Python Fundamentals

Week 2

Python Syntax

- Can be executed by writing directly in the Command Line:

```
>>> 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\Your Name>python myfile.py
```

Python Indentation

Refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Indention ndicate a block of code.

```
if 5 > 2:
  print("Five is greater than two!")
```

```
Five is greater than two!
```

Incorrect Indention

- Python will give you an error if you skip the indentation:

```
if 5 > 2:
print("Five is greater than two!")
```

```
  File "demo_indentation_test.py", line 2
    print("Five is greater than two!")
        ^
IndentationError: expected an indented block
```

Whitespaces

- The number of spaces is up to you as a programmer, the most common use is four, but it must be at least one.

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

```
Five is greater than two!
Five is greater than two!
```

Incorrect use of whitespace

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

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

```
  File "demo_indentation2_error.py", line 3
    print("Five is greater than two!")
    ^
IndentationError: unexpected indent
```

Python Keywords

- Keywords are predefined, reserved words used in Python programming that have special meanings to the compiler.

- We cannot use a keyword as a variable name, function name, or any other identifier. They are used to define the syntax and structure of the Python language.

- All the keywords except True, False and None are in lowercase, and they must be written as they are. The list of all the keywords is given below.

Python Keywords

Python Keywords List				
False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Python Identifiers

- Identifiers are the name given to variables, classes, methods, etc. For example,

```
language = 'Python'
```

- Here, language is a variable (an identifier) which holds the value 'Python'.

```
continue = 'Python'
```

- We cannot use keywords as variable names as they are reserved names that are built-in to Python. For example,
- The above code is wrong because we have used continue as a variable name. To learn more about variables, visit Python Variables.

Rules for Naming an Identifier

- Identifiers cannot be a keyword.
- Identifiers are case-sensitive.
- It can have a sequence of letters and digits. However, it must begin with a letter or _. The first letter of an identifier cannot be a digit.
- It's a convention to start an identifier with a letter rather _.
- Whitespaces are not allowed.
- We cannot use special symbols like !, @, #, $, and so on.

Some Valid and Invalid Identifiers in Python

Valid Identifiers	Invalid Identifiers
score	@core
return_value	return
highest_score	highest score
name1	1name
convert_to_string	convert to_string

Things to Remember

- Python is a case-sensitive language. This means, Variable and variable are not the same.

- Always give the identifiers a name that makes sense. While c = 10 is a valid name, writing count = 10 would make more sense, and it would be easier to figure out what it represents when you look at your code after a long gap.

- Multiple words can be separated using an underscore, like this_is_a_long_variable.

Python Naming Conventions

General

- Avoid using names that are too general or too wordy. Strike a good balance between the two.
- Bad: data_structure, my_list, info_map, dictionary_for_the_purpose_of_storing_data_representing_word_definitions
- Good: user_profile, menu_options, word_definitions
- Don’t be a jackass and name things “O”, “l”, or “I”
- When using CamelCase names, capitalize all letters of an abbreviation (e.g. HTTPServer)

Packages

- Package names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names

Modules

- Module names should be all lower case
- When multiple words are needed, an underscore should separate them
- It is usually preferable to stick to 1 word names

Classes

- Class names should follow the UpperCaseCamelCase convention
- Python’s built-in classes, however are typically lowercase words
- Exception classes should end in “Error”

Global (module-level) Variables

- Global variables should be all lowercase
- Words in a global variable name should be separated by an underscore

Instance Variables

- Instance variable names should be all lower case
- Words in an instance variable name should be separated by an underscore
- Non-public instance variables should begin with a single underscore
- If an instance name needs to be mangled, two underscores may begin its name

Methods

- Method names should be all lower case
- Words in an method name should be separated by an underscore
- Non-public method should begin with a single underscore
- If a method name needs to be mangled, two underscores may begin its name

Method Arguments

- Instance methods should have their first argument named ‘self’.
- Class methods should have their first argument named ‘cls’

Functions

- Function names should be all lower case
- Words in a function name should be separated by an underscore

Constants

- Constant names must be fully capitalized
- Words in a constant name should be separated by an underscore

Assigning values to Python Variables

Many Values to Multiple Variables

- Python allows you to assign values to multiple variables in one line:
- Note: Make sure the number of variables matches the number of values, or else you will get an error.

```
x, y, z = "Orange", "Banana", "Cherry"

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

