Chapter 2 - Python Fundamentals Part 2

2 Python Fundamentals Part 2

2.1 Python Operators and Operands

In mathematics, operations are functions where by values (called operands) are worked on by operators and a result is generally returned. Take the example shown below in figure 1:

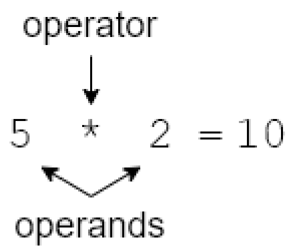


Figure 1: What are operands and operators

The number `5`and `2` are the operands to the `*` (multiply) operator and the result `10` is returned using the `=` equal operator. In Python, operators and operands uses the same concepts as in math but the difference is that there are a lot more types of operators compared to math and its operands are the variables that specify what the value of this data that is to be operated upon.

Python operators can be classified into the following types:

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Membership Operators
- Identity Operators
- Bitwise Operators

2.1.1 Arithmetic Operators

Arithmetic operators are operators that we have all learnt in math in school. Assume that the variables **a = 10** and **b = 21**.

Operator	Description	Example
- (unary)	Unary operator that is used mainly to show unary negation.	-11
+ (Addition)	**Adds** values on either side of the operator.	a + b = 31
- (Subtraction)	**Subtracts** right hand operand from left hand operand.	a - b = -11
* (Multiplication)	**Multiplies** values on either side of the operator.	a * b = 210
/ (Division)	**Divides** left hand operand by right hand operand.	b / a = 2.1
% (Modulus)	Divides left hand operand by right hand operand and returns **remainder**.	b % a = 1
** (Exponent)	Performs **exponential** (power) calculation on operators.	a**20 = 10^{20}
// (Floor Division)	The division of operands where the result is the **quotient** in which the digits after the decimal point are removed. But if one of the operands is a negative number, the result is floored, i.e., rounded away from zero (towards negative infinity).	9 // 2 = 4 -11 // 3 = -4

2.1.2 Comparison Operators

Comparison operators are used to compare the values of the operands and return the relationship between the operands. Assume that the variables **a = 10** and **b = 21**.

Operator	Description	Example
==	Compares the values of two operands for **equality**, true if they are equal.	(a == b) is not true.
!=	Compares the values of two operands for **inequality**, true if they are equal.	(a != b) is true.
>	If the value of left operand is **greater than** the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is **less than** the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is **greater than or equal to** the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is **less than or equal to** the value of right operand, then condition becomes true.	(a <= b) is true.

2.1.3 Logical Operators

Logical operators are used mainly to control the flow of a program. Assume that the variables **a = True** and **b = False**.

Operator	Description	Example
and (logical and)	If **both the operands** are true then condition becomes true.	(a and b) result is False.
or (logical or)	If **any** of the two operands are **non-zero** then condition becomes true.	(a or b) result is True.
not (logical not)	Used to **reverse** the logical state of its operand.	not(a and b) result is True. not(a) results is False

2.1.4 Assignment Operators

Assignment operators are operators that assigns the result of the computation to the **leftmost** operand. These operators can be combined with the Arithmetic operators (from the previous section) to result in a shortened computation line of code. Assume that the variables **a = 10** and **b = 21**.

Operator	Description	Example
=	Assigns the value from right side operands computation to left side operand.	c = a + b assigns value of a + b into c
+= (add then equate)	Adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= (subtract then equate)	Subtracts right operand from the left operand and assign the result to left operand.	c -= a is equivalent to c = c - a
*= (multiply then equate)	Multiplies right operand with the left operand and assign the result to left operand.	c *= a is equivalent to c = c * a
/= (divide then equate)	Divides left operand with the right operand and assign the result to left operand.	c /= a is equivalent to c = c / a
%= (modulus then equate)	Performs modulus on the two operands and assign the result to left operand.	c %= a is equivalent to c = c % a
**= (exponent then equate)	Performs exponential (power) calculation on operators and assign value to the left operand.	c **= a is equivalent to c = c ** a
//= (floor division then equate)	Performs floor division on operators and assign value to the left operand.	c //= a is equivalent to c = c // a

2.1.5 Membership Operators

Membership operators tests whether the object is in a sequence. Example of sequences are strings, lists or tuples. Assume that the variables **x = [1,5,8,3,9,2]**, **y = 5** and **z = 10**

Operator	Description	Example
in	Evaluates to true if it **finds** the object in the specified sequence and false otherwise.	y in x, here ***in*** results in a true if y is a member of sequence x.
not in	Evaluates to true if it **does not finds** a variable in the specified sequence and false otherwise.	z not in x, here ***not in*** results in a true if z is not a member of sequence x.

