Introduction to Python Programming

Hello World

The first thing we do while learning any Programming Language is printing Hello World on to the screen

So we start learning python with printing Hello World

In [1]:

```
print("Hello World!")
```

```
Hello World!
```

Numbers and more in Python!

In this Session, we will learn about numbers in Python and how to use them.

We'll learn about the following topics:

```
1.) Types of Numbers in Python
2.) Basic Arithmetic
3.) Differences between classic division and floor division
4.) Object Assignment in Python
```

Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Throughout this course we will be mainly working with integers or simple float number types.

Here is a table of the two main types we will spend most of our time working with some examples:

Examples	Number "Type"
1,2,-5,1000	Integers
1.2,-0.5,2e2,3E2	Floating-point numbers

Now let's start with some basic arithmetic.

Basic Arithmetic

In [2]:

```
# Addition
2+1
```

Out[2]:

```
3
```

In [3]:

```
# Subtraction
2-1
```

Out[3]:

```
1
```

In [4]:

```
# Multiplication
2*2
```

Out[4]:

```
4
```

In [5]:

```
# Division
3/2
```

Out[5]:

```
1.5
```

In [6]:

```
# Floor Division
7//4
```

Out[6]:

```
1
```

Whoa! What just happened? Last time I checked, 7 divided by 4 equals 1.75 not 1!

The reason we get this result is because we are using "*floor*" division. The // operator (two forward slashes) truncates the decimal without rounding, and returns an integer result.

So what if we just want the remainder after division?

In [7]:

```
# Modulo
7%4
```

Out[7]:

```
3
```

4 goes into 7 once, with a remainder of 3. The % operator returns the remainder after division.

Arithmetic continued

In [8]:

```
# Powers
2**3
```

Out[8]:

```
8
```

In [9]:

```
# Can also do roots this way
4**0.5
```

Out[9]:

```
2.0
```

In [10]:

```
# Order of Operations followed in Python
2 + 10 * 10 + 3
```

Out[10]:

```
105
```

In [11]:

```
# Can use parentheses to specify orders
(2+10) * (10+3)
```

Out[11]:

```
156
```

Variables

- Variables are containers for storing data values.
- Unlike other programming languages, Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.

Rules for variable names

- names can not start with a number
- names can not contain spaces, use _ intead
- names can not contain any of these symbols:

```
:'",<>/?|\!@#%^&*~-+
```

- it's considered best practice (PEP8 (https://www.python.org/dev/peps/pep-0008/#function-and-variable-names)) that names are lowercase with underscores
- avoid using Python built-in keywords like `list` and `str`
- avoid using the single characters `l` (lowercase letter el), `O` (uppercase letter oh) and `I` (uppercase letter eye) as they can be confused with `1` and `0`

Assigning Variables

Variable assignment follows `name = object`, where a single equals sign `=` is an *assignment operator*

In [12]:

```
# Let's create an object called "a" and assign it the number 5
a = 5
a
```

Out[12]:

```
5
```

Here we assigned the integer object `5` to the variable name `a`.
Now if I call *a* in my Python script, Python will treat it as the number 5.

Reassigning Variables

Python lets you reassign variables with a reference to the same object.

What happens on reassignment? Will Python let us write it over?

In [13]:

```
# Reassignment
a = 10
```

In [14]:

```
# Check
a
```

Out[14]:

```
10
```

In [15]:

```
# Adding the objects
a+a
```

Out[15]:

```
20
```

Yes! Python allows you to write over assigned variable names. We can also use the variables themselves when doing the reassignment. Here is an example of what I mean:

Dynamic Typing

Python uses *dynamic typing*, meaning you can reassign variables to different data types. This makes Python very flexible in assigning data types; it differs from other languages that are *statically typed*.

In [16]:

```
val = 2
```

In [17]:

```
val
```

Out[17]:

```
2
```

In [18]:

```
val='Python'
```

In [19]:

```
val
```

Out[19]:

```
'Python'
```

Pros and Cons of Dynamic Typing

Pros of Dynamic Typing

- very easy to work with
- faster development time

Cons of Dynamic Typing

- may result in unexpected bugs!
- you need to be aware of `type()`

In [20]:

```
a = a + 10
```

In [21]:

```
a
```

Out[21]:

```
20
```

There's actually a shortcut for this. Python lets you add, subtract, multiply and divide numbers with reassignment using `+=`, `-=`, `*=`, and `/=`.

In [22]:

```
a += 10
```

In [23]:

```
a
```

Out[23]:

```
30
```

In [24]:

```
a *= 2
```

In [25]:

```
a
```

Out[25]:

```
60
```

Determining variable type with `type()`

You can check what type of object is assigned to a variable using Python's built-in `type()` function. Common data types include:

- **int** (for integer)
- **float**
- **str** (for string)
- **list**
- **tuple**
- **dict** (for dictionary)
- **set**
- **bool** (for Boolean True/False)

In [26]:

```
type(a)
```

Out[26]:

```
int
```

In [27]:

```
a = 2.5
```

In [28]:

```
type(a)
```

Out[28]:

```
float
```

Simple Exercise

This shows how variables make calculations more readable and easier to follow.

In [29]:

```
my_income = 100
tax_rate = 0.1
my_taxes = my_income * tax_rate
```

In [30]:

```
my_taxes
```

Out[30]:

```
10.0
```

Operators in Python

1. Arithmetic operators
2. Comparison operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Identity operators
7. Membership operators

1. Arithmetic operators

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

2. Comparision operators

Operator	Meaning	Example
>	Greater that - True if left operand is greater than the right	x > y
<	Less that - 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

3. Logical operators

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

4. Bitwise operators

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`)

5. Assignment operators

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

6. Identity operators in Python

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

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

Python Namespace and Scope

What is Name in Python?

Name (also called identifier) is simply a name given to objects. Everything in Python is an object. Name is a way to access the underlying object.

For example, when we do the assignment a = 2, here 2 is an object stored in memory and a is the name we associate it with. We can get the address (in RAM) of some object through the built-in function, id(). Let's check it.

In [31]:

```
# Note: You may get different value of id

a = 2
print('id(2) =', id(2))

print('id(a) =', id(a))
```

```
id(2) = 140732356531040
id(a) = 140732356531040
```

Here, both refer to the same object. What is happening in the above sequence of steps? A diagram will help us explain this.

Memory diagram of a variable

Initially, an object 2 is created and the name a is associated with it, when we do a = a+1, a new object 3 is created and now a associates with this object.

Note that id(a) and id(3) have same values.

Furthermore, when we do b = 2, the new name b gets associated with the previous object 2.

This is efficient as Python doesn't have to create a new duplicate object. This dynamic nature of name binding makes Python powerful; a name could refer to any type of object

What is a Namespace in Python?

So now that we understand what names are, we can move on to the concept of namespaces.

To simply put it, namespace is a collection of names.

In Python, you can imagine a namespace as a mapping of every name, you have defined, to corresponding objects.

Different namespaces can co-exist at a given time but are completely isolated.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long we don't exit.

This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program. Each module creates its own global namespace.

These different namespaces are isolated. Hence, the same name that may exist in different modules do not collide.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar, is the case with class. Following diagram may help to clarify this concept.

So what have we learned? We learned some of the basics of numbers in Python. We also learned how to do arithmetic and use Python as a basic calculator. We then wrapped it up with learning about Variable Assignment in Python.

Up next we'll learn about Strings!