```
Orange
Banana
Cherry
```

One Value to Multiple Variables

- And you can assign the same value to multiple variables in one line:

```
x = y = z = "Orange"

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

```
Orange
Orange
Orange
```

Python Data Types

Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

Getting the Data Type

- You can get the data type of any object by using the type() function:

```
x = 5
print(type(x))
```

```
<class 'int'>
```

Setting the Data Type

- In Python, the data type is set when you assign a value to a variable:

Example	Data Type
`x = "Hello World"`	str
`x = 20`	int
`x = 20.5`	float
`x = 1j`	complex
`x = ["apple", "banana", "cherry"]`	list
`x = ("apple", "banana", "cherry")`	tuple
`x = range(6)`	range
`x = {"name" : "John", "age" : 36}`	dict
`x = {"apple", "banana", "cherry"}`	set
`x = frozenset({"apple", "banana", "cherry"})`	frozenset
`x = True`	bool
`x = b"Hello"`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview
`x = None`	NoneType

Setting the Specific Data Type (Casting)

- If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
`x = str("Hello World")`	str
`x = int(20)`	int
`x = float(20.5)`	float
`x = complex(1j)`	complex
`x = list(("apple", "banana", "cherry"))`	list
`x = tuple(("apple", "banana", "cherry"))`	tuple
`x = range(6)`	range
`x = dict(name="John", age=36)`	dict
`x = set(("apple", "banana", "cherry"))`	set
`x = frozenset(("apple", "banana", "cherry"))`	frozenset
`x = bool(5)`	bool
`x = bytes(5)`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview

How to Run a Python Script

Running Python Scripts

- Python scripts are Python code files saved with a .py extension. You can run these files on any device if it has Python installed on it. They are very versatile programs and can perform a variety of tasks like data analysis, web development, etc.

- You might get these Python scripts if you are a beginner in Python so in this discussion, we will explore various techniques for executing a Python script.

What is Python Script?

- A file containing Python-written code.
- Has the extension ‘.py’ or can also have the extension ‘.pyw’ if it is being run on a Windows 10 machine.

- To run, a Python interpreter is installed on the device.

Running Python Code Demo

Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

Creating a Comment

- Comments starts with a #, and Python will ignore them:

```
#This is a comment.
print("Hello, World!")
```

```
Hello, World!
```

Comments at the end

- Comments can be placed at the end of a line, and Python will ignore the rest of the line:

```
print("Hello, World!") #This is a comment.
```

```
Hello, World!
```

Disabling code with comments

- A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:

```
#print("Hello, World!")
print("Cheers, Mate!")
```

```
Cheers, Mate!
```

Multiline Comments

- Python does not really have a syntax for multiline comments.

- To add a multiline comment, you could insert a # for each line:

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

```
Hello, World!
```

Multiline Strings

- Or, not quite as intended, you can use a multiline string.

- Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

```
Hello, World!
```

Strings in Python

- In computer programming, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

- We use single quotes or double quotes to represent a string in Python. For example,

```
# create a string using double quotes
string1 = "Python programming"

# create a string using single quotes
string1 = 'Python programming'
```

Python String Example

```
# create string type variables

name = "Python"
print(name)

message = "I love Python."
print(message)
```

```
Python
I love Python.
```

Python Multiline String

- For this, we use triple double quotes """ or triple single quotes '''. For example,

```
# multiline string
message = """
Never gonna give you up
Never gonna let you down
"""

print(message)
```

```
Never gonna give you up
Never gonna let you down
```

- In the above example, anything inside the enclosing triple-quotes is one multiline string.

Escape Sequences in Python

- Used to escape some of the characters present inside a string.

- Suppose we need to include both double quote and single quote inside a string,

```
1  example = "He said, "What's there?""
2
3  print(example) # throws error
```

```
ERROR!
File "<string>", line 1
    example = "He said, "What's there?""
                        ^
SyntaxError: unterminated string literal (detected at line 1)
```

Solving the Escape Sequence problem

- Since strings are represented by single or double quotes, the compiler will treat "He said, " as the string. Hence, the above code will cause an error.

- To solve this issue, we use the escape character \ in Python.

```
# escape double quotes
example = "He said, \"What's there?\""

# escape single quotes
example = 'He said, "What\'s there?"'

print(example)
```

```
He said, "What's there?"
```

Continuation next meeting

String Interpolations

Python String Interpolation

- A process substituting values of variables into placeholders in a string.
- For instance, if you have a template for saying hello to a person like "Hello {Name of person}, nice to meet you!", you would like to replace the placeholder for name of person with an actual name.
- This process is called string interpolation.

