zenu = "zenu"
7. #the (“ “) in the middle is to help separate the 2 words
8. jambo = habari + " " + zenu
9. print(jambo)

Explanation: On line 1, we assign 1 to a variable called moja, on line 2 we assign 2 to a variable called mbili. On line 3 we add the two previous variables (1 and 2) and assign them to number 3.

On line 4 we print the output to the screen.

On line 5 we assign a swahili greeting to a variable habari . The statement is a string hence it is enclosed in double quotes. Single quotes can also be used here. In line 6 we assign string “zenu” to the variable zenu. In line 7 we combine the two variables (habari and zenu). In line 8 we print the result. Please type this into your computer and run it. See what it brings. First a first time user, it brings a bunch of joy to see a running program at work.

In Python programming, assignments can be done on more than one variable "simultaneously" on the same line like this

**Example code:**

1. a, b, c, d = 3, 4, 5, 6
2. print(a,b, c, d)

**Output:** 3 4 5 6

Mixing operators between numbers and strings is not supported:

**Example:**

# This will not work!

1. one = 1.0
2. two = 2
3. hello = "hello"

print(one + two + hello)

## Exercise

The target of this exercise is to create a string, an integer, and a floating point number. The string should be named mystring and should contain the word "hello". The floating point number should be named myfloat and should contain the number 10.0, and the integer should be named myint and should contain the number 20.

**Solution:**

1. # change this code
2. mystring = None
3. C
4. myint = None
5. # testing code
6. if mystring == "hello":
7. print("String: %s" % mystring)
8. if isinstance(myfloat, float) and myfloat == 10.0:
9. print("Float: %f" % myfloat)
10. if isinstance(myint, int) and myint == 20:
11. print("Integer: %d" % myint)

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## **Standard Data Types**

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types: −

* Numbers
* String
* List
* Tuple
* Dictionary

## Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example: −

**value1 = 2**

**Value2 = 3**

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is −

**del value1[,value2[,value3[....,valueN]]]]**

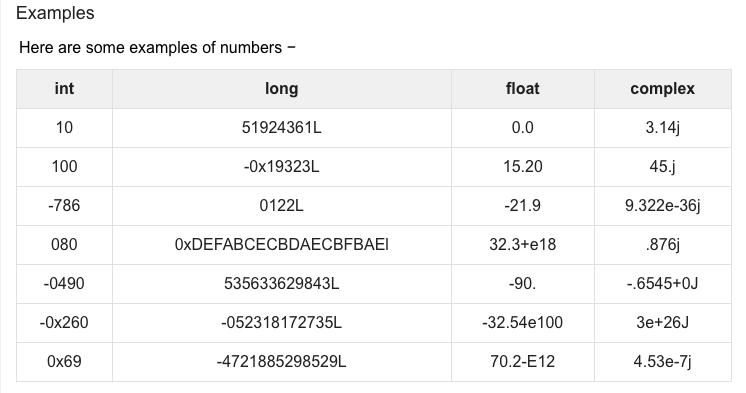
You can delete a single object or multiple objects by using the del statement. For example: −

**del value1**

**del value\_2, value\_3**

**Python supports four different numerical types −**

* int (signed integers)
* long (long integers, they can also be represented in octal and hexadecimal)
* float (floating point real values)
* complex (complex numbers)



* Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
* A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

## **Python Strings**

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator. For example: −

1 str = 'Hello World!'

2 print (str) # Prints complete string

3 print (str[0]) # Prints first character of the string

4 print (str[2:5]) # Prints characters starting from 3rd to 5th

5 print (str[2:]) # Prints string starting from 3rd character

6 print (str \* 2) # Prints string two times

7 print (str + "TEST CASE") # Prints concatenated string

This will produce the following result: −

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST CASE

## Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data types.

Lists can contain any type of variable, and they can contain as many variables as you wish. Lists can also be iterated over in a very simple manner.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (\*) is the repetition operator. An example of list: −

1 list = [ 'abcd3', 886 , 3.25, 'Jonathan', 90.3 ]

2 tinylist = [1123, 'George']

3 print (list) # Prints complete list (correct in Python 2.7)

4 print (list) # Prints complete list (correct as from Python 3.x and above)

5 print (list[0]) # Prints first element of the list

6 print (list[1:3]) # Prints elements starting from 2nd to 3rd

7 print (list[2:]) # Prints elements starting from 3rd element

8 print (tinylist \* 2) # Prints list two times

9 print (list + tinylist) # Prints concatenated lists

This produce the following result: −

['abcd3', 886, 3.25, 'Jonathan', 90.3]

abcd3

[886, 3.25]

[3.25, 'Jonathan', 90.3]

[1123, 'George', 1123, 'George']

['abcd3', 886, 3.25, 'Jonathan', 90.2, 1123, 'George']

**Another example:-**

1 mylist = []

2 mylist.append(1)

3 mylist.append(2)

4 mylist.append(3)

5 print(mylist[0]) # prints 1

6 print(mylist[1]) # prints 2

7 print(mylist[2]) # prints 3

8 # prints out 1,2,3

9 for x in mylist:

10 print(x)

**Output of the above program when run:**

1

2

3

1

2

3

When you try to access an index which does not exist the program generates an exception (an error).

