
Module 2

Python Programming Fundamentals

Interpreter Vs Compiler

Interpreters Vs Compilers

- Two kinds of program translator to convert from high-level languages into low-level languages:

- Interpreters

- Compilers.

-

Compiler

- Compiler translates a high level program to low level completely before the program starts running.

- High-level program is called **source code**

Translated program is called the **object code** or the **executable**.

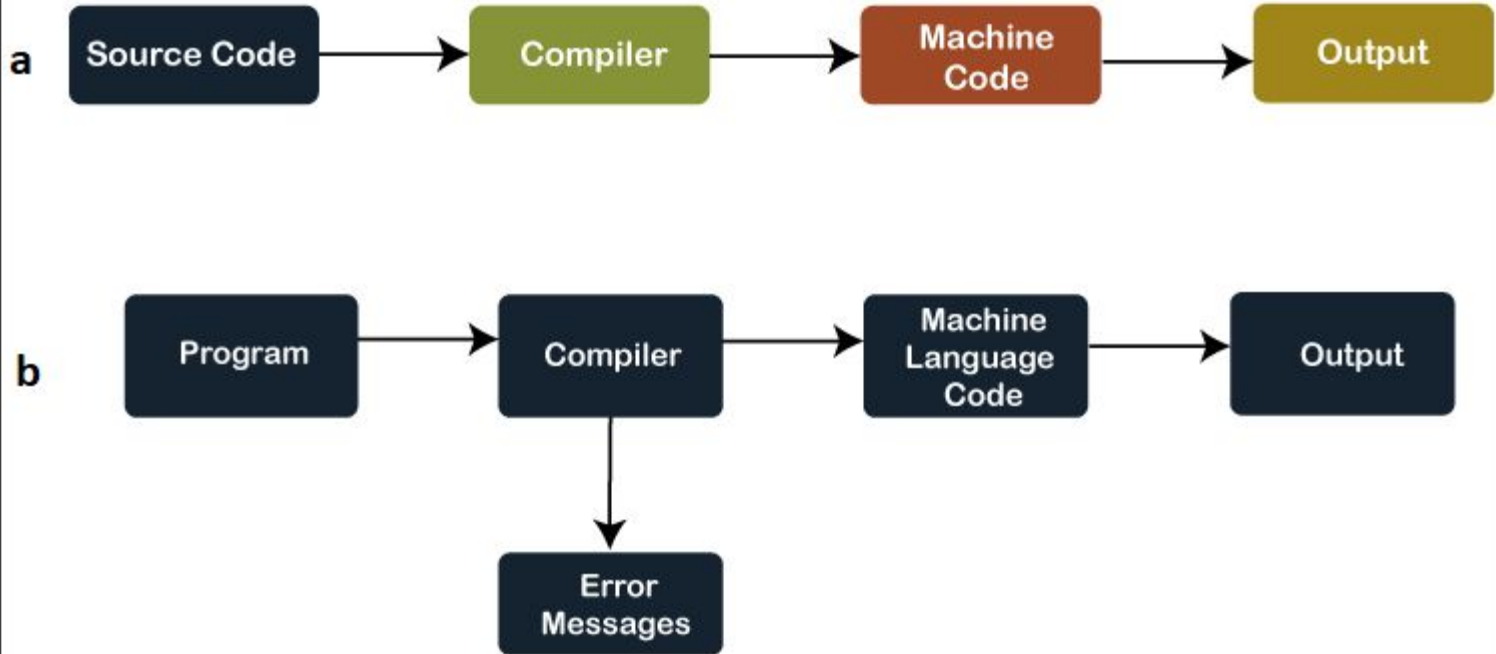
It does not fix the error

If no error is determined the complete source code is converted into object code(machine code)

Compiler

- The compiler links all the code files into a single runnable program, which is known as the exe file.
- Finally, it runs the program and generates output.