F-strings

- Python 3.6 added new string interpolation method called literal string interpolation and introduced a new literal prefix f. This new way of formatting strings is powerful and easy to use. It provides access to embedded Python expressions inside string constants.

F-String Example

- In the example literal prefix f tells Python to restore the value of two string variable name and program inside braces {}. So, that when we print, we get the above output.
- This new string interpolation is powerful as we can embed arbitrary Python expressions, we can even do inline arithmetic with it.

```
name = 'World'
program = 'Python'
print(f'Hello {name}! This is {program}')
```

```
Hello World! This is Python
```

F-String Example

- In the given code, we did inline arithmetic which is only possible with this method.

```
a = 12
b = 3
print(f'12 multiply 3 is {a * b}.')
```

```
12 multiply 3 is 36.
```

%-formatting

- Strings in Python have a unique built-in operation that can be accessed with the % operator.
- Using % we can do simple string interpolation very easily.

%-formatting Example

- In this example we used two %s string format specifier and two strings Hello and World in parenthesis ().
- We got Hello World as output. %s string format specifier tell Python where to substitute the value.
- String formatting syntax changes slightly, if we want to make multiple substitutions in a single string, and as the % operator only takes one argument, we need to wrap the right-hand side in a tuple as shown in the example below.

```
print("%s %s" %('Hello','World',))
```

%-formatting Example

- In this example we used two string variable name and program.
- We wrapped both variable in parenthesis ().
- It’s also possible to refer to variable substitutions by name in our format string, if we pass a mapping to the % operator:

```
name = 'world'
program ='python'
print('Hello %s! This is %s.'%(name,program))
```

```
Hello world! This is python.
```

%-formatting Example

- It’s also possible to refer to variable substitutions by name in our format string, if we pass a mapping to the % operator:
- This makes our format strings easier to maintain and easier to modify in the future.
- We don’t have to worry about the order of the values that we’re passing with the order of the values that are referenced in the format string.

```
name = 'world'
program ='python'
print('Hello %(name)s! This is %(program)s.'%(name,program))
```

```
Hello world! This is python.
```

Str.format()

- In this string formatting we use format() function on a string object and braces {}, the string object in format() function is substituted in place of braces {}.
- We can use the format() function to do simple positional formatting, just like % formatting.

Str.format() Example

- In this example we used braces {} and format() function to pass name object .
- We got the value of name in place of braces {} in output.
- We can refer to our variable substitutions by name and use them in any order we want. This is quite a powerful feature as it allows for re-arranging the order of display without changing the arguments passed to the format function.

```
name = 'world'
print('Hello, {}'.format(name))
```

```
Hello,world
```

Str.format() Example

- We can refer to our variable substitutions by name and use them in any order we want.
- This is quite a powerful feature as it allows for re-arranging the order of display without changing the arguments passed to the format function.
- In this example we specified the variable substitutions place using the name of variable and pass the variable in format().

```
name = 'world'
program ='python'
print('Hello {name}!This is{program}.'.format(name=name,program=program))
```

```
Hello world!This is python.
```

Template Strings

- Template Strings is simpler and less powerful mechanism of string interpolation.
- We need to import Template class from Python's built-in string module to use it.
- In this example we import Template class from built-in string module and made a template which we used to pass two variable.

```
from string import Template
name = 'world'
program ='python'
new = Template('Hello $name! This is $program.')
print(new.substitute(name= name,program=program))
```

```
Hello world! This is python.
```

Key Things to Remember with String Interpolations

- %-format method is very old method for interpolation and is not recommended to use as it decrease the code readability.
- In str.format() method we pass the string object to the format() function for string interpolation.
- In template method we make a template by importing template class from built in string module.
- Literal String Interpolation method is powerful interpolation method which is easy to use and increase the code readability.
- F-string is powerful yet extremely easy to use and maintain.