1 mylist = [1,2,3]

2 print(mylist[10])

**Output of the above program’s execution**

**Traceback (most recent call last):**

**File "<stdin>", line 2, in <module>**

**print(mylist[10])**

**IndexError: list index out of range**

**Exercise**

In this exercise, you will need to add numbers and strings to the correct lists using the "append" list method. You must add the numbers 1,2, and 3 to the "numbers" list, and the words 'hello' and 'world' to the strings variable.

You will also have to fill in the variable second\_name with the second name in the names list, using the brackets operator []. Note that the index is zero-based, so if you want to access the second item in the list, its index will be 1.

1 numbers = []

2 strings = []

3 names = ["John", "Eric", "Jessica"]

4 # write your code here

5 second\_name = None

6 # this code should print out the filled arrays and the second name in the names list (Eric).

7 print(numbers)

8 print(strings)

9 print("The second name on the names list is %s" % second\_name)

**Output of the above program when run**

[1, 2, 3]

['hello', 'world']

The second name on the names list is Eric

## **Python Tuples**

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses. "*A tuple is a data structure that has a specific number and sequence of elements*". The only key difference is tuples are immutable. Also tuples use less space than lists. You can use tuples when you want a list of constants.

You can create a new tuple with the same name and modify it in some way, but you can’t modify an existing tuple. Lists are mutable, which means that you can change them.

A tuple is a collection used to create complex lists in Python, in which you can embed one tuple within another. This embedding lets you create hierarchies with tuples. A hierarchy could be something as simple as the directory listing of your hard drive or an organizational chart for your company. The idea is that you can create complex data structures using a tuple.

The main differences between lists and tuples are: Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as read-only lists.

**For example: −**

1 tuple = ( 'abcd',586 ,3.23, 'Rebecca',70.3 )

2 tinytuple = (123, 'Kamau')

3 print (tuple) # Prints the complete tuple

4 print (tuple[0] ) # Prints first element of the tuple

5 print (tuple[1:3]) # Prints elements of the tuple starting from 2nd till 3rd

6 print (tuple[2:]) # Prints elements of the tuple starting from 3rd element

7 print (tinytuple \* 2) # Prints the contents of the tuple twice

# Prints concatenated tuples

8 print (tuple + tinytuple)

**The above program when run will produce the following results**

('abcd', 586, 3.23, 'Rebecca', 70.3)

abcd

(586, 3.23)

(3.23, 'Rebecca', 70.3)

(123, 'Kamau', 123, 'Kamau')

('abcd', 586, 3.23, 'Rebeccas', 70.3, 123, 'Kamau')

Tuple is not updateable. So the following block of code is invalid at line #3 while line #4 is valid.

1 tuple = ( 'abcd', 783 , 3.13, 'Ali', 71.2 )

2 list = [ 'abcd', 783 , 3.13, 'Ali', 71.2 ]

3 tuple[2] = 1000 # Invalid syntax with tuple

4 list[2] = 1000 # Valid syntax with list

## Python Dictionary

Python's dictionaries are a kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

1 dict = {}

2 dict['one'] = "This is number one"

3 dict[2] = "This is number two"

4 tinydict = {'name': 'Henry','code':6733, 'dept': 'IT'}

5 print (dict['one']) # Prints value for 'one' key

6 print (dict[2]) # Prints value for 2 key

7 print (tinydict) # Prints complete dictionary

8 print (tinydict.keys()) # Prints all the keys

9 print (tinydict.values()) # Prints all the values

The above code when run will produce the following results

**This is number one**

**This is number two**

**{'dept': 'IT', 'code': 6733, 'name': 'Henry'}**

**['dept', 'code', 'name']**

**['IT', 6733, 'Henry']**

**Note:** Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

With Python, creating and using a dictionary is much like working with a list, except that you must now define a **key** and **value** pair. Here are the special rules for creating a key:

**The key must be unique.** When you enter a duplicate key, the information found in the second entry wins — the first entry is simply replaced with the second.

**The key must be immutable.** This rule means that you can use strings, numbers, or tuples for the key. You can’t, however, use a list for a key.

You have no restrictions on the values you provide. A value can be any Python object, so you can use a dictionary to access an employee record or other complex data. The following steps help you understand how to use dictionaries better.

At the Python prompt or in your IDE, type the following.

