Module 01. Introduction

September 26, 2018

1 Introduction to Python Programming

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

We generally write a computer program using a high-level language. A high-level language is one which is understandable by us humans. It contains words and phrases from the English (or other) language. But a computer does not understand high-level language. It only understands program written in 0's and 1's in binary, called the machine code. A program written in high-level language is called a source code. We need to convert the source code into machine code and this is accomplished by compilers and interpreters. Hence, a compiler or an interpreter is a program that converts program written in high-level language into machine code understood by the computer.

Interpreter Translates program one statement at a time.

It takes less amount of time to analyze the source code but the overall execution time is slower.

No intermediate object code is generated, hence are memory efficient.

Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.

Programming language like Python, Ruby use interpreters.

Compiler Scans the entire program and translates it as a whole into machine code.

It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.

Generates intermediate object code which further requires linking, hence requires more memory.

It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.

Programming language like C, C++ use compilers.

1.0.1 Python 2 vs Python 3

Today, two versions of Python are available: Python 2 and the newer Python 3. Every programming language evolves as new ideas and technologies emerge, and the developers of Python have continually made the language more versatile and powerful. Most changes are incremental and hardly noticeable, but in some cases, code written for Python 2 may not run properly on systems with Python 3 installed.

1.0.2 Python on Different Operating Systems

Python is a cross-platform programming language, which means it runs on all the major operating systems. Any Python program you write should run on any modern computer that has Python installed. However, the methods for setting up Python on different operating systems vary slightly.

Python Editors – 1. IDLE 2. Jupyter notebook 3. Geany – ubuntu 4. Sublime text – OSX

1.0.3 Hello World!

A long-held belief in the programming world has been that printing a Hello world! message to the screen as your first program in a new language will bring you luck.

In Python, you can write the Hello World program in one line:

print("Hello world!")

Such a simple program serves a very real purpose. If it runs correctly on your system, any Python program you write should work as well.

Code:

```
In [1]: print("Hello World!!!")
```

```
Hello World!!!
```

2 Defining a variable

A variable is a name that refers to a value. The assignment statement creates new variables and gives them values

2.0.1 Naming and Using Variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand. Be sure to keep the following variable rules in mind:

Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable message_1 but not 1_message.

Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, greeting_message works, but greeting message will cause errors.

Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word print.

Variable names should be short but descriptive. For example, name is better than n, student_name is better than s_n, and name_length is better than length_of_persons_name.

Be careful when using the lowercase letter l and the uppercase letter O because they could be confused with the numbers 1 and 0.

2.0.2 Statements

A statement is an instruction that the Python interpreter can execute.

Code:

```
In [ ]: x = 10 # Creating a variable

        print("Value of x is :", x) # Print the variable
```

2.0.3 input() - Taking user input

Code:

```
In [ ]: x = input()

        print("value of x = ", x)
```

Code:

```
In [ ]: x = input("Enter a value : ")
        print("value of x = ", x)
```

2.0.4 type() - tells the data type of value

Code:

```
In [ ]: x = 10
        print("type of x = ",type(x))
```

Code:

```
In [ ]: x = 10.5
        print(type(x))
```

Code:

```
In [ ]: x = "abc"
        print(type(x))
```

Code:

```
In [ ]: x = False
        print(type(x))
```

3 Evaluating expressions

An expression is a combination of values, variables, and operators.

3.0.1 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are called operands.

1. Arithmetic Operators
Assume variable a holds 10 and variable b holds 20, then
Operator
Description
Example

- Addition
  Adds values on either side of the operator.
  a + b = 30
  - Subtraction
    Subtracts the right-hand operand from left-hand operand.
    a – b = -10
    * Multiplication
      Multiplies values on either side of the operator
      a * b = 200
      / Division
      Divides left hand operand by right hand operand
      b / a = 2
      % Modulus
      Divides left-hand operand by right-hand operand and returns remainder
      b % a = 0
      ** Exponent
      Performs exponential (power) calculation on operators
      a**b =10 to the power 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 negative, the result is floored, i.e., rounded away from zero (towards negative infinity)
      9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

2. Arithmetic Operators
These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.
Assume variable a holds 10 and variable b holds 20, then
Operator
Description
Example
==
If the values of two operands are equal, then the condition becomes true.
(a == b) is not true.