Namespaces and Scopes

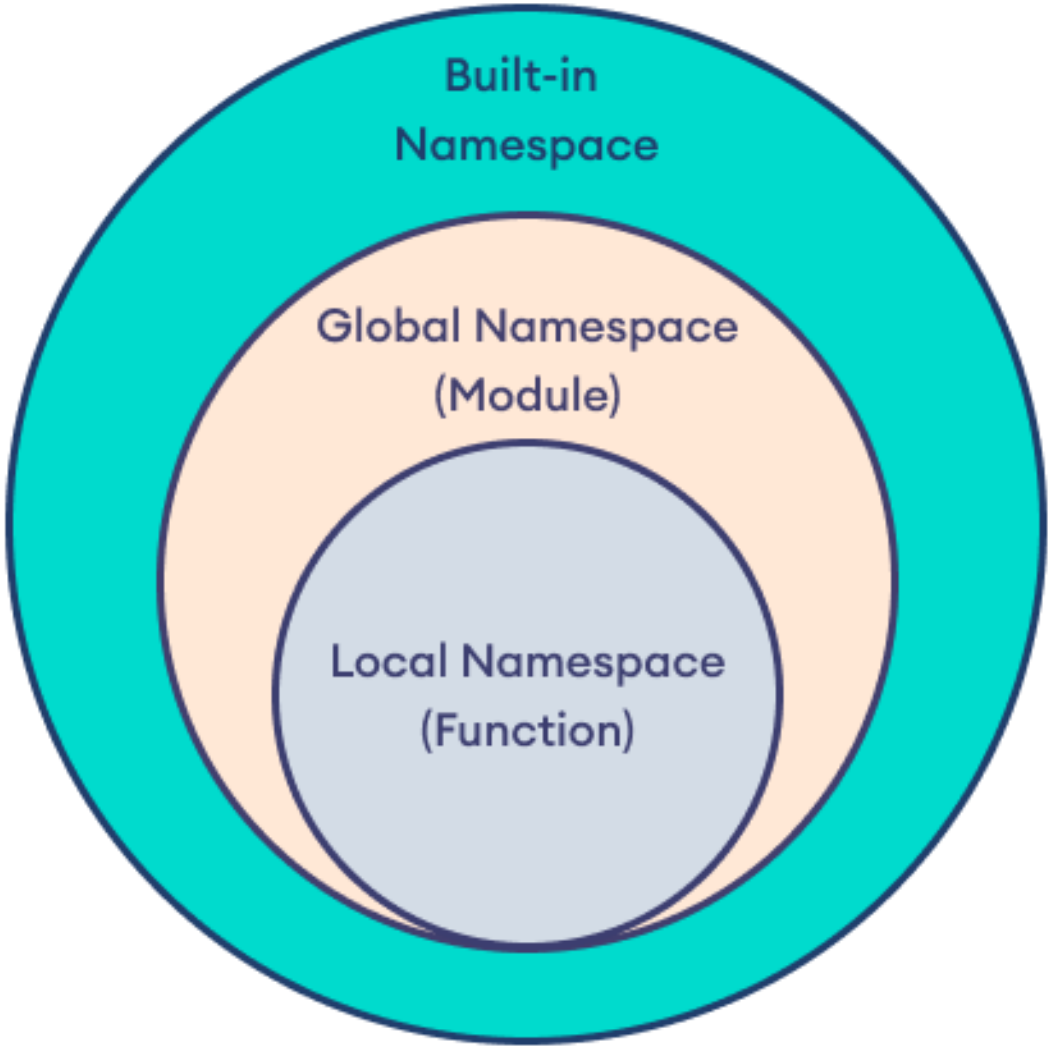
Namespaces and Scopes

- To simply put it, a namespace is a collection of names.
- In Python, we can imagine a namespace as a mapping of every name we have defined to corresponding objects.
- It is used to store the values of variables and other objects in the program, and to associate them with a specific name.
- This allows us to use the same name for different variables or objects in different parts of your code, without causing any conflicts or confusion.

Types of Python namespace

- A namespace containing all the built-in names is created when we start the Python interpreter and exists if the interpreter runs.
- 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 does 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. The following diagram may help to clarify this concept.

Python Namespace Bubble



Python Variable Scope

- Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.
- A scope is the portion of a program from where a namespace can be accessed directly without any prefix.

Three Nested Scopes

- Scope of the current function which has local names
- Scope of the module which has global names
- Outermost scope which has built-in names

- When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.

- If there is a function inside another function, a new scope is nested inside the local scope.