2.1.6 Identity Operators

Identity operators compare the memory locations of two objects. Note that this is **not equivalent** to the comparison operator `==` as that compares the values of the variables and identity operators compare the physical memory address of the objects. Memory addresses can shown using the built-in function `id()`. Assume that the variables **a = 1001**, **b = 1000 + 1** and **c = a**.

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	c is a, here ***is*** results in a true if id(c) equals id(a).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	a is not b, here ***is not*** results in a true if id(a) is not equal to id(b).

2.1.7 Bitwise Operators

Bitwise operators only works on bits and performs bit-by-bit operations. Assume that the variables **a = 0011 1100** binary for 60 and **b = 0000 1101** binary for 13. Note binary representation of integer number can obtained from the built-in function `bin()`.

Operator	Description	Example
& (binary AND)	Copies a bit, to the result, if there exista a bit at the same position of the operand.	(a & b) = 12 (0000 1100 in binary)
| (binary OR)	Copies a bit, to the result, if there exists a bit in either position of the operand.	(a | b) = 61 (0011 1101 in binary)
^ (binary XOR)	Copies the bit, to the result, set 1 if the bits in the operands are different and 0 if they are the same.	(a ^ b) = 49 (0011 0001 in binary)
~ (binary Ones Complement)	Each bit position in the result is the logical negation of the bit in the operand. It has the effect of 'flipping' bits	(~a) = -61 (1100 0011 in binary but in 2's complement form due to a signed binary number)
<< (binary Left Shift)	Left operand's value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (1111 0000 in binary)
>> (binary Right Shift)	Left operand's value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (0000 1111 in binary)

2.1.8 Operators Precedence

The brackets that is used in conjunction with operators are the rounded brackets `()`. These brackets are used to group equations that are to be evaluated together. But if the rounded brackets are omitted, operators following the operators precedence order according to the table below is used to evaluate the equation. Note that if the operators have the same precedence, the operators are evaluated from left to right. Refer to the examples after the table.

	Operator	**Descriptor**
Highest Precedence	**	Exponentiation (raise to the power of)
	~ + -	Complement, Unary Plus and Minus
	* / % //	Multiply, Divide, Modulus and Floor Division
	+ -	Addition and Subtraction
	>> <<	Right and Left Bitwise Shift
	&	Bitwise 'AND'
	^ |	Bitwise 'XOR' and regular 'OR'
	<= < > >=	Comparison operators
	< > == !=	Equality operators
	= %= /= //= -= += *= **=	Assignment operators
	is is not	Identity operators
	in not in	Membership operators
Lowest Precedence	not or and	Logical operators

An example is the equation $5 + 9/2 * 4$, if there were no rounded brackets, it would be evaluated as in the figure 2 below.

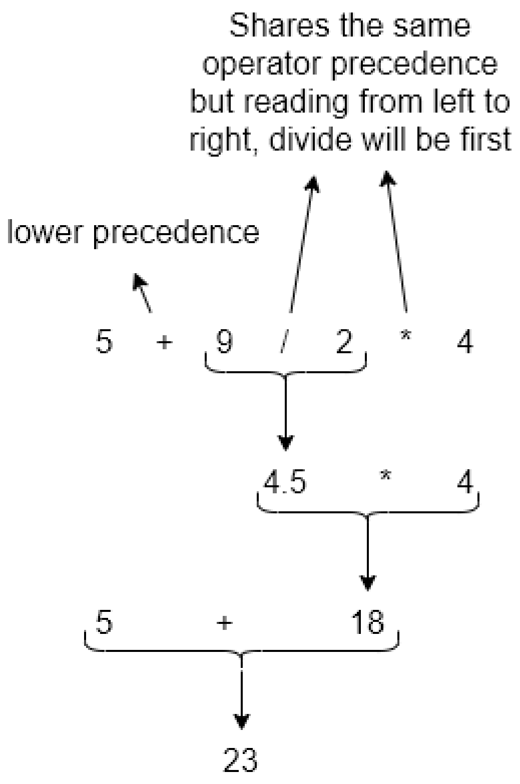

Shares the same operator precedence but reading from left to right, divide will be first
lower precedence
5 + 9 / 2 * 4
4.5 * 4
5 + 18
23