How Compiler Works



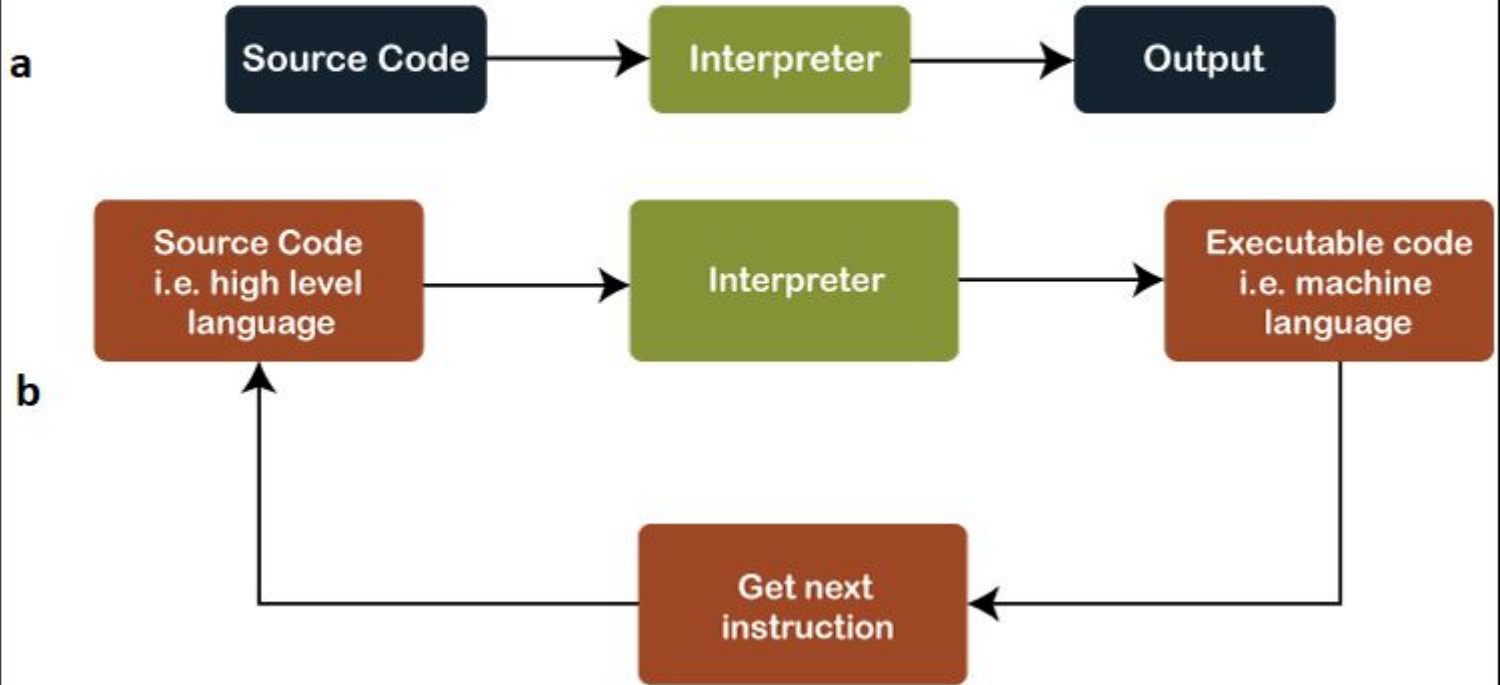
Interpreters

- ★ The source code programming statements are executed line-by-line during their execution.
- ★ If an error is found at any specific statement interpreter, it stops further execution until the error gets removed.
- ★ No linking of files happens, or no machine code will generate separately.

Interpreters

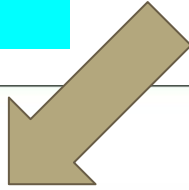


How Interpreter Works

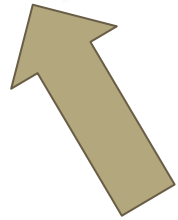


Python in idle

Displays month and year



```
Python 3.8.5 Shell
File Edit Shell Debug Options Window Help
Python 3.8.5 (default, Sep 3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```



Continuation lines

– Interactive and Script Mode

In Python programming, two distinct modes,
namely

1. Script Mode
2. Interactive Mode

Script mode:

- ★ Script Mode in Python refers to writing and executing Python scripts or programs.
- ★ It involves creating a standalone Python script file with a series of Python statements saved with the .py extension

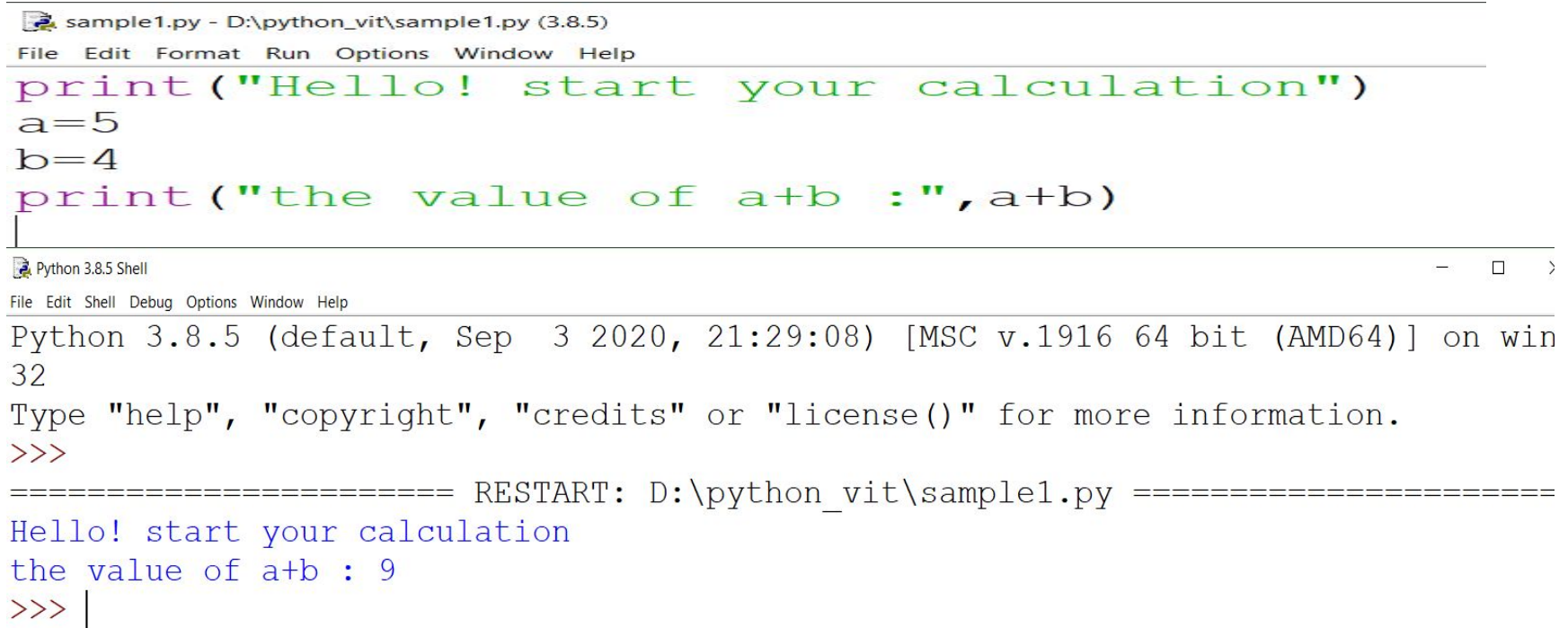
Using Script Mode

Creating a Script

Saving the Script

Executing the Script

Example



The image shows a screenshot of a Python script editor and its execution output. The top window is titled "sample1.py - D:\python_vit\sample1.py (3.8.5)" and contains the following code:

```
print("Hello! start your calculation")
a=5
b=4
print("the value of a+b :",a+b)
```

The bottom window is titled "Python 3.8.5 Shell" and shows the output of the script:

```
Python 3.8.5 (default, Sep 3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\python_vit\sample1.py =====
Hello! start your calculation
the value of a+b : 9
>>> |
```

Interactive mode

- ★ Interactive Mode in Python provides a dynamic environment for developers to interactively write and execute Python statements line by line.
- ★ It offers a way to experiment, test code snippets, and receive immediate feedback for each command entered.

Using Interactive Mode

1. Accessing Interactive Mode:

entering the python command in continuation lines without specifying a script filename

```
>>>
```

```
>>>
```

```
>>> |
```


Using Interactive Mode

2. Entering Statements:

```
>>>  
>>> print("Welcome")
```

Using Interactive Mode

3. Immediate Feedback:

```
>>> print("Welcome")  
Welcome  
>>> |
```

Identifiers

- ★ An identifier is a sequence of one or more characters used to name a given program element.
- ★ In Python, an identifier may contain letters (A-Z or a-z) and digits, but cannot begin with a digit.
- ★ Special underscore character can also be used
- ★ Example : line, salary, emp1, emp_salary

Rules for Identifier

Python is case sensitive

```
>>> NUM=5
>>> num=10
>>> print (NUM)
5
>>> print (num)
10
>>> |
```

Rules for Identifier

Identifiers may contain letters and digits, but cannot begin with a digit.

```
>>>  
>>> NUM12= 5 # numbers are allowed at the end  
>>> print(NUM12)
```

5

```
>>> 12NUM=5#identifiers should not start with numbers  
SyntaxError: invalid syntax  
>>> |
```

Rules for Identifier

The underscore character, `_`, is also allowed to aid in the readability of long identifier names.

```
>>> num_12=10 #usage of underscores in identifiers
>>> print(num_12)
10
>>> _num_12=10
>>> print(_num_12)
10
>>> _num=10
>>> print(_num)
10
>>> |
```

Rules for Identifier

Spaces are not allowed as part of an identifier

```
>>> num ab=10 #spaces are not allowed in identifiers
SyntaxError: invalid syntax
>>> num _=10
SyntaxError: invalid syntax
>>> num 12=10
SyntaxError: invalid syntax
```

Rules for Identifier

Quotes are not allowed in identifiers

```
'''  
>>> 'num_12'=10  
SyntaxError: cannot assign to literal  
>>> |
```


Rules for Identifier

```
val2@ = 35 # Identifier can't use special symbols
```

```
File "<ipython-input-14-cfbf60736601>", line 1
```

```
    val2@ = 35 # Identifier can't use special symbols
        ^
```

SyntaxError: invalid syntax

```
import = 125 # Keywords can't be used as identifiers
```

```
File "<ipython-input-15-f7061d4fc9ba>", line 1
```

```
    import = 125 # Keywords can't be used as identifiers
        ^
```

SyntaxError: invalid syntax

Keywords

- ★ Keywords are the reserved words in Python and can't be used as an identifier.
- ★ A keyword is an identifier that has pre-defined meaning in a programming language.

```
>>> and = 10
SyntaxError: invalid syntax
>>> |
```

keywords

The read-only attribute `kwlist` returns a list of all keywords reserved for the interpreter

```
In [3]: print(keyword.kwlist) # List all Python Keywords
```

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

```
In [4]: len(keyword.kwlist) # Python contains 35 keywords
```

```
Out[4]: 35
```

Comments in Python

```
In [18]: # Single line comment  
val1 = 10
```

```
In [19]: # Multiple  
# line  
# comment  
val1 = 10
```

```
In [20]: '''  
Multiple  
line  
comment  
'''  
val1 = 10
```

Comments in Python

```
In [21]: """  
Multiple  
line  
comment  
"""  
val1 = 10
```

Indentation

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

Indentation

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.

Input function()

```
>>> print("What is your  
name?")
```

What is your name?

In the print function, the output is printed and the stops execution

```
>>> input("What is your  
name?")
```

What is your
name?George

'George'

In the input function, the prompt is printed and waits till it receives data.

Input()

- Concatenation of print and input function
- >>> print("Hello "+input("What is your name?"))
- What is your name?George
- Hello George

Practice: Read two numbers from users and print the result

```
num1=int(input())  
num2=int(input())  
print(num1*num2)
```

Result: 5

5

25... Answer

Variables

- ★ A variable is a named location (identifier) 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.
- ★ Variables are assigned with values by use of the assignment operator

Variables



```
number = 10
```

Example 1: Declaring and assigning value to a variable

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

Output

```
apple.com
```

Example 2: Changing the value of a variable

```
website = "apple.com"  
print(website)  
  
# assigning a new value to website  
website = "programiz.com"  
  
print(website)
```

Output

```
apple.com  
programiz.com
```

Example 3: Assigning multiple values to multiple variables

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

```
print (a)
```

```
print (b)
```

```
print (c)
```

What is the expected output ??

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:

```
my_name  
current_salary
```

Rules and Naming Convention for Variables and Constants

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

```
PI  
G  
MASS  
SPEED_OF_LIGHT  
TEMP
```

5. Never use special symbols like !, @, #, \$, %, etc.

6. Don't start a variable name with a digit.


Variables

- ★ `input("What is your name?")`
- ★ Result:
 - What is your name?George
- ★ `print(input("What is your name?"))`
- ★ Result:
 - What is your name?George
 - George


- ★ `print(len(input("What is your name?")))`
- ★ Result:
 - What is your name?George
 - 6

How do you include a variable for the last slide??