## **Type the following code and run it**

## **Colors = {“Tamara”: “Pink”,**

## **“Sarah”: “Red”,**

## **“Odero”: “Blue” }**

## **then press Enter.**

Python creates a dictionary containing three entries with people’s favorite colors. Notice how you create the **key** and **value** pair. The key comes first, followed by a colon and then the value. Each entry is separated by a comma.

Type **Colors** and then **press Enter**

You will see the key and value pairs. However, notice that the entries are sorted in key order. A dictionary automatically keeps the keys sorted to make access faster, which means that you get fast search times even when working with a large data set. The downside is that creating the dictionary takes longer than using something like a list because the computer is busy sorting the entries.

## **Type Colors[“Tamara”] and press Enter. The result here should return color Pink**

## **Type Colors[“Odero”] and press Enter. The result here should return color Blue**

## **Type Colors[“Odero”] and press Enter. The result here should return color Red**

You will see the color associated with Sarah, **Red**. Using a string as a key, rather than using a numeric index, makes the code easier to read and makes it self-documenting to an extent.

**Basic Python Operators**

Operators are the constructs which can manipulate the value of operands.

Consider the expression **1 + 4 = 5**. Here, **1** and **4** are called **operands** and **+** is called **operator**.

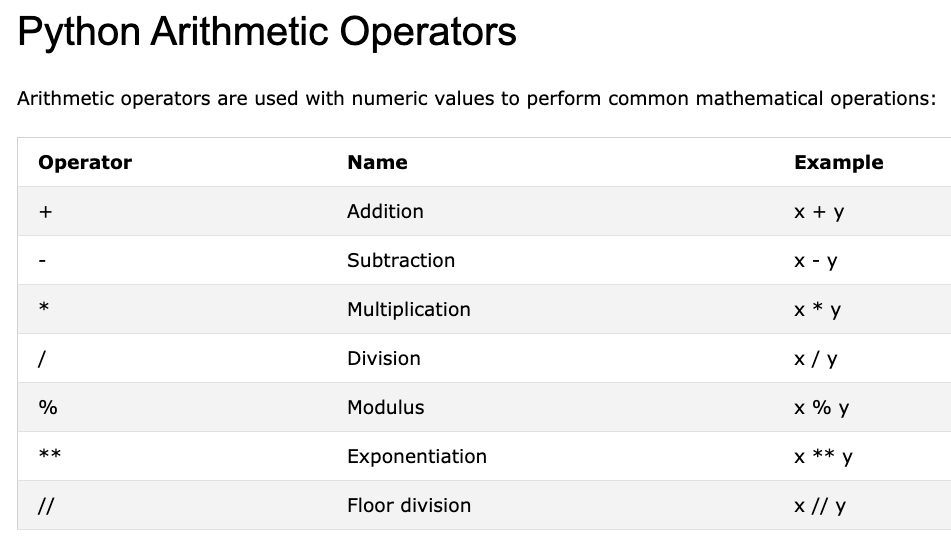
Just like any other programming language, the addition, subtraction, multiplication, and division operators can be used with numbers.

## Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

* Arithmetic operators +, /, -, \*
* Assignment operators =
* Comparison operators ==
* Logical operators > < =
* Identity operators
* Membership operators
* Bitwise operators



**Examples:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Addition** | x = 5  y = 3  print(x + y) |  | **Modulus** | x = 5  y = 2  print(x % y) |
|  | **Subtraction** | x = 5  y = 3  print(x - y) |  | **Exponential** | x = 2  y = 5  print(x \*\* y) #same as 2\*2\*2\*2\*2 |
|  | **Multiplication** | x = 5  y = 3  print(x \* y) |  | **Floor division** | x = 15  y = 2  print(x // y)  #the floor division // rounds the result down to the nearest whole number |
|  | **Division** | x = 12  y = 3  print(x / y) |  |  |  |

