

INFORMATION SHEET PL 101-7.1.1
“Python Statement, Indentation, Comments, Variables, Constants”

In this lesson, you will learn about Python statements, why indentation is important and use of comments in programming, you will also learn about Python variables, constants, literals and their use cases.

References:

- Python Programming for Beginners

--	--	--	--	--

INFORMATION SHEET PL 101-7.1.1

“Python Statement, Indentation, Comments, Variables, Constants and Literals”

Python Statement

Instructions that a Python interpreter can execute are called **statements**. For example, `a = 1` is an **assignment statement**. **if statement**, **for statement**, **while statement**, etc. are other kinds of statements which will be discussed later.

Multi-line statement

In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line **continuation character** (`\`). For example:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

This is an **explicit line continuation**. In Python, line continuation is implied inside **parentheses** `()`, **brackets** `[]`, and **braces** `{}`. For instance, we can implement the above multi-line statement as:

```
a = (1 + 2 + 3 +
    4 + 5 + 6 +
    7 + 8 + 9)
```

Here, the surrounding parentheses `()` do the line continuation implicitly. Same is the case with `[]` and `{}`. For example:

```
colors = ['red',
          'blue',
          'green']
```

We can also put multiple statements in a single line using semicolons, as follows:

--	--	--	--	--


```
a = 1; b = 2; c = 3
```

Python Indentation

Most of the programming languages like C, C++, and Java use braces `{ }` to define a **block of code**. Python, however, uses **indentation**.

A code block (body of a function, loop, etc.) **starts with indentation and ends with the first unindented line**. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally, four whitespaces are used for indentation and are preferred over tabs. Here is an example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

The enforcement of indentation in Python makes the code look **neat and clean**. This results in Python programs that look similar and consistent.

Indentation can be ignored in line continuation, but it's always a good idea to indent. It makes the **code more readable**. For example:

```
if True:
    print('Hello')
    a = 5
```

and

```
if True: print('Hello'); a = 5
```

--	--	--	--	--

both are valid and do the same thing, but the former style is clearer.

Incorrect indentation will result in `IndentationError`.

Python Comments

Comments are very important while writing a program. They describe what is going on inside a program, so that a person looking at the source code does not have a hard time figuring it out.

You might forget the key details of the program you just wrote in a month's time. So taking the time to explain these concepts in the form of comments is always fruitful.

In Python, we use the hash (#) symbol to start writing a comment.

It extends up to the newline character. Comments **are for programmers to better understand a program**. Python Interpreter **ignores comments**.

```
#This is a comment
#print out Hello
print('Hello')
```

Multi-line comments

We can have comments that extend up to multiple lines. One way is to use the hash(#) symbol at the beginning of each line. For example:

--	--	--	--	--


```
#This is a long comment
#and it extends
#to multiple lines
```

Another way of doing this is to use triple quotes, either `'''` or `"""`.

These triple quotes are generally used for **multi-line strings**. But they can be used as a multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a
perfect example of
multi-line comments"""
```

Docstrings in Python

A **docstring** is short for documentation string.

Python docstrings (documentation strings) are the string literals that appear right after the definition of a function, method, class, or module.

Triple quotes are used while writing docstrings. For example:

```
def double(num):
    """Function to double the value"""
    return 2*num
```

Docstrings appear right after the definition of a function, class, or a module. This separates docstrings from multiline comments using triple quotes.

The docstrings are associated with the object as their `__doc__` attribute.

So, we can access the docstrings of the above function with the following lines of code:

```
def double(num):
```

--	--	--	--	--


```

"""Function to double the value"""
return 2*num
print(double.__doc__)

```

Output

```

Function to double the value

```

Python Variables

A variable is a **named location used to store data in the memory**. It is helpful to think of variables as a **container that holds data that can be changed later in the program**. For example,

```

number = 10

```

Here, we have created a variable named `number`. We have assigned the value `10` to the variable.

You can think of variables as a **bag to store books in it and that book can be replaced at any time**.