```
name=input("what is your  
name?")  
length=len(name)  
print(length)  
Result:  
what is your name?George  
6
```



```
name=input("what is your  
name?\n")  
length=len(name)  
print(length)  
Result:  
what is your name?  
George  
6
```



Variables

- ★ Result
 - Input:
 - Hello
 - world
- ★ Output:
 - a:world
 - b: Hello

Code:

```
a=input()
b=input()
c=a
a=b
b=c
print("a:"+a)
print("b:",b)
```

Practice

- #1. Create a greeting for your program.
- #2. Ask the user for the place that they most love to settle.
- #3. Ask the user for the name of a city to which they have traveled.
- #4. Combine the name of their place and city and show them their Startup Company name .
- #5. Make sure the input cursor shows on a new line:

Answer

#1. Create a greeting for your program.

```
print("Welcome to the Startup Company name generator\n")
```

#2. Ask the user for the place that they most love to settle.

```
place=input("Which city you love to settle?\n")
```

#3. Ask the user for the name of a city to which they have traveled.

```
city=input("What is the name of a city to which you have traveled?\n")
```

#4. Combine the name of their place and city and show them their Startup

```
Company name .print("Your Startup Company name could be "+place+" "+city\n")
```

#5. Make sure the input cursor shows on a new line:

Data types

Data types supported by Python

1. Numeric Types
2. Sequence Types
3. Set Types
4. Mapping Types
5. Binary Types
6. None Type

1. Numeric Types

Integer (int): Represents whole numbers. Example: 42, -7, 0

Floating-point (float): Represents real numbers with a decimal point. Example: 3.14, -0.001, 2.0

Complex (complex): Represents complex numbers with real and imaginary parts. Example: $2 + 3j$, $-1.5 + 0.5j$

Numeric Types

Boolean (bool): Represents truth values True and False. Example: True, False

Fraction (fractions.Fraction): Represents rational numbers as fractions. Example: Fraction(1, 3), Fraction(2, 5)

Examples

```
# Numeric Types  
int_num = 42  
float_num = 3.14  
complex_num = 2 + 3j  
boolean_value = True
```

Fraction

```
from fractions import Fraction

# From two integers
fraction1 = Fraction(3, 4) # 3/4
print(fraction1)           # Output: 3/4

# From a string
fraction2 = Fraction('0.5') # 1/2
print(fraction2)           # Output: 1/2

# From a float
fraction3 = Fraction(0.75)  # 3/4
print(fraction3)           # Output: 3/4
```

Binary, Octal and Hex Literals

- 0b1, 0b10000, 0b11111111 # Binary literals:
base 2, digits 0-1
- 0o1, 0o20, 0o377
- # Octal literals: base 8, digits 0-7
- 0x01, 0x10, 0xFF # Hex literals: base 16,
digits 0-9/A-F
- (1, 16, 255)

Boolean values

Primitive datatype having one of two values: True or False

<code>print bool(True)</code>	True
<code>print (bool(False))</code>	False
<code>print (bool("text"))</code>	True
<code>print (bool(""))</code>	False
<code>print (bool(' '))</code>	True
<code>print (bool(0))</code>	False
<code>print (bool())</code>	False
<code>print (bool(3))</code>	True
<code>print (bool(None))</code>	False

2. Sequence Types (Note : all these are ordered)

String (str): Represents sequences of characters. Example: "hello", 'world'

List (list): Represents ordered, mutable sequences of elements. Example: [1, 2, 3], ['a', 'b', 'c']. Lists can store multiple values but string will store multiple characters.

Tuple (tuple): Represents ordered, immutable sequences of elements. Example: (1, 2, 3), ('a', 'b', 'c')

Range (range): Represents sequences of numbers, commonly used in loops. Example: range(10), range(1, 10, 2)

Strings

Strings in Python are identified as a set of characters represented in quotation marks.

Python allows for either pair of single or double quotes.

Strings are immutable

Strings

We cannot delete or remove characters from a string. But you can delete the entire string using 'del' keyword.

```
▶ name="Hello"  
  print(name)
```

⇒ Hello

```
▶ name="Hello"  
  del name  
  print(name)
```

⇒

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-2-f66690a6008b> in <cell line: 3>()  
      1 name="Hello"  
      2 del name  
>>> 3 print(name)
```

NameError: name 'name' is not defined

Strings

Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

★ plus (+) sign is the string concatenation operator

★ asterisk (*) is the repetition operator

String exercise

```
name="Welcome to the Python lab"
```

```
print(name)
```

```
print(name[0])
```

```
print(name[-1])
```

```
print(name[2:5])
```

```
print(name[2:])
```

```
print(name[:9])
```

```
print(name[:-1])
```

```
print(name * 2)
```

Hemavathy S/Python notes

```
print(name + "Text")
```

Welcome to the Python lab

W

b

lco

lcome to the Python lab

Welcome t

Welcome to the Python la

Welcome to the Python labWelcome
to the Python lab


Welcome to the Python labText

```
str="Hello World!"  
print (str)  
print (str[0])  
print (str[3:6])  
print (str [3: ])  
print(str[-1])  
print (str[ :-1])  
print str * 4  
print str+ "TEST"
```


Lists

Lists store multiple values separated with a comma.