## Python Assignment Operators

# Assignment operators are used to assign values to variables:

# 

## 

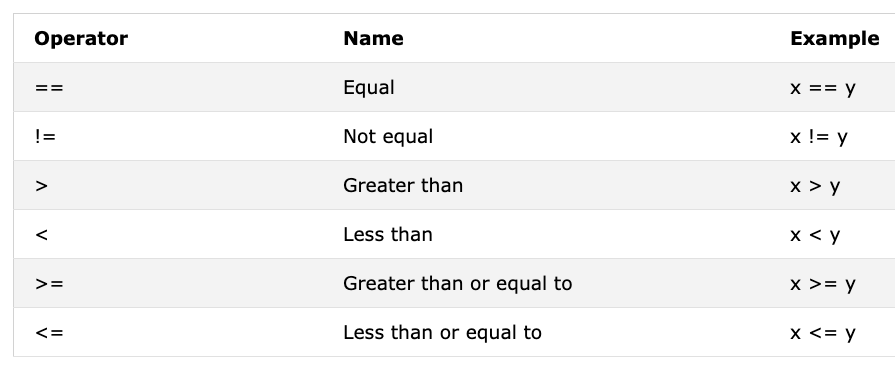
## 

## 

## 

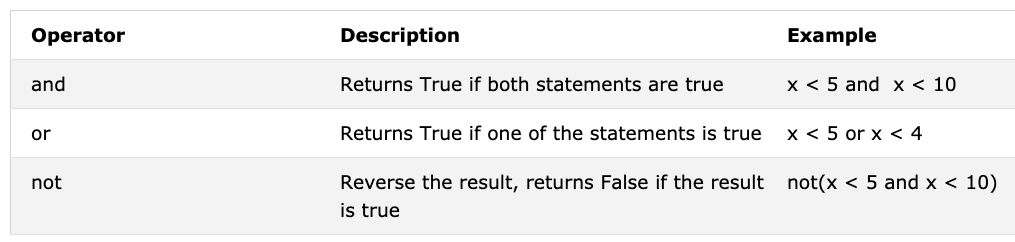
## Python Comparison Operators

Comparison operators are used to compare two values:



## Python Logical Operators

Logical operators are used to combine conditional statements:



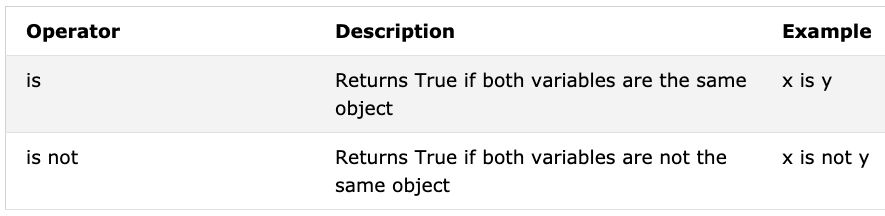
## 

## 

## 

## Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:



**Code Example:**

1 x = ["apple", "banana"]

2 y = ["apple", "banana"]

3 z = x

4 print(x is z)

5 # returns True because z is the same object as x

6 print(x is y)

7 # returns False because x is not the same object as y, even if they have the same content

8 print(x == y)

9 # to demonstrate the difference between "is" and "==": this comparison returns True because x is equal to y

**Output:**

True

False

True

**Another Code Example:**

1 x = ["apple", "banana"]

2 y = ["apple", "banana"]

3 z = x

4 print(x is not z)

5 # returns False because z is the same object as x

6 print(x is not y)

7 # returns True because x is not the same object as y, even if they have the same content

8 print(x != y)

9 # to demonstrate the difference between "is not" and "!=": this comparison returns False

10 because x is equal to y

**Output:**

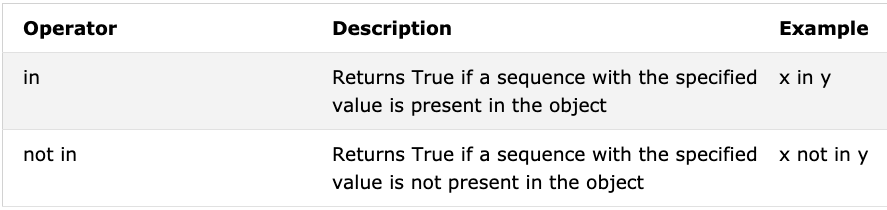
False

True

False

## Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:



**Code example**

1 x = ["apple", "banana"]

2 print("banana" in x)