!=

If values of two operands are not equal, then condition becomes true.

(a != b) is true.

<>

If values of two operands are not equal, then condition becomes true.

(a <> b) is true. This is similar to != operator.

>

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.

3. Assignment Operators

Assume variable a holds 10 and variable b holds 20, then

Operator

Description

Example

=

Assigns values from right side operands to left side operand

c = a + b assigns the value of a + b into c

+= Add AND

It adds right operand to the left operand and assigns the result to left operand

c += a is equivalent to c = c + a

-= Subtract AND

It subtracts right operand from the left operand and assigns the result to left operand

c -= a is equivalent to c = c - a

*= Multiply AND

It multiplies right operand with the left operand and assigns the result to left operand

c = *a is equivalent to c = c* a

/= Divide AND

It divides left operand with the right operand and assign the result to left operand

c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a

%= Modulus AND

It takes modulus using two operands and assign the result to left operand

c %= a is equivalent to c = c % a

**= Exponent AND

Performs exponential (power) calculation on operators and assign value to the left operand

c **= a is equivalent to c = c** a

//= Floor Division

It performs floor division on operators and assign value to the left operand

c //= a is equivalent to c = c // a

4. Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows

a = 0011 1100

b = 0000 1101

———————

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Operator

Description

Example

& Binary AND

Operator copies a bit to the result if it exists in both operands

(a & b) (means 0000 1100)

| Binary OR

It copies a bit if it exists in either operand.

(a | b) = 61 (means 0011 1101)

^ Binary XOR

It copies the bit if it is set in one operand but not both.

(a ^ b) = 49 (means 0011 0001)

~ Binary Ones Complement

It is unary and has the effect of 'flipping' bits.

(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.

<< Binary Left Shift

The left operands value is moved left by the number of bits specified by the right operand.

a << 2 = 240 (means 1111 0000)

>> Binary Right Shift

The left operands value is moved right by the number of bits specified by the right operand.

a >> 2 = 15 (means 0000 1111)

5. Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Operator

Description

Example

and Logical AND

If both the operands are true then condition becomes true.

(a and b) is true.

or Logical OR

If any of the two operands are non-zero then condition becomes true.

(a or b) is true.

not Logical NOT

Used to reverse the logical state of its operand.

Not(a and b) is false.

6. Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below

Operator

Description

Example

in

Evaluates to true if it finds a variable in the specified sequence and false otherwise.

x in y, here in results in a 1 if x is a member of sequence y.

not in

Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

x not in y, here not in results in a 1 if x is not a member of sequence y.

7. Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below

Operator

Description

Example

is

Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.

x is y, here is results in 1 if id(x) equals id(y).

is not

Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

x is not y, here is not results in 1 if id(x) is not equal to id(y).

8. Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr.No.

1

**

Exponentiation (raise to the power)

2

~ + -

Complement, unary plus and minus (method names for the last two are +@ and -@)

3

* / % //

Multiply, divide, modulo and floor division

4

+ -

Addition and subtraction

5

>> <<

Right and left bitwise shift

6

&

Bitwise 'AND'

7

^ |
Bitwise exclusive OR' and regularOR'
8
<= < > >=
Comparison operators
9
<> == !=
Equality operators
10
= %= /= //= -= += *= **=
Assignment operators
11
is is not
Identity operators
12
in not in
Membership operators
13
not or and
Logical operators

Ex. Addition of 2 nos

Code:

```
In [ ]: a = input("enter value for a : ")
        b = input("enter value for b : ")
        c = a + b

        print("Addition of ",a,"and",b,"=",c)
```

```
In [ ]: print(type(a),type(b))
```