Lists can have any data types as values in it.




```
['Hello', 1, 2.56, 3j+4]
```



```
['Hello', 1, 2.56, (4+3j)]
```



```
['Hello', 1, 2.56, 3j+4, [18, "Call", 2+3j], 5.2]
```



```
['Hello', 1, 2.56, (4+3j), [18, 'Call', (2+3j)], 5.2]
```

Tuples

Tuple is similar to List except that the objects in tuple are immutable which means we cannot change the elements of a tuple once assigned.

When we do not want to change the data over time, tuple is a preferred data type

Tuple

```
In [533]: tup1 = ()      # Empty tuple
```

```
In [534]: tup2 = (10,30,60)      # tuple of integers numbers
```

```
In [535]: tup3 = (10.77,30.66,60.89)      # tuple of float numbers
```

```
In [536]: tup4 = ('one','two' , "three")  # tuple of strings
```

```
In [537]: tup5 = ('Asif', 25 ,(50, 100),(150, 90))  # Nested tuples
```

```
In [538]: tup6 = (100, 'Asif', 17.765)  # Tuple of mixed data types
```

```
# Creating Tuples
```

```
tup1 = (1, 2, 3)
```

```
tup2 = (1, "hello", 3.14)
```

```
tup3 = (1, (2, 3), [4, 5])
```

```
tup4 = (1,) # Single element tuple
```

```
tup5 = () # Empty tuple
```

```
print(tup1) # Output: (1, 2, 3)
```

```
print(tup2) # Output: (1, 'hello', 3.14)
```

```
print(tup3) # Output: (1, (2, 3), [4, 5])
```

```
print(tup4) # Output: (1,)
```

```
print(tup5) # Output: ()
```

```
# Accessing Tuple Elements
```

```
tup = (1, 2, 3, 4, 5)
```

```
print(tup[0]) # Output: 1
```

```
print(tup[3]) # Output: 4
```

```
print(tup[1:3]) # Output: (2, 3)
```

```
print(tup[:2]) # Output: (1, 2)
```

```
print(tup[3:]) # Output: (4, 5)
```

Range

The range function in Python is used to generate a sequence of numbers.

It is commonly used in loops to iterate over a sequence of numbers.

The range function can be used in several ways depending on the number of arguments provided.

Syntax

1. `range(stop)`
2. `range(start, stop)`
3. `range(start, stop, step)`

```
# Example: range(stop)
for i in range(5):
    print(i) # Output: 0, 1, 2, 3, 4
```

```
# Example: range(start, stop)
for i in range(2, 6):
    print(i) # Output: 2, 3, 4, 5
```

```
# Example: range(start, stop, step)
for i in range(1, 10, 2):
    print(i) # Output: 1, 3, 5, 7, 9

for i in range(10, 1, -2):
    print(i) # Output: 10, 8, 6, 4, 2
```


3. Set

- 1) Unordered & Unindexed collection of items.
 - 2) Set elements are unique. Duplicate elements are not allowed. Application : in statistical analysis
 - 3) A set is a collection which is unordered, unchangeable*, and unindexed.
 - 4) FrozenSet elements are immutable (cannot be changed)
- * Note: Set items are unchangeable, but you can remove items and add new items.

3. Set

- 1) Unordered & Unindexed collection of items.
- 2) Set elements are unique. Duplicate elements are not allowed. Application : in statistical analysis
- 3) Set elements are immutable (cannot be changed)

Set Creation

```
In [634]: myset = {1,2,3,4,5} # Set of numbers  
myset
```

```
Out[634]: {1, 2, 3, 4, 5}
```

```
In [635]: len(myset) #Length of the set
```

```
Out[635]: 5
```

Set

```
In [636]: my_set = {1,1,2,2,3,4,5,5}
my_set                                     # Duplicate elements are not allowed.
```

```
Out[636]: {1, 2, 3, 4, 5}
```

```
In [637]: myset1 = {1.79,2.08,3.99,4.56,5.45} # Set of float numbers
myset1
```

```
Out[637]: {1.79, 2.08, 3.99, 4.56, 5.45}
```

```
In [638]: myset2 = {'Asif' , 'John' , 'Tyrion'} # Set of Strings
myset2
```

```
Out[638]: {'Asif', 'John', 'Tyrion'}
```

Frozen set

```
# Creating a frozenset  
frozenset_a = frozenset([1, 2, 3, 4])  
print(frozenset_a) # Output: frozenset({1, 2, 3, 4})
```

Try this

```
thisset = {"apple", "banana",  
"cherry", True, 1, 2}
```

```
print(thisset)
```

```
thisset = {"apple", "banana",  
"cherry", False, True, 0}
```

```
print(thisset)
```

4. Mapping data type

Dictionary (dict): Represents collections of key-value pairs.

Example: {'a': 1, 'b': 2}, {1: 'one', 2: 'two'}

Dictionary

Python's dictionaries consist of key-value pairs

A dictionary key can be almost any Python type, **but are usually numbers or strings.**

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

The values of the dictionary can be changed.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

Note: As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Dictionaries cannot have two items with the same key:.