Figure 2: An operation tree showing the operator precedence steps of `5+9/2*4` when brackets are omitted.

However, if there were rounded brackets added to the equation such that the equation now becomes $5 + (9/(2 * 4))$, if would be evaluated as in the figure 3 below.

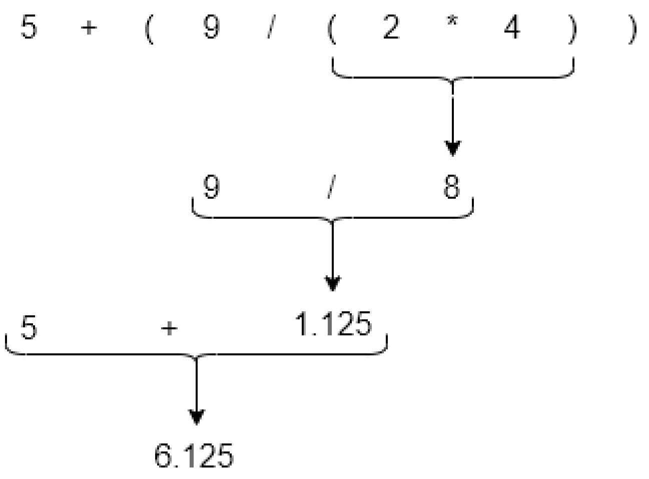


Figure 3: An operation tree showing the operator precedence steps of `5+(9/(2*4))` when brackets are used.

2.2 Standard Data Types

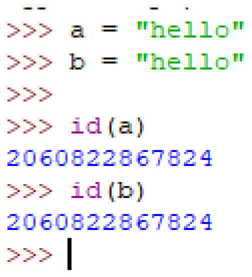
From the start, we have learnt that Python is an object-oriented programming language and therefore **everything** in Python is an object. Think of an *object* is a custom data structure that contains attributes (also called *variables*) and behaviours (also called *functions* or *methods*). An example would be a cardboard box. Its variables are the measurements for width, height and depth and its behaviour is to store one or more items in it (ie the purpose of a box is for storage).

Datatypes in Python are no different. They are also objects with attributes and behaviours. A general definition for the word "datatype" from Oxford Dictionary is as follows

> a particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

The data item in the definition is normally referred to as an object and these objects store values that are what programmers use in programs for data processing. In Python, there are 2 main ways to classify these values, they are *Literals* and *Variables*. A *literal* can be thought of as a constant and a *variable* is a named location that is used to store data in memory while the program is executing. The main difference between *literals* and *variables* is the concept of **mutability**. A literal is immutable (thus it is a constant) and a variable is mutable.

So what is this concept of mutability? Think of mutability in terms of an object that has an identity (this is their memory address), a type and a value. When an object is created, a place in memory (note that memory address never changes) and a value is allocated to it. The type is dependent on the value therefore it also determines the type of operations that the object supports. For instance, below in figure 4, we have 2 objects `a` and `b`. Both are assigned with the word "hello".



```
>>> a = "hello"
>>> b = "hello"
>>>
>>> id(a)
2060822867824
>>> id(b)
2060822867824
>>> |
```

Figure 4: Memory addresses of objects `a` and `b`.

When the `id()` function (this function shows the memory address of the objects) is used, we can see that the memory addresses are the same. From this, we can infer that the objects `a` and `b` are "pointing" to the same "hello" value. Refer to figure 5 below.

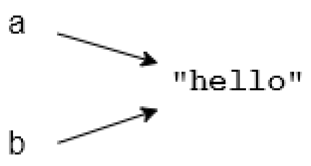
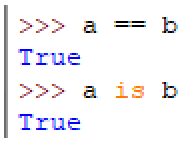


Figure 5: Objects `a` and `b` are pointing to the same value.