3 # returns True because a sequence with the value "banana" is in the list

## 

## 

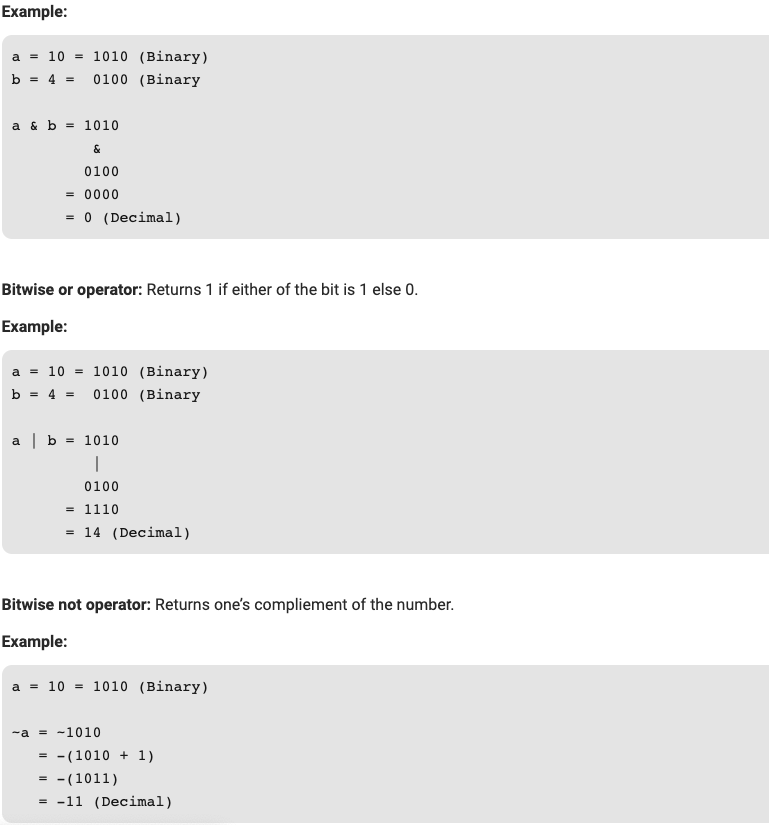
## 

## Python Bitwise Operators

# Bitwise operators are used to compare (binary) numbers:

# 

# **Examples codes**



# 

**Code Example:**

|  |
| --- |
| 1 # Python program to show  2 # bitwise operators  3 a = 10  4 b = 4  5 # Print bitwise AND operation  6 print("a & b =", a & b)  7  8 # Print bitwise OR operation  9 print("a | b =", a | b)  10  11 # Print bitwise NOT operation  12 print("~a =", ~a)  13  14 # print bitwise XOR operation  15 print("a ^ b =", a ^ b) |

The above program when run will create the following output

a & b = 0

a | b = 14

~a = -11

a ^ b = 14

# 

## **Shift Operators:**

# These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two.

# **Bitwise right shift:** Shifts the bits of the number to the right and fills 0 on voids left as a result. Similar effect as of dividing the number with some power of two.

**Example:**

Example 1:

a = 10

a >> 1 = 5

Example 2:

a = -10

a >> 1 = -5

**Bitwise left shift:** Shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.

**Example:**

a = 5 = 0000 0101

b = -10 = 1111 0110

a << 1 = 0000 1010 = 10

a << 2 = 0001 0100 = 20

b << 1 = 0000 1010 = -20

b << 2 = 0001 0100 = -40

# 

# **Python Sample Program**

|  |
| --- |
| 1 # Python program to show  2 # shift operators    3 a = 10  4 b = -10    5 # print bitwise right shift operator  6 print("a >> 1 =", a >> 1)  7 print("b >> 1 =", b >> 1)    8 a = 5  9 b = -10    10 # print bitwise left shift operator  11print("a << 1 =", a << 1)  12 print("b << 1 =", b << 1)  **Output:**  a >> 1 = 5  b >> 1 = -5  a << 1 = 10  b << 1 = -20 |

# 

# 

# 

# **Python’s Core Data Types**

Data Types may be defined as a classification that specifies which type of value a variable has, and what type of mathematical, relational, or logical operations can be applied to it without causing an error. Python Data Types are mainly of three types: **Boolean**, **integer**, and **string**. These may also be called the **core data types** in Python.

The core data types are known as core data types because they are effectively built into the Python programming language, this means that there is a specific syntax for generating most of them.

**Python’s Core Data Types include:**

* Numbers
* Strings
* Lists
* Dictionaries
* Tuples
* Files
* Other types include: Sets, types, None, Booleans

Since we have looked at the numbers, strings, lists, dictionaries and tuples earlier in this chapter, we shall just brush over them real quick, but concentrate on the Files, Sets, types, None and Booleans for now.

**Files:**

File objects may be described as Python code’s main interface to external files on your computer. They are one of the popular core data types in Python, but without any specific literal syntax for creating them. To create a file object, one needs to call the built-in open function, passing in an external filename as a string, and a processing mode string.

For example, to create an output file, you would pass in its name and the ‘w’ processing mode string to write data:

From the command prompt type

>>> f = open(‘data.txt’, ‘w’) # Makes a new file in output mode

>>> f.write(‘Habari \n’) # Write this string of byte to the file

# >>> f.write(‘Yako \n’)

>>> f.close() # Close to flush memory buffers to disk

# The other file object methods support additional features, offering more ways of reading and writing (read accepts an optional byte size, read line reads one line at a time, and so on), as well as other tools (seek moves to a new file position).

**Sets:**

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

**Note:** Sets are unordered, so you cannot be sure in which order the items will appear.

### **Example**

Create a Set:

1 thisset = {"apple", "banana", "cherry"}

2 print(thisset)

3 thisset = {"apple", "banana", "cherry"}

4 print(thisset)

5 # Note: the set list is unordered, meaning: the items will appear in a random order.