Practice

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
print(thisdict)  
print(thisdict["brand"])
```

```
{'brand': 'Ford', 'model':  
'Mustang', 'year': 1964}
```

Ford

Duplicate values will overwrite existing values:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}
```

```
print(thisdict)
```

```
{'brand': 'Ford', 'model':  
'Mustang', 'year': 2020}
```

Note: The values in dictionary items can be of any data type:

```
# Creating a dictionary using curly braces
```

```
dict_a = {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```
print(dict_a) # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```
# Creating a dictionary using the dict() constructor
```

```
dict_b = dict(name='Bob', age=30, city='Los Angeles')
```

```
print(dict_b) # Output: {'name': 'Bob', 'age': 30, 'city': 'Los Angeles'}
```

```
# Modifying values
```

```
dict_a['age'] = 26
```

```
print(dict_a) # Output: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

```
dict = {}  
dict['one'] = "This is one"  
dict[2]      = "This is two"  
  
tinydict = {'name': 'john', 'code':6734, 'dept': 'sales'}  
  
print dict['one']      # Prints value for 'one' key  
print dict[2]          # Prints value for 2 key  
print tinydict          # Prints complete dictionary  
print tinydict.keys()   # Prints all the keys  
print tinydict.values() # Prints all the values
```

This produce the following result –

```
This is one  
This is two  
{'dept': 'sales', 'code': 6734, 'name': 'john'}  
['dept', 'code', 'name']  
['sales', 6734, 'john']
```

5. None data type

Special data type - None

Basically, the data type means non existent, not known or empty

Can be used to check for emptiness

None data type

```
def check_return():  
    pass  
print(check_return())
```

Result: None

List, tuples, dictionary

LIST	TUPLE	DICTIONARY	SET
[]	()	{ }	{ }
Mutable	Immutable	Mutable	Mutable(can add or remove item)
Duplicates possible	Duplicates possible	No Duplicates	No Duplicates
Ordered	Ordered	Ordered	Unordered

```
In [3]: a = ['abcd', 786, 2.35, 'john', 70.2]
....: b = ('abcd', 786, 2.35, 'john', 70.2)
....: c = {'name': 'abcd', 'age': 65, 'weight': 60}
....: print(type(a))
....: print(type(b))
....: print(type(c))
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

Operators

Operators are special symbols in Python which are used to perform operations on variables/values.

- ★ Arithmetic operators
- ★ Ternary operator
- ★ Relational operators
- ★ Assignment operators
- ★ Logical operators
- ★ Bitwise operators
- ★ Special operators

Arithmetic operators

Command	Name	Example	Output
Unary +	positive	+3	3
Unary -	negative	-4	-4
+	Addition	4 + 5	9
-	Subtraction	8 - 5	3
*	Multiplication	4 * 5	20
/	Division	19 / 3	6.3333
//	Floor Division	19//3	6
%	Remainder (modulo)	19 % 3	1
**	Exponent	2 ** 4	16

Arithmetic operators

Unary +

```
a=5  
print(+a)
```

5

```
a=0  
print(+a)
```

0

Unary -

```
: a=5  
print(-a)
```

-5

```
: a=0  
print(-a)
```

0

Arithmetic operators

Addition operator(+)

```
3+6
```

9

```
a=8  
a+4
```

12

```
a,b=2,5  
a+b
```

7

Subtraction operator(-)

```
15-5
```

10

```
a=10  
a-4
```

6

```
a,b=12,5  
a-b
```

7

Arithmetic operators

Multiplication operator(*)

```
4*5
```

20

```
a=10
```

```
a*4
```

40

```
a,b=12,5
```

```
a*b
```

60

Division operator(/)

```
4/2
```

2.0

```
7/2.5
```

2.8

```
100/32
```

3.125

```
-7/4
```

-1.75

```
-7/-2
```

3.5

```
7/-2
```

-3.5

Arithmetic operators

Floor division operator(//)

```
7//2
```

3

```
4//2
```

2

```
6.5//2
```

3.0

```
6//2.1
```

2.0

```
-5//2
```

-3

```
5//-2
```

-3

```
-5//-2
```

2

Arithmetic operators

Modules operator(%)

$19\%2$

1

$7.2\%3$

1.2000000000000002

$6\%2.5$

1.0

$-7\%2$

1

$-7\%-2$

-1

$7\%-2$

-1

Exponentiation operator(**)

$2^{**}3$

8

$4^{**}2$

16

$x=6$

$x^{**}2$

36

Ternary operators

These are operators that test a condition and, based on that, evaluate a value.

```
a,b=2,3
```

```
if a>b:
```

```
    print("a")
```

```
else:
```

```
    print("b")
```

Relational operator

Operator	Syntax	Output
==	(a == b)	If the values of a and b are equal, then the condition becomes true.
!=	(a != b)	If values of a and b are not equal, then the condition becomes true.
>	(a > b)	If the value of a is greater than the value of b, then the condition becomes true.
<	(a < b)	If the value of a is less than the value of b, then the condition becomes true.
>=	(a >= b)	If the value of a is greater than or equal to the value of b, then the condition becomes true.
<=	(a <= b)	If the value of b is less than or equal to the value of b, then the condition becomes true.