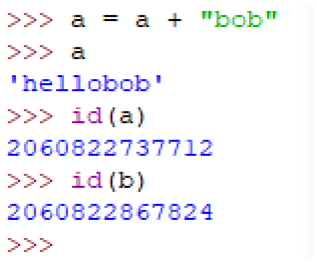
We can further test this hypothesis by using the `==` comparison operator which tests if the values are the same and the `is` identity operator which tests if the objects `a` and `b` are pointing to the same `"hello"` object. Refer to figure 6 below.



```
>>> a == b
True
>>> a is b
True
```

Figure 6: Objects `a` and `b` are both equal in value and pointing to the same memory location.

From figure 6 above, we can see that indeed objects `a` and `b` are pointing to the same `"hello"` object. But what happens if we add the word `"bob"` to object `a` using the `+` operator and store the result back to `a`? So object `a` now holds the value `"hellobob"`. Refer to figure 7 below.

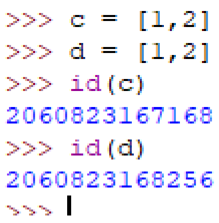


```
>>> a = a + "bob"
>>> a
'hellobob'
>>> id(a)
2060822737712
>>> id(b)
2060822867824
>>>
```

Figure 7: Changes to object `a` creates a new object.

Logically speaking, we would expect that the value of object `a` changes and its memory address to not change. But that is not what had happened. From figure 7 above, we can see that the memory address of object `a` has changed (remember that both object `a` and `b` are pointing to the same memory address in figure 4). This change in memory address is because the word "hello" is of the type *String* and the string type is *immutable* in Python. In other words, each time a change is made to an immutable object, Python creates a new object with a new memory address and returns it while the previous object is **not** changed.

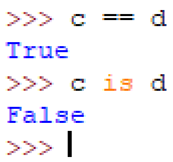
Let's look at what happens with a mutable object. We now use the objects `c` and `d` that has `list` type values assigned to it. Again we are going to look at their memory addresses with the `id()` function.



```
>>> c = [1,2]
>>> d = [1,2]
>>> id(c)
2060823167168
>>> id(d)
2060823168256
>>>
```

Figure 8: Memory addresses of object `c` and `d`.

From figure 8 above, we can tell that something is definitely different with `list` type values. We are still going to use the `==` comparison operator and the `is` identity operator to check if objects `c` and `d` are the same. Refer to figure 9 below.



```
>>> c == d
True
>>> c is d
False
>>>
```

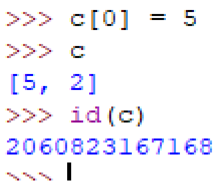
Figure 9: Object `c` and `d` are equal in value but not equal in memory locations.

The results is that the values are the same but the memory addresses are different. That is what we expect based on the memory addresses returned by the `id()` function. This means that both objects `c` and `d` contains 2 separate `list` objects, even though their values are the same. Refer to figure 10 below.

$$\begin{tikzcd}
c \arrow[r] & {[1,2]} \\
d \arrow[r] & {[1,2]}
\end{tikzcd}$$

Figure 10: Object `c` and `d` have different memory locations.

Due to the separate memory addresses, changes made to any of the objects `c` or `d` are tracked individually and thus manipulating a value in one object will **not effect** the other object's value **nor** will Python create a new object when the previous value has changed. As seen in figure 11 below.



```
>>> c[0] = 5
>>> c
[5, 2]
>>> id(c)
2060823167168
>>>
```

Figure 11: Changing elements in object `c` does not effect its memory location.

In summary, mutable objects are characterized by different memory address and changing their values do not cause any new objects to be created but immutable objects are the exact opposite. The reason why we need to know this concept is because as a developer, we need to know exactly what is going on with our objects as this effects how our future programs are developed in Python.

With this concept of mutability in mind, we can now get to know the various datatypes Python has to offer. In addition, each datatype in Python is an *object*, with a set of attributes and behaviours associated with it. For more in-depth information about each Python datatype and their attributes and behaviours, refer to the *Built-in Types* section of the Python Standard Library reference docs here.