# Refresh this page to see the change in the result.

Further examples of sets in Python.

>>> basket = [‘apple’, ‘orange’, ‘apple’, ‘pear’, ‘orange’, ‘banana’]

>>> fruit = set(basket) # create a set

>>> fruit

set([’orange’, ’pear’, ’apple’, ’banana’])

>>> ’orange’ in fruit # fast membership testing

True

>>> ’crabgrass’ in fruit

False

>>> a = {x for x in ’abracadabra’ if x not in ’abc’} >>> a set([’r’, ’d’])

## **Python Numbers**

Integers, floating point numbers and complex numbers fall under [Python numbers](https://www.programiz.com/python-programming/numbers) category. They are defined as int, float and complex classes in Python.

We can use the type() function to know which class a variable or a value belongs to. Similarly, the isinstance() function is used to check if an object belongs to a particular class.

a = 5

print(a, "is of type", type(a))

a = 2.0

print(a, "is of type", type(a))

a = 1+2j

print(a, "is complex number?", isinstance(1+2j,complex))

**Output**

**5 is of type <class 'int'>**

**2.0 is of type <class 'float'>**

**(1+2j) is complex number? True**

Integers can be of any length, it is only limited by the memory available.

A floating-point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is an integer, 1.0 is a floating-point number.

Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part. Here are some examples.

>>> a = 1234567890123456789

>>> a

1234567890123456789

>>> b = 0.1234567890123456789

>>> b

0.12345678901234568

>>> c = 1+2j

>>> c

(1+2j)

Notice that the float variable b got truncated.

# 

# 

# 

# 

# 

# Python Syntax

## Execute Python Syntax

As we learned in the previous page, Python syntax can be executed by writing directly in the Command Line as seen below:

>>> print("Hello, World!")

Hello, World!

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line:

C:\Users\*gndede*>python myfile.py

## Python Indentation

Indentation in Python Programming refers to the spaces at the beginning of a line of code.

Whereas in other programming languages the indentation in code is for readability only, the indentation in Python is very important and is part of the language’s syntax.

Python uses indentation to indicate a block of code.

### Example

if 10 > 7:

print("Ten is greater than seven!")

Python will alert you with an error if you skip the indentation: The number of spaces is up to you as a programmer, but it has to be at least one.

**if 15 > 12:**

**print("Fifteen is greater than twelve!")**

**if 15 > 12:**

**print("Fifteen is greater than twelve!")**

You have to use the same number of spaces (be consistent) in the same block of code, otherwise Python will give you an error:

Good examples would be to use something like this than the example above;

**if 15 > 12:**

**print("Fifteen is greater than twelve!")**

**if 15 > 12:**

**print("Fifteen is greater than twelve!")**

**Further Reading**

Further Variable Names and Basic Operators in Python - II

## **Conversion between data types also known as Type Casting**

Python defines type conversion functions to directly convert one data type to another which is useful in day to day and competitive programming. The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1. Implicit Type Conversion
2. Explicit Type Conversion

## **Implicit Type Conversion**

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

### **Example 1: Converting integer to float**

1. num\_int = 123 #We add two variables **num\_int** and **num\_flo**, storing the value in num\_new.
2. num\_flo = 1.23 #We will look at the data type of all three objects respectively.
3. num\_new = num\_int + num\_flo #In the output, we can see the data type of num\_int is an integer while the data type of num\_flo is a float.
4. print("datatype of num\_int:",type(num\_int)) # Also, we can see the num\_new has a float data type because Python always converts smaller data types to larger data types to avoid the loss of data.
5. print("datatype of num\_flo:",type(num\_flo))
6. print("Value of num\_new:",num\_new)
7. print("datatype of num\_new:",type(num\_new))
8. **If we run this above code, below is what we get**
9. datatype of num\_int: <class 'int'>
10. datatype of num\_flo: <class 'float'>
11. Value of num\_new: 124.23
12. datatype of num\_new: <class 'float'>

**1. int(a,base)** : This function converts **any data type to integer**. ‘Base’ specifies the **base in which string is** if the data type is string.

**2. float()** : This function is used to convert **any data type to a floating point number**

**3. str() :** constructs a string from a wide variety of data types, including strings, integer literals and float literalsA

We can convert between different data types by using different type conversion functions like int(), float(), str(), etc.

>>> float(5)

5.0

Conversion from float to int will truncate the value (make it closer to zero).

>>> int(10.6)

10

>>> int(-10.6)

-10

Conversion to and from string must contain compatible values.

>>> float('2.5')

2.5

>>> str(25)

'25'

>>> int('1p')

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

ValueError: invalid literal for int() with base 10: '1p'

We can even convert one sequence to another.