```

number = 10
number = 1.1

```

Initially, the value of `number` was `10`. Later, it was changed to `1.1`.

Note: In Python, we don't actually assign values to the variables. Instead, Python gives the reference of the object(value) to the variable.

--	--	--	--	--

Assigning values to Variables in Python

As you can see from the above example, you can use the assignment operator `=` to assign a value to a variable.

Example 1: Declaring and assigning value to a variable

```
website = "apple.com"
print(website)
```

Output

```
apple.com
```

In the above program, we assigned a value `apple.com` to the variable `website`. Then, we printed out the value assigned to `website` i.e. `apple.com`

Note: Python is a type-inferred language, so you don't have to explicitly define the variable type. It automatically knows that `apple.com` is a **string** and declares the `website` variable as a string.

Example 2: Changing the value of a variable

```
website = "apple.com"
print(website)

# assigning a new value to website
website = "samsung.com"

print(website)
```

Output

--	--	--	--	--


```
apple.com
samsung.com
```

In the above program, we have assigned `apple.com` to the `website` variable initially. Then, the value is changed to `samsung.com`.

Example 3: Assigning multiple values to multiple variables

```
a, b, c = 5, 3.2, "Hello"

print (a)
print (b)
print (c)
```

If we want to assign the same value to multiple variables at once, we can do this as:

```
x = y = z = "same"

print (x)
print (y)
print (z)
```

The second program assigns the `same` string to all the three variables `x`, `y` and `z`.

Constants

A **constant** is a type of variable whose *value cannot be changed*. It is helpful to think of constants as containers that hold information which **cannot be changed later**.

--	--	--	--	--

You can think of constants as a bag to store some books which cannot be replaced once placed inside the bag.

Assigning value to constant in Python

In Python, constants are usually **declared and assigned in a module**. Here, the module is a new file containing variables, functions, etc. which is imported to the **main file**. Inside the module, constants are written in all capital letters and underscores separating the words.

Example 3: Declaring and assigning value to a constant

Create a **constant.py**:

```
PI = 3.14
GRAVITY = 9.8
```

Create a **main.py**:

```
import constant
print(constant.PI)
print(constant.GRAVITY)
```

Output

```
3.14
9.8
```

In the above program, we create a **constant.py** module file. Then, we assign the constant value to `PI` and `GRAVITY`. After that, we create a **main.py** file and import the `constant` module. Finally, we print the constant value.

--	--	--	--	--

Note: In reality, we don't use constants in Python. Naming them in all **capital letters** is a convention to separate them from variables, however, it does not actually prevent reassignment.

Rules and Naming Convention for Variables and constants

1. Constant and variable names should have a combination of letters in lowercase (a to z) or uppercase (**A to Z**) or digits (**0 to 9**) or an underscore (**_**). For example:

snake_case

MACRO_CASE

camelCase

CapWords

2. Create a name that makes sense. For example, `vowel` makes more sense than `v`.
3. If you want to create a variable name having two words, use underscore to separate them. For example:

1. `my_name`

`current_salary`

4. Use capital letters possible to declare a constant. For example:

--	--	--	--	--

2. PI
3. G
4. MASS
5. SPEED_OF_LIGHT

TEMP

5. Never use special symbols like !, @, #, \$, %, etc.
6. Don't start a variable name with a digit.

Literals

Literal is a **raw data given in a variable or constant**. In Python, there are various types of literals they are as follows:

Numeric Literals

Numeric Literals are **immutable (unchangeable)**. Numeric literals can belong to 3 different numerical types: `Integer`, `Float`, and `Complex`.

Example 4: How to use Numeric literals in Python?

```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal

#Float Literal
float_1 = 10.5
float_2 = 1.5e2
```

--	--	--	--	--


```
#Complex Literal
x = 3.14j

print(a, b, c, d)
print(float_1, float_2)
print(x, x.imag, x.real)
```

Output

```
10 100 200 300
10.5 150.0
3.14j 3.14 0.0
```

In the above program,

- We assigned integer literals into different variables. Here, `a` is binary literal, `b` is a decimal literal, `c` is an octal literal and `d` is a hexadecimal literal.
- When we print the variables, all the literals are **converted into decimal values**.
- `10.5` and `1.5e2` are floating-point literals. `1.5e2` is expressed with exponential and is equivalent to `1.5 * 102`.
- We assigned a complex literal i.e `3.14j` in variable `x`. Then we use **imaginary** literal (`x.imag`) and **real** literal (`x.real`) to create imaginary and real parts of complex numbers.
- To learn more about Numeric Literals, refer to Python Numbers.

String literals

A **string literal** is a **sequence of characters surrounded by quotes**. We can use both single, double, or triple quotes for a string. And, a character literal is a single character surrounded by **single or double quotes**.

--	--	--	--	--

Example 7: How to use string literals in Python?