Name of Datatype	Type in Python Library	Mutable?	Description	Example
None	`NoneType`	No	This is a constant that is frequently used to represent the absence of a value, as when default arguments are not passed to a function.	None
Boolean	`bool`	No	These store only the values `True` or `False`. The value `True` can also be represented by any number, positive or negative on the number line and the value `False` can also be represented as any integer, float or complex number set to zero.	True, False
Integer	`int`	No	These are the whole numbers on the number line. Both negative and positive numbers.	999, 861, -542
Floating Point	`float`	No	These are the numbers with the decimal points. Both negative and positive floating point numbers.	58.456, -96.584, 94.4e8
Complex	`complex`	No	These are numbers in the complex plane. Both negative and positive complex numbers.	4j-8, -4j+8
String	`str`	No	These are text strings. ASCII and Unicode	'hello', "hello", """hello"""

List	`list`	Yes	These are containers that are able to store multiple types of data thus creating a list. It is index (position) driven and you will need to know the particular data's position in the list to retrieve the value.	[999, 'hello', -542]
Tuple	`tuple`	No	These are read-only lists. Note the difference in the brackets from the Lists datetype.	(999, 'hello', -542)
Bytes	`bytes`	No	These are 8 bits long data. Data can be from Strings, numbers, list, etc then coverted into bytes.	b'ab\xff'
BytesArray	`bytearray`	Yes	These are an array of 8 bits long data. Data can be from Strings, numbers, list, etc then coverted into bytearray.	bytearray(b'hello')
Set	`set`	Yes	These looks like tuples but it does not behave like a tuple. It is only for homogeneous data and the data are unordered.	set([999, 145, -542])
Frozen Set	`frozenset`	no	These are the immutable versions of the `set` datatype.	freozenset([999, 145, -542])
Dictionary	`dict`	Yes	These are containers that consist of key-value pairs. As such, they are key driven and a key is required to retrieve a value.	{'name': 'john', 'code':6734, 'dept': 'sales'}

A method to find out which internal Python datatype a literal or a variable is, is to use the `type()` built-in function. To find out if a literal or a variable is of a specific datatype, the `isinstance()` function can be used.

```
type(7)  # <class 'int'>
type(7) == int  # True
isinstance(7, int)  # True
```

Conversion from one datatype object to another datatype object is through the use of its object name as stated in the column *Type in Python Library* from the table above. The most used conversions are generally from strings to numbers and vice versa. Example

```python
str(5)  # will change the int to a string
float('58.2')  # will change a string to a floating point number
```

Note that due to Python's interpreted properties, changing a variable's type during execution is perfectly legal and sometimes required but it is used with caution as it may cause confusion during code maintenance especially when the variable names were not descriptive.

2.3 Expressions and Statements

An expression as defined in the field of mathematics (from the Oxford Languages Dictionary) as

> a collection of symbols that jointly express a quantity

For instance, the math expression for the area of a circle is πr^2 but in programming terms that same expression would be written as `3.142 * r * r` where `r` is some value.

In Python, **expressions** are lines of code that consist of only operands and no operators. The main property of expressions is that it **must always** return at least one value. Examples of expressions are shown below

```python
# variables
a = 10
b = 20
str_val = "hello"

a + b - 9   # expression, output: 21

# where len() is a built-in Python function to return the
# length of the string (for this example), output: 5
print(len(str_val))
```

On the other hand, a statement is defined (from Wikipedia) as

> a syntactic unit of an imperative programming language that expresses some action to be carried out.

In general terms, **a statement** is a complete instruction that a Python interpreter can execute. A statement may or may not return any value therefore an expression is **also** considered a statement. Statements can also be made up of more than one line of code. Examples of these statements are the `if` statement , `while` statement, `for` statement and many more which we will be looking at in the later chapters.

2.4 References

1. Python 3 - Basic Operators, https://www.tutorialspoint.com/python3/python_basic_operators.htm
2. Built-in Types, https://docs.python.org/3/library/stdtypes.html
3. Lubanovic, 2019, 'Chapter 2. Data: Types, Values, Variables, and Names' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.