Less than operator (<)

```
10 < 20
```

True

```
30 < 20
```

False

```
x = 10
```

```
y = 20
```

```
x < y
```

True

```
x = 10
```

```
y = 20
```

```
y < x
```

False

Assignment operator

Operator	Description	Syntax	Output
=	Equal to	c = a + b	assigns a value of a + b into c
+=	Add AND	c += a	is equivalent to c = c + a
-=	Subtract AND	c -= a	is equivalent to c = c - a
*=	Multiply AND	c *= a	is equivalent to c = c * a
/=	Divide AND	c /= a	is equivalent to c = c / a;
%=	Modulus AND	c %= a	is equivalent to c = c % a
**=	Exponent AND	c **= a	is equivalent to c = c ** a
//=	Floor Division	c //= a	is equivalent to c = c // a

Logical operators

Logical operators are used on conditional statements (either True or False). They perform Logical AND, Logical OR and Logical NOT operations.

Logical operators

Operator	Description	Syntax	Output
and	Logical AND: True if both the operands are true	x and y	(2>1)and(5>3) True (1>4)and(5>3) False
or	Logical OR: True if either of the operands is true	x or y	(1>4)or(5>3) True (5<2)or(3<1) False
not	Logical NOT: True if operand is false	not x	

a	b	a and b
True	True	True
True	False	False
False	False	False
False	True	False

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

a	not a
True	False
False	True

Precedence of logical operators

NOT	Decreasing order 
AND	
OR	

Bitwise operator

Refers to the operators working on a bit

Bitwise operator

Operator	Description	Syntax	Output
&	Binary AND	<u>a&b</u>	copies a bit to the result if it exists in both operands
	Binary OR	a b	copies a bit if it exists in either operand.
^	Binary XOR	<u>a^b</u>	copies the bit if it is set in one operand but not both.
~	Binary One's Complement	~a	Unary operation of flipping bits
<<	Binary Left Shift	a<<b	The left operands value is moved left by the number of bits specified by the right operand.

```
In [10]:  
....: a = 10  
....: b = 4  
....: print(a & b)  
....: print(a|b)  
....: print(~a)  
....: print(a^b)  
....: print(a>>2)  
....: print(b<<4)
```

0

14

-11


14

2

64

Operator	Operation	Precedence
()	parentheses	0
**	exponentiation	1
*	multiplication	2
/	division	2
//	<u>int</u> division	2
%	remainder	2
+	addition	3
-	subtraction	3

Precedence of Operators

Arithmetic	Decreasing order 
Relational	
Logical	

Special operators

Special operators provide additional functionality to perform specific tasks or comparisons in a more specialized manner.

Identity operator

Identity operator

The identity operators in Python are used to **compare the identity of two objects**, which means they check whether two objects **refer to the same memory location**.

★ is

★ is not

“is” identity operator

The "is" operator returns **True** if two objects have the same **identity**, indicating that they are the same object.

It **compares the memory addresses** of the objects rather than their values.

If the objects have different memory addresses, even if their values are the same, the "is" operator will return **False**.

Syntax: **object1 is object2**

“is not” operator

The "is not" operator returns True if two objects have different identities, indicating that they are not the same object.

It is the negation of the "is" operator.

Syntax: **object1** is not **object2**

Example

x1 = 5

y1 = 5

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

print(x1 is not y1)

print(x2 is y2)

print(x3 is y3)

False

True

False

Membership Operators

Membership operators are used to check the membership of a value in a sequence or collection, such as strings, lists, tuples, or sets. The membership operators available in Python are "in" and "not in".

"in" operator:

The "in" operator returns True if a value is found in the given sequence or collection.

It checks whether the value is present within the sequence and returns True if it is, and False otherwise.

Membership Operators

"not in" operator:

The "not in" operator returns True if a value is not found in the given sequence or collection.

It checks whether the value is absent within the sequence and returns True if it is, and False otherwise.

```
message = 'Hello world'  
dict1 = {1:'a', 2:'b'}
```

```
# check if 'H' is present in message string  
print('H' in message) # prints True
```

```
# check if 'hello' is present in message string  
print('hello' not in message) # prints True
```

```
# check if '1' key is present in dict1  
print(1 in dict1) # prints True
```

```
# check if 'a' key is present in dict1  
print('a' in dict1) # prints False
```

```
In [5]: x = 'VIT Chennai'
y = {3:'a',4:'b'}
print('VIT' in x)
print('VIT' not in x)
print('Chennai' not in x)
print(3 in y)
print('b' in y)
```

True

False

False

True

False

Built-in functions

COMMON BUILT-IN FUNCTIONS

Basic Functions

`print()`: Prints the specified message to the screen.

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

`type()`: Returns the type of an object.

```
print(type(5)) # Output: <class 'int'>\
```


COMMON BUILT-IN FUNCTIONS

`id()`: Returns the unique identifier (memory address) of an object.

```
a = 10 print(id(a))
```

COMMON BUILT-IN FUNCTIONS

Numeric Functions

abs(): Returns the absolute value of a number.

`print(abs(-5))` # Output: 5

round(): Rounds a floating-point number to the nearest integer.

`print(round(4.6))` # Output: 5

max(): Returns the largest item in an iterable or the largest of two or more arguments.

`print(max(1, 2, 3))` # Output: 3

COMMON BUILT-IN FUNCTIONS

min(): Returns the smallest item in an iterable or the smallest of two or more arguments.

`print(min(1, 2, 3))` # Output: 1

sum(): Sums the items of an iterable.

`print(sum([1, 2, 3]))` # Output: 6

COMMON BUILT-IN FUNCTIONS

complex(): Returns a complex number by specifying a real and an imaginary number.