3.0.2 Type Conversion

Code:

```
In [ ]: a = int("2")
        print(type(a),a)
```

Code:

```
In [ ]: a = int(input("enter value for a : "))
        b = int(input("enter value for b : "))
        c = a + b

        print("Addition of ",a,"and",b,"=",c)
```

3.0.3 Conversion to int()

Valid cases

Code:

```
In [ ]: a = int("2")
        print(type(a),"value of a : ", a)

        a = int(2.7)
        print(type(a),"value of a : ", a)

        a = int(True)
        print(type(a),"value of a : ", a)
```

Invalid cases

```
In [ ]: # a = int("2.5")
        # print(type(a),"value of a : ", a)  # - ValueError: invalid literal for int() with ba

        # a = int("abc")
        # print(type(a),"value of a : ", a)  # - ValueError: invalid literal for int() with ba
```

3.0.4 - Conversion to float()

Valid cases

```
In [ ]: a = float(10)
        print(type(a),"value of a : ", a)

        a = float("10")
        print(type(a),"value of a : ", a)

        a = float("10.5")
        print(type(a),"value of a : ", a)

        a = float(False)
        print(type(a),"value of a : ", a)
```

Invalid cases

```
In [ ]: # a = float("abc")
        # print(type(a),"value of a : ", a)  # - ValueError: could not convert string to float
```

3.0.5 - Conversion to str()

```
In [ ]: a = str(10)
        print(type(a),"value of a : ", a)
```

3.0.6 - Conversion to bool()

```
In [ ]: a = bool(10)
        print(type(a),"value of a : ", a)

        a = bool(0)
        print(type(a),"value of a : ", a)

        a = bool(10.2)
        print(type(a),"value of a : ", a)

        a = bool(0.0)
        print(type(a),"value of a : ", a)

        a = bool("abc")
        print(type(a),"value of a : ", a)

        a = bool("False")
        print(type(a),"value of a : ", a)
```

3.0.7 - Importing a module (3-ways)

```
In [ ]: import math

        math.sqrt(8)

In [ ]: import math as m
        m.sqrt(8)

In [ ]: from math import *

        sqrt(8)
```

Ex. Write a Python program to compute the distance between the points (x1, y1) and (x2, y2).

```
In [ ]: import math

        x1,y1,x2,y2 = 3,4,5,6
        x  = (x1**2 - x2**2) + (y1**2 - y2**2)
        dist = math.sqrt(abs(x))
        print(dist)
```

Ex. Write a python program to calculate the hypoteneous of a right angled triangle when sides are given

```
In [ ]: a,b = 3,4

        h = math.sqrt((a**2)+(b**2))
        print(int(h))
```

Ex. Write a Python program which accepts the radius of a circle from the user and computes the area.

```
In [ ]: import math as m
        radius = int(input("Enter radius of a circle : "))
        area = m.pi * (radius ** 2)
        print("Area of circle with radius {} = {}".format(radius,round(area, 2)))
```

Ex. Write a Python program to get the third side of right angled triangle from two given sides.

```
In [ ]: import math

        a, b = int(input("Input a : ")),int(input("Input b : "))
        h = int(math.sqrt((a**2) + (b**2)))
        print(h)
```

Ex. Write a Python program to generate a random number between 1 to 10

```
In [ ]: import random as r

        random_number = r.randint(1,10)

        print(random_number)
```

Ex. Write a Python program to print the calendar of a given month and year.

```
In [ ]: import calendar

        y = int(input("Input the year : "))
        m = int(input("Input the month in numbers : "))
        print(calendar.month(y, m))
```

Ex. Write a Python script to display the -

1. Current date and time
2. Current year
3. Month of year
4. Week number of the year
5. Weekday of the week
6. Day of year
7. Day of the month
8. Day of week
9. Week number

```
In [ ]: import time
        import datetime
        print("Current date and time: " , datetime.datetime.now())
        print("Current year: ", datetime.date.today().strftime("%Y"))
        print("Month of year: ", datetime.date.today().strftime("%B"))
```

```
        print("Week number of the year: ", datetime.date.today().strftime("%W"))
        print("Weekday of the week: ", datetime.date.today().strftime("%w"))
        print("Day of year: ", datetime.date.today().strftime("%j"))
        print("Day of the month : ", datetime.date.today().strftime("%d"))
        print("Day of week: ", datetime.date.today().strftime("%A"))
        print(datetime.date(2015, 6, 16).isocalendar()[1])
```

```
In [ ]: import time

        start_time = time.time()

        # Some code here

        end_time = time.time()
        print( end_time - start_time)
```