>>> set([1,2,3])

{1, 2, 3}

>>> tuple({5,6,7})

(5, 6, 7)

>>> list('hello')

['h', 'e', 'l', 'l', 'o']

To convert to dictionary, each element must be a pair:

>>> dict([[1,2],[3,4]])

{1: 2, 3: 4}

>>> dict([(3,26),(4,44)])

{3: 26, 4: 44}

**Code Example of Type conversion**

1 # Python code to demonstrate Type conversion

2 # using int(), float()

3 # initializing string

4 string = "10010"

5 # printing string converting to int base 2

6 converter = int(string,2)

7 print ("After converting to integer base 2 : ", end="")

8 print (converter)

9 # printing string converting to float

10 outcome = float(string)

11 print ("After converting to float : ", end="")

12 print (outcome)

If we run the above program, we shall see the following outcome

**Output:**

After converting to integer base 2 : 18

After converting to float : 10010.0

# Now, let's try adding a string and an integer, and see how Python deals with it.

### 

### 

### 

### **Example 2: Addition of string(higher) data type and integer(lower) datatype**

1 num\_int = 123

2 num\_str = "456"

3 print("Data type of num\_int:",type(num\_int))

4 print("Data type of num\_str:",type(num\_str))

5 print(num\_int+num\_str)

**Below is the output of the above program when run**

Data type of num\_int: <class 'int'>

Data type of num\_str: <class 'str'>

Traceback (most recent call last):

File "<string>", line 7, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

**In the above program,**

* We add two variables num\_int and num\_str.
* As we can see from the output, we got TypeError. Python is not able to use Implicit Conversion in such conditions.
* However, Python has a solution for these types of situations which is known as Explicit Conversion.

**Explicit Type Conversion**

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

**Syntax :**

<required\_datatype>(expression)

Typecasting can be done by assigning the required data type function to the expression.

### **Example 3: Addition of string and integer using explicit conversion**

1 num\_int = 123

2 num\_str = "456"

3 print("Data type of num\_int:",type(num\_int))

4 print("Data type of num\_str before Type Casting:", type(num\_str))

5 num\_str = int(num\_str)

6 print("Data type of num\_str after Type Casting:", type(num\_str))

7 num\_sum = num\_int + num\_str

8 print("Sum of num\_int and num\_str:", num\_sum)

9 print("Data type of the sum:", type(num\_sum))

**When we run the above program, the output will be the following:**

Data type of num\_int: <class 'int'>

Data type of num\_str before Type Casting: <class 'str'>

Data type of num\_str after Type Casting: <class 'int'>

Sum of num\_int and num\_str: 579

Data type of the sum: <class 'int'>

**In the above program,**

We add num\_str and num\_int variables.

We converted num\_str from string(higher) to integer(lower) type using int() function to perform the addition.

After converting num\_str to an integer value, Python is able to add these two variables.

We got the num\_sum value and data type to be an integer.

Summary and key points to take away

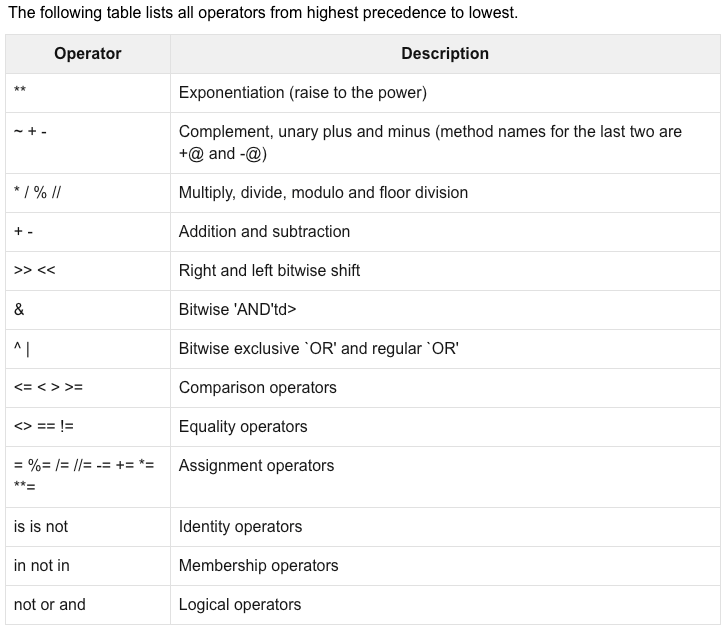
# 

# 

# **Python Type Conversion and Type Casting**

* Type Conversion is the conversion of an object from one data type to another data type.
* Implicit Type Conversion is automatically performed by the Python interpreter.
* Python avoids the loss of data in Implicit Type Conversion.
* Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.
* In Type Casting, loss of data may occur as we enforce the object to a specific data type.

Normally before diving into type conversion in Python, we should have some knowledge of Python data types which we had already covered earlier.