Example of different namespaces

In the example, there are three separate namespaces: the global namespace, the local namespace within the outer function, and the local namespace within the inner function.

```
# global_var is in the global namespace
global_var = 10

def outer_function():
    #  outer_var is in the local namespace
    outer_var = 20

    def inner_function():
        #  inner_var is in the nested local namespace
        inner_var = 30

        print(inner_var)

    print(outer_var)

    inner_function()

# print the value of the global variable
print(global_var)

# call the outer function and print local and nested local variables
outer_function()
```

```
10
20
30
```

Different namespaces in the example

- global_var - is in the global namespace with value 10
- outer_val - is in the local namespace of outer_function() with value 20
- inner_val - is in the nested local namespace of inner_function() with value 30

- When the code is executed, the global_var global variable is printed first, followed by the local variable: outer_var and inner_var when the outer and inner functions are called.

Global Keyword Example

Here, when the function is called, the global keyword is used to indicate that global_var is a global variable, and its value is modified to 30.

So, when the code is executed, global_var is printed first with a value of 10, then the function is called and the global variable is modified to 30 from the inside of the function.

And finally the modified value of global_var is printed again.

```
10
30
```

```
# define global variable
global_var = 10

def my_function():
    # define local variable
    local_var = 20

    # modify global variable value
    global global_var
    global_var = 30

# print global variable value
print(global_var)

# call the function and modify the global variable
my_function()

# print the modified value of the global variable
print(global_var)
```

Booleans

Boolean Values

- In programming you often need to know if an expression is True or False.
- You can evaluate any expression in Python, and get one of two answers, True or False.
- When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

```
True
False
False
```

Evaluating Values and Variables

- The bool() function allows you to evaluate any value, and give you True or False in return,

```
print(bool("Hello"))
print(bool(15))
```

```
True
True
```

```
x = "Hello"
y = 15

print(bool(x))
print(bool(y))
```

```
True
True
```

Most Values are True

- Almost any value is evaluated to True if it has some sort of content.
- Any string is True, except empty strings.
- Any number is True, except 0.
- Any list, tuple, set, and dictionary are True, except empty ones.

```
print(bool("abc"))
print(bool(123))
print(bool(["apple", "cherry", "banana"]))
```

```
True
True
True
```

Some Values are False

- In fact, there are not many values that evaluate to False, except empty values, such as (), [], {}, "", the number 0, and the value None. And of course the value False evaluates to False.

```
print(bool(False))
print(bool(None))
print(bool(0))
print(bool(""))
print(bool(()))
print(bool([]))
print(bool({}))
```

```
False
False
False
False
False
False
False
```

Truth Table

The logical AND operator indicates whether both operands are true.

If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0.

The logical OR (||) (logical disjunction) operator for a set of operands is true if and only if one or more of its operands is true.

The logical NOT (!) (logical complement, negation) operator takes truth to falsity and vice versa.

AND

a	b	a && b
T	T	T
T	F	F
F	T	F
F	F	F

OR

a	b	a || b
T	T	T
T	F	T
F	T	T
F	F	F

NOT

a	!a
T	F
F	T

Using Operators

Python Operators

- Operators are used to perform operations on variables and values.

- In the example below, we use the + operator to add together two values:

```
print(10 + 5)
```

```
15
```

Operator Groups

- Python divides the operators in the following groups:

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

Python Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python Comparison Operators

- Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python Logical Operators

- Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

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:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Python Membership Operators

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

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python Bitwise Operators

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

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

Operator Precedence

Operator precedence describes the order in which operations are performed.

Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first:

```
print((6 + 3) - (6 + 3))
```

```
0
```

```
"""
Parenthesis have the highest precedence, and need to be evaluated first.
The calculation above reads 9 - 9 = 0
"""
```

```
print(100 + 5 * 3)
```

```
115
```

```
"""
Multiplication has higher precedence than addition, and needs to be evaluated first.
The calculation above reads 100 + 15 = 115
"""
```

Bitwise Operators

OPERATOR	NAME	DESCRIPTION	SYNTAX
&	Bitwise AND	Result bit 1,if both operand bits are 1;otherwise results bit 0.	x & y
|	Bitwise OR	Result bit 1,if any of the operand bit is 1; otherwise results bit 0.	x | y
~	Bitwise NOT	inverts individual bits	~x
^	Bitwise XOR	Results bit 1,if any of the operand bit is 1 but not both, otherwise results bit 0.	x ^ y
>>	Bitwise right shift	The left operand's value is moved toward right by the number of bits specified by the right operand.	x>>
<<	Bitwise left shift	The left operand's value is moved toward left by the number of bits specified by the right operand.	x<<