`complex(3,5)`

`(3+5j)`

pow(): Returns value of X to the power of y (x^y)

`pow(3,4)` # Output: 81

COMMON BUILT-IN FUNCTIONS

Sequence Functions

len(): Returns the number of items in an object.

`print(len("hello"))` # Output: 5

sorted(): Returns a sorted list of the specified iterable.

`print(sorted([3, 1, 2]))` # Output: [1, 2, 3]

reversed(): Returns a reverse iterator.

`for i in reversed([1, 2, 3]):`

`print(i)` # Output: 3 2 1

COMMON BUILT-IN FUNCTIONS

Sequence Functions

chr(): Returns a character for an ascii value.

`print(chr(65))` # Output:A

ord(): Returns an integer that represents a Unicode point for a given Unicode character. This is complementary to `chr()`.

`print(ord('A'))`# Output: 65

Python all() Function

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

Python all() Function Example

```
# all values true
```

```
k = [1, 3, 4, 6]
```

```
print(all(k))
```

Python bin() Function

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

Python bin() Function Example

```
x = 10  
y = bin(x)  
print (y)
```


Built-in Functions

A

[abs\(\)](#)
[aiter\(\)](#)
[all\(\)](#)
[anext\(\)](#)
[any\(\)](#)
[ascii\(\)](#)

B

[bin\(\)](#)
[bool\(\)](#)
[breakpoint\(\)](#)
[bytearray\(\)](#)
[bytes\(\)](#)

C

[callable\(\)](#)
[chr\(\)](#)
[classmethod\(\)](#)
[compile\(\)](#)
[complex\(\)](#)

D

[delattr\(\)](#)
[dict\(\)](#)
[dir\(\)](#)
[divmod\(\)](#)

E

[enumerate\(\)](#)
[eval\(\)](#)
[exec\(\)](#)

F

[filter\(\)](#)
[float\(\)](#)
[format\(\)](#)
[frozenset\(\)](#)

G

[getattr\(\)](#)
[globals\(\)](#)

H

[hasattr\(\)](#)
[hash\(\)](#)
[help\(\)](#)
[hex\(\)](#)

I

[id\(\)](#)
[input\(\)](#)
[int\(\)](#)
[isinstance\(\)](#)
[issubclass\(\)](#)
[iter\(\)](#)

L

[len\(\)](#)
[list\(\)](#)
[locals\(\)](#)

M

[map\(\)](#)
[max\(\)](#)
[memoryview\(\)](#)
[min\(\)](#)

N

[next\(\)](#)

O

[object\(\)](#)
[oct\(\)](#)
[open\(\)](#)
[ord\(\)](#)

P

[pow\(\)](#)
[print\(\)](#)
[property\(\)](#)

R

[range\(\)](#)
[repr\(\)](#)
[reversed\(\)](#)
[round\(\)](#)

S

[set\(\)](#)
[setattr\(\)](#)
[slice\(\)](#)
[sorted\(\)](#)
[staticmethod\(\)](#)
[str\(\)](#)
[sum\(\)](#)
[super\(\)](#)

T

[tuple\(\)](#)
[type\(\)](#)

V

[vars\(\)](#)

Z

[zip\(\)](#)

[__import__\(\)](#)

TYPE Conversion

Python int()

Python float()

Python str()

Python bool()

Python type() and

Python isinstance() # Checks if an object is an instance of a particular type or class.

Initialize variables

a = "123" # a is a string

b = 45.67 # b is a float

c = True # c is a boolean

Convert string to integer

int_a = int(a)

print("int(a):", int_a, " | Type:", type(int_a))

Convert float to integer

int_b = int(b)

print("int(b):", int_b, " | Type:", type(int_b))

Convert integer to float

float_a = float(int_a)

print("float(int_a):", float_a, " | Type:", type(float_a))

Convert integer to string

str_b = str(int_b)

print("str(int_b):", str_b, " | Type:", type(str_b))

Convert integer to boolean

bool_a = bool(int_a)

Import Module in Python

```
import math
```

```
print(math.pi)
```

```
print(math.sqrt(4))
```

Example

Exercise 1: Simple Interest Calculation

Problem: Write a Python program to calculate the simple interest given the principal amount, rate of interest, and time period.

Formula: Simple Interest $SI = P \times R \times T / 100$

Prompt the user to enter the principal amount, rate of interest, and time period

```
principal = float(input("Enter the principal amount: "))
```

```
rate = float(input("Enter the rate of interest: "))
```

```
time = float(input("Enter the time period in years: "))
```

Calculate simple interest

```
simple_interest = (principal * rate * time) / 100
```

Print the result

```
print("The Simple Interest is:", simple_interest)
```

Example 2:

Write a Python program to calculate the compound interest given the principal amount, rate of interest, time period, and number of times the interest is compounded per year.

$$CI = A - P$$

And

$$CI = P\left(1 + \frac{r}{n}\right)^{nt} - P$$

- A = amount
- P = principal
- r = rate of interest
- n = number of times interest is compounded per year
- t = time (in years)


```
# Prompt the user to enter the principal amount, rate of  
interest, time period, and number of times interest is  
compounded per year
```

```
principal = float(input("Enter the principal amount: "))  
rate = float(input("Enter the annual rate of interest: "))  
time = float(input("Enter the time period in years: "))  
n = int(input("Enter the number of times interest is  
compounded per year: "))
```

```
# Calculate the compound interest
```

```
amount = principal * (1 + (rate / (n * 100))) ** (n * time)  
compound_interest = amount - principal
```

```
# Print the result
```

```
print("The Compound Interest is:", compound_interest)
```