3.0.8 - An Introduction to PEP-8

The Python programming language has evolved over the past year as one of the most favourite programming languages. This language is relatively easy to learn than most of the programming languages. It is a multi-paradigm, it has lots of open source modules that add up the utility of the language and it is gaining popularity in data science and web development community.

However, you can use the benefits of Python only when you know how to express better with your code. Python was made with some goals in mind, these goals can be seen when you type "import this".

3.0.9 - The Zen of Python

The Python community's philosophy is contained in "The Zen of Python" by Tim Peters. You can access this brief set of principles for writing good Python code by entering import this into your interpreter. I won't reproduce the entire "Zen of Python".

```
In [ ]: import this
```

The above are the 20 principles that Python programming uses. You also see "Readability Counts" in the output above, which should be your main concern while writing code: other programmers or data scientists should understand and should be able to contribute to your code so that it can solve the task at hand.

Indentation

When programming in Python, indentation is something that you will definitely use. However, you should be careful with it, as it can lead to syntax errors. The recommendation is therefore to use 4 spaces for indentation. For example, this statement uses 4 spaces of indentation:

And also this for loop with print statement is indented with 4 spaces:

When you write a big expression, it is best to keep the expression vertically aligned. When you do this, you'll create a "hanging indent".

Here are some examples of the hanging indent in big expressions, which show some variations of how you can use it:

Every developer, working with Python or another programming language, asks him or herself the question at some point whether to use tabs or spaces for indentation. The difference between tabs and spaces is an ongoing discussion in the community.

Generally, spaces are the preferred indentation means but if you find some Python scripts already using the tabs, you should go on doing indentation with tabs. Otherwise, you should change the indentation of all the expressions in your script with spaces.

Note that Python 3 doesn't allow mixing tabs and spaces for indentation. That's why you should choose one of the two and stick with it!

Maximum Line Length

Generally, it's good to aim for a line length of 79 characters in your Python code.

Following this target number has many advantages. A couple of them are the following:

It is possible to open files side by side to compare;

You can view the whole expression without scrolling horizontally which adds to better readability and understanding of the code.

Comments should have 72 characters of line length. You'll learn more about the most common conventions for comments later on in this tutorial!

In the end, it is up to you what coding conventions and style you like to follow if you are working in a small group and it is acceptable for most of the developers to divert from the maximum line length guideline. However, if you are making or contributing to an open source project, you'll probably want and/or need to comply with the maximum line length rule that is set out by PEP-8.

While using the + operator, you can better use a proper line break, which makes your code easier to understand:

You should use…

You should avoid…

Alternatively, you could also write:

In short, you can add a break before or after a binary operator, as long as you are consistent. If you're writing new code, you should try to follow the last option that was presented, where you add a break before the binary operator.

Blank Lines

In Python scripts, top-level function and classes are separated by two blank lines. Method definitions inside classes should be separated by one blank line. You can see this clearly in the following example:

The classes SwapTestSuite and OddOrEvenTestSuite are separated by two blank lines, whereas the method definitions, such as .setUp() and .test_swap_operations() only have one blank line to separate them.

Whitespaces in Expressions and Statements

You should try to avoid whitespaces when you see your code is written just like in the following examples:

You should use…

You should avoid…

or

or

Source File Encoding

A computer cannot store "letters", "numbers", "pictures" or anything else; It can only store and work with bits, which can only have binary values: yes or no, true or false, 1 or 0, etc. As you already know, a computer works with electricity; This means that an "actual" bit is the presence or absence of a blip of electricity. You would usually represent this (lack of) presence with 1 and 0.