**Precedence and Associativity**

In Python programming language, an operator is a special symbol that is used to carry out specific operations on its operand. In Python, we have a rich set of built in operators to help carry out different types of operations. There are operators for assignment, arithmetic operations, logical operations and comparison operations among others. Python operators can be used with many types of variab;es or constants, but some of the operators are binary, meaning they take two operands, but a few are unary and only take one operand.

**Python divides the operators in the following groups:**

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Identity operators
* Membership operators
* Bitwise operators

**Operator precedence affects how an expression is evaluated.**

For example, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has higher precedence than +, so it first multiplies 3\*2 and then adds into 7.

The operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python follows the same precedence rules for its mathematical operators that mathematics does.

Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60: in this case, the parentheses don’t change the result, but they reinforce that the expression in parentheses will be evaluated first.

Exponentiation has the next highest precedence, so 2\*\*1+1 is 3 and not 4, and 3\*1\*\*3 is 3 and not 27. Can you explain why?

Multiplication and both division operators have the same precedence, which is higher than addition and subtraction, which also have the same precedence. So 2\*3-1 yields 5 rather than 4, and 5-2\*2 is 1, not 6.

Operators with the same precedence are evaluated from left-to-right. In algebra we say they are left-associative. So in the expression 6-3+2, the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been 6-(3+2), which is 1.

**Note:**

Due to some historical quirk, an exception to the left-to-right left-associative rule is the exponentiation operator \*\*. A useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:

1 print(2 \*\* 3 \*\* 2) # the right-most \*\* operator gets done first!

2 print((2 \*\* 3) \*\* 2) # use parentheses to force the order you want!

**OUTPUT:**

512

64

**Exception Handling**

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

The try block lets you test a block of code for errors.

The except block lets you handle the error.

The finally block lets you execute code, regardless of the result of the try- and except blocks.

### Example

The try block will generate an exception, because x is not defined:

try:

print(x)

except:

print("An exception occurred")

**Code samples:**

1 #The try block will generate an error, because x is not defined:

2 try:

3 print(x)

4 except:

5 print("An exception occurred")

The above program when run will produce the following results

An exception occurred

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

If we run the following program, it will produce an error since x is not yet defined.

1 #This will raise an exception, because x is not defined:

2 print(x)

**Following is the output of the above program**

Traceback (most recent call last):

File "demo\_try\_except\_error.py", line 3, in <module>

print(x)

NameError: name 'x' is not defined

**How to make exceptions**

We can define as many exception blocks as we want in a code, e.g. if we want to execute a special block of code for a special kind of error, for example:

### Example

Print one message if the try block raises a NameError and another for other errors:

try:

print(x)

except NameError:

print("Variable x is not defined")

except:

print("Something else went wrong")

**Code example**

#The try block will generate a NameError, because x is not defined:

1 try:

2 print(x)

3 except NameError:

4 print("Variable x is not defined")

5 except:

6 print("Something else went wrong")

**Program out when ran is displayed below**

Variable x is not defined

## Else

we can use the else keyword to define a block of code to be executed if no errors were raised:

**See example below**

### Example

In this example, the try block does not generate any error:

1 try:

2 print("Hello")

3 except:

4 print("Something went wrong")

5 else:

5 print("Nothing went wrong")

**Program output is displayed below**

Hello

Nothing went wrong

## Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

### Example

1 try:

2 print(x)

3 except:

4 print("Something went wrong")

5 finally:

6 print("The 'try except' is finished")

**Program output**

**Something went wrong**

**The 'try except' is finished**

Finally can be useful to close objects and clean up resources:

### Example

Try to open and write to a file that is not writable:

1 try:

2 f = open("demofile.txt")

3 f.write("Lorum Ipsum")

4 except:

5 print("Something went wrong when writing to the file")

6 finally:

7 f.close()

**The program output**

**Something went wrong when writing to the file**

Here the program can continue, without leaving the file object open.

## Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, you will use the raise keyword.

### Example

Raise an error and stop the program if x is lower than 0:

1 x = -1

2 if x < 0:

3 raise Exception("Sorry, no numbers below zero")

**Program output**

Traceback (most recent call last):

File "demo\_ref\_keyword\_raise.py", line 4, in <module>

raise Exception("Sorry, no numbers below zero")

Exception: Sorry, no numbers below zero

The raise keyword is used to raise an exception. We can define what kind of error to raise, and the text to print to the user as an alert.

### 

### Example

Raise a TypeError if x is not an integer:

1 x = "hello"

2 if not type(x) is int:

3 raise TypeError("Only integers are allowed")

Below is the output of what we get back

Traceback (most recent call last):

File "demo\_ref\_keyword\_raise2.py", line 4, in <module>

raise TypeError("Only integers are allowed")

TypeError: Only integers are allowed