Precedence Order

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x` `-x` `~x`	Unary plus, unary minus, and bitwise NOT
`*` `/` `//` `%`	Multiplication, division, floor division, and modulus
`+` `-`	Addition and subtraction
`<<` `>>`	Bitwise left and right shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==` `!=` `>` `>=` `<` `<=` `is` `is not` `in` `not in`	Comparisons, identity, and membership operators
`not`	Logical NOT
`and`	AND
`or`	OR

Same Precedence

If two operators have the same precedence, the expression is evaluated from left to right.

```
print(5 + 4 - 7 + 3)
```

```
5
```

Addition + and subtraction - has the same precedence, and therefor we evaluate the expression from left to right:

```
"""
Additions and subtractions have the same precedence, and we need to calculate from left to right.
The calculation above reads:
5 + 4 = 9
9 - 7 = 2
2 + 3 = 5
"""
```

Python Input and Output

Python Output

- In Python, we can simply use the print() function to print output. For example,
- Here, the print() function displays the string enclosed inside the single quotation.

```
1  print('Python is powerful')
```

```
Python is powerful
```

Syntax of print()

In the above code, the print() function is taking a single parameter.

However, the actual syntax of the print function accepts 5 parameters

```
print(object= separator= end= file= flush=)
```

Here,

object - value(s) to be printed

sep (optional) - allows us to separate multiple objects inside print().

end (optional) - allows us to add add specific values like new line "\n", tab "\t"

file (optional) - where the values are printed. It's default value is sys.stdout (screen)

flush (optional) - boolean specifying if the output is flushed or buffered. Default: False

Python Print Statement Example

- In this example, the print() statement only includes the object to be printed. Here, the value for end is not used. Hence, it takes the default value '\n’.
- So we get the output in two different lines.

```
1  print('Good Morning!')
2  print('It is rainy today')
```

```
Good Morning!
It is rainy today
```

Python Print Example with End Parameter

- Notice that we have included the end= ' ' after the end of the first print() statement.
- Hence, we get the output in a single line separated by space.

```
1  # print with end whitespace
2  print('Good Morning!', end= ' ')
3
4  print('It is rainy today')
```

```
Good Morning! It is rainy today
```

Python Print Example with Sep Parameter

- In this example, the print() statement includes multiple items separated by a comma.
- Notice that we have used the optional parameter sep= ". " inside the print() statement.
- Hence, the output includes items separated by . not comma.

```
print('New Year', 2023, 'See you soon!', sep= '. ')
```

```
New Year. 2023. See you soon!
```

Print Python Variables and Literals

- We can also use the print() function to print Python variables. For example,

```
1  number = -10.6
2
3  name = "BatStateU"
4
5  # print literals
6  print(5)
7
8  # print variables
9  print(number)
10 print(name)
```

```
5
-10.6
BatStateU
```

Print Concatenated Strings

- We can also join two strings together inside the print() statement. For example,
- Here,
- the + operator joins the individual strings
- the print() function prints the joined string

```
1  print('BatStateU' + ' CICS is ' + 'awesome.')
```

```
BatStateU CICS is awesome.
```

Output Formatting

- Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. For example,
- The curly braces {} are used as placeholders.
- We can specify the order in which they are printed by using numbers (tuple index).

```
1  x = 5
2  y = 10
3
4  print('The value of x is {} and y is {}'.format(x,y))
```

```
The value of x is 5 and y is 10
```

Python Input

- While programming, we might want to take the input from the user. In Python, we can use the input() function.
- Here, prompt is the string we wish to display on the screen. It is optional.

```
input(prompt)
```

Python User Input Example

- In this example, we used the input() function to take input from the user and stored the user input in the num variable.
- It is important to note that the entered value 5 is a string (default behavior), not a number. So, type(num) returns <class 'str'>.
- To convert user input into a number we can use int() or float() functions as:

```
1  # using input() to take user input
2  num = input('Enter a number: ')
3
4  print('You Entered:', num)
5
6  print('Data type of num:', type(num))
```

```
Enter a number: 5
You Entered: 5
Data type of num: <class 'str'>
```

```
num = int(input('Enter a number: '))
```