```
strings = "This is Python"
char = "C"
multiline_str = """This is a multiline string with more than one line code."""
unicode = u"\u00dcnic\u00f6de"
raw_str = r"raw \n string"

print(strings)
print(char)
print(multiline_str)
print(unicode)
print(raw_str)
```

Output

```
This is Python
C
This is a multiline string with more than one line code.
Ünicöde
raw \n string
```

In the above program, `This is Python` is a string literal and `C` is a character literal.

The value in triple-quotes `"""` assigned to the `multiline_str` is a multi-line string literal.

The string `u"\u00dcnic\u00f6de"` is a Unicode literal which supports characters other than English. In this case, `\u00dc` represents `Ü` and `\u00f6` represents `ö`.

`r"raw \n string"` is a raw string literal.

Boolean literals

A **Boolean literal** can have any of the two values: `True` or `False`.

Example 8: How to use boolean literals in Python?

```
x = (1 == True)
y = (1 == False)
```

--	--	--	--	--


```
a = True + 4
b = False + 10

print("x is", x)
print("y is", y)
print("a:", a)
print("b:", b)
```

Output

```
x is True
y is False
a: 5
b: 10
```

In the above program, we use boolean literal `True` and `False`. In Python, `True` represents the value as `1` and `False` as `0`. The value of `x` is `True` because `1` is equal to `True`. And, the value of `y` is `False` because `1` is not equal to `False`.

Similarly, we can use the `True` and `False` in numeric expressions as the value. The value of `a` is `5` because we add `True` which has a value of `1` with `4`. Similarly, `b` is `10` because we add the `False` having value of `0` with `10`.

Special literals

Python contains one **special literal** i.e. `None`. We use it to specify that the field has not been created.

Example 9: How to use special literals in Python?

```
drink = "Available"
food = None

def menu(x):
    if x == drink:
        print(drink)
```

--	--	--	--	--


```

else:
    print(food)

menu(drink)
menu(food)

```

Output

```

Available
None

```

In the above program, we define a `menu` function. Inside `menu`, when we set the argument as `drink` then, it displays `Available`. And, when the argument is `food`, it displays `None`.

Literal Collections

There are four different literal collections **List literals**, **Tuple literals**, **Dict literals**, and **Set literals**.

Example 10: How to use literals collections in Python?

```

fruits = ["apple", "mango", "orange"] #list
numbers = (1, 2, 3) #tuple
alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary
vowels = {'a', 'e', 'i', 'o', 'u'} #set

print(fruits)
print(numbers)
print(alphabets)
print(vowels)

```

Output

```

['apple', 'mango', 'orange']
(1, 2, 3)

```

--	--	--	--	--


```
{'a': 'apple', 'b': 'ball', 'c': 'cat'}
{'e', 'a', 'o', 'i', 'u'}
```

In the above program, we created a list of `fruits`, a tuple of `numbers`, a dictionary `dict` having values with keys designated to each value and a set of `vowels`.

STUDENT NAME: _____

SECTION: _____

PERFORMANCE TASK PL 101-7.1.1	
WRITTEN WORK TITLE: "Comment and Variables"	
WRITTEN TASK OBJECTIVE:	
MATERIALS: <ul style="list-style-type: none"> Pen and Paper 	
TOOLS & EQUIPMENT: <ul style="list-style-type: none"> None 	
ESTIMATED COST: None	
Instruction: <ol style="list-style-type: none"> 1. Create a Python program that assign "I want to be the best programmer!" to str_1 variable and display it on screen. 2. Create a Python program that assign "I Love Programming!" to str_2 variable and display it on screen. 3. Add a single line comment for each number above 	
PRECAUTIONS: <ul style="list-style-type: none"> Do not just copy all your output from the internet. 	

--	--	--	--	--

- Use citation and credit to the owner if necessary.

ASSESSMENT METHOD: WRITTEN WORK CRITERIA CHECKLIST

--	--	--	--	--

STUDENT NAME: _____

SECTION: _____

PERFORMANCE OUTPUT CRITERIA CHECKLIST PL 101-7.1.1

Did I . . .	CRITERIA	SCORING				
		1	2	3	4	5
1.	Focus - The single controlling point made with an awareness of a task about a specific topic.					
2.	Content - The presentation of ideas developed through facts, examples, anecdotes, details, opinions, statistics, reasons, and/or opinions					
3.	Organization - The order developed and sustained within and across paragraphs using transitional devices and including the introduction and conclusion.					
4.	Style - The choice, use, and arrangement of words and sentence structures that create tone and voice.					
5.	.					
6.	.					
7.	.					
8.	.					
9.	.					
10.	.					
TEACHER'S REMARKS: <input type="checkbox"/> QUIZ <input type="checkbox"/> RECITATION <input type="checkbox"/> PROJECT						
		GRADE:				

- 5 - Excellently Performed
- 4 - Very Satisfactorily Performed
- 3 - Satisfactorily Performed
- 2 - Fairly Performed
- 1 - Poorly Performed

TEACHER

Date: _____

--	--	--	--	--