**1. INTRODUCTION TO PYTHON**

**1.1. What is PYTHON?**

* **Python** is a widely used general-purpose high level popular programming language. It was **created in 1991** by **Guido van Rossum** and developed by Python Software Foundation.
* **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
* **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
* **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
* **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

**1.2. Feature of Python:**

* **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
* **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
* **Easy-to-maintain**− Python's source code is fairly easy-to-maintain.
* **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
* **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
* **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
* **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add or customize their tools to be more efficient.
* **Databases** − Python provides interfaces to all major commercial databases.
* **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
* **Scalable**− Python provides a better structure and support for large programs than shell scripting.

**1.3. Python is used for:**

* Python can be used on a server to create web applications.
* Python can be used alongside software to create workflows.
* Python can connect to database systems. It can also read and modify files.
* Python can be used to handle big data and perform complex mathematics.
* Python can be used for rapid prototyping, or for production-ready software development.

**1.4. Why Python?**

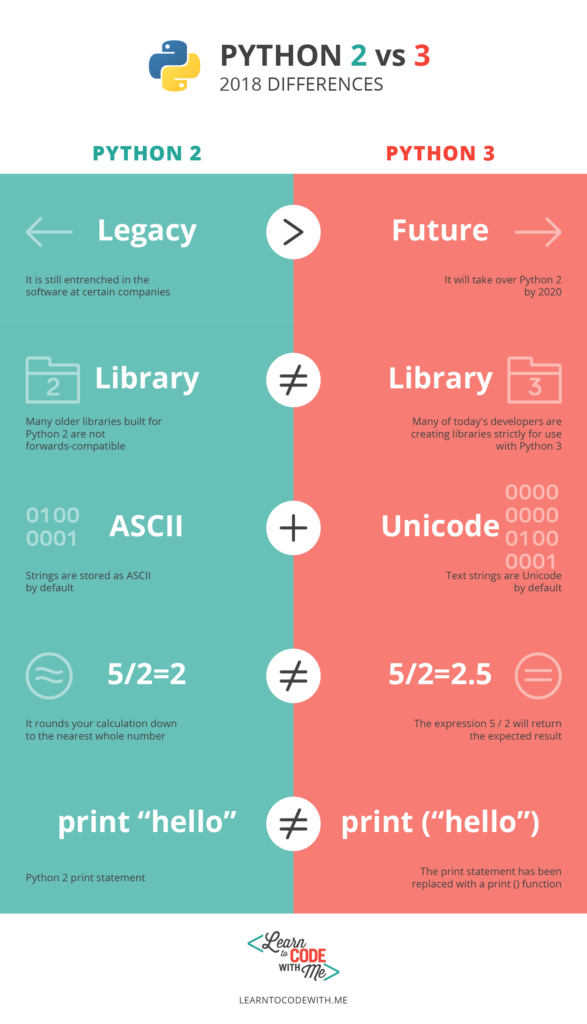
* Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
* Python has a simple syntax similar to the English language.
* Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
* Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
* Python can be treated in a procedural way, an object-orientated way or a functional way.

**1.5. A LITTLE HISTORY OF PYTHON 2 VS 3:**

There are two major Python versions- **Python 2 and Python 3**. Both are quite different

* Python 2.0 was first released in 2000. Its latest version, 2.7, was released in 2010.
* Python 3.0 was released in 2008. Its newest version, 3.6, was released in 2016, and [**version 3.**](https://www.python.org/downloads/release/python-370b5/)**8**is currently in development.
* Although Python 2.7 is still widely used, Python 3 adoption is growing quickly. In 2016, 71.9% of projects used Python 2.7, but [**by 2017, it had fallen to 63.7%**](https://semaphoreci.com/blog/2017/10/18/python-versions-used-in-commercial-projects-in-2017.html). This signals that the programming community is turning to Python 3–albeit gradually–when developing real-world applications.
* Notably, on January 1, 2018, Python 2.7 will “retire” and no longer be maintained. ([The clock is literally ticking!](https://pythonclock.org/)).

5 Main Differences between Python 2 and Python 3 2018:



**1.6. Python Syntax compared to other programming languages:**

* Python was designed to for readability and has some similarities to the English language with influence from mathematics.
* Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
* Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

**1.7. Beginning with Python programming:**

**1) Finding an Interpreter:**

Before we start Python programming, we need to have an interpreter to interpret and run our programs.

***Windows*:** There are many interpreters available freely to run Python scripts like IDLE ( Integrated Development Environment ) which is installed when you install the python software from [**http://python.org/**](http://python.org/)

***Linux*:** For Linux, Python comes bundled with the linux.

**2) Writing first program:**

Following is first program in Python

#script Begins

print("This is our First Python Program")

#Script Ends

**OUTPUT**: This is our First Python Program

Let us analyze the script line by line.

***Line 1 : [*# Script Begins]** In Python, comments begin with #. So this statement is for readability of code and ignored by Python interpreter.

***Line 2 : [print(“***This is our First Python Program***”)]***In a Python script to print something on the console print() function is used – it simply prints out a line ( and also includes a newline unlike in C ). One difference between Python 2 and Python 3 is the print statement. In Python 2, the “print” statement is not a function, and therefore can be invoked without a parenthesis. However, in Python 3, it is a function, and must be invoked with parentheses.

***Line 3 : [*# Script Ends]**This is just another comment like Line 1.

## Python code in interactive mode and script mode:

* **Interactive mode** is used when the user wants to run one single line or one block of code. It runs very quickly and gives the output instantly.
* **Script Mode** is used when the user is working with more than one single code or a block of code.

**2. PYTHON KEYWORDS:**

* Keywords are special words which are reserved and have a specific meaning.
* Python has a set of keywords that cannot be used as constants or variables in programs or any other identifier names.
* **All keywords in Python are case sensitive**. So, you must be careful while using them in your code.

help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

False def if raise

None del import return

True elif in try

and else is while

as except lambda with

assert finally nonlocal yield

break for not

class from or

continue global pass

>>> import keyword

>>> keyword.kwlist

['False', 'None', 'True', 'and', 'as', 'assert', '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']

**# Python code to demonstrate True, False, None, and, or , not**

# showing that None is not equal to 0   
# prints False as its false.   
print (None == 0)       Output : False

# showing objective of None   
# two None value equated to None   
# here x and y both are null   
# hence true   
x = None  
y = None  
print (x == y)              Output : True

# **Python code to demonstrate del**

# initializing list   
a = [1, 2, 3]

# printing list before deleting any value   
print ("The list before deleting any value")   
print (a)

# using del to delete 2nd element of list   
del a[1]

# printing list after deleting 2nd element   
print ("The list after deleting 2nd element")   
print (a)

Output:

The list before deleting any value

[1, 2, 3]

The list after deleting 2nd element

[1, 3]

**# demonstrating use of assert**

**assert** : This function is used for **debugging purposes**. Usually used to check the correctness of code. If a statement evaluated to true, nothing happens, but when it is false, “**AssertionError**” is raised. One can also **print a message with the error, separated by a comma**.  
# prints AssertionError   
assert 5 < 3, "5 is not smaller than 3"

Output:

Traceback (most recent call last):

File "9e957ae60b718765ec2376b8ab4225ab.py", line 19, in

assert 5<3, "5 is not smaller than 3"

AssertionError: 5 is not smaller than 3

# **demonstrating non local**

**non-local**: This keyword works similar to the global, but rather than global, this keyword declares a variable to point to variable of outside enclosing function, in case of nested functions.

# inner loop changing the value of outer a

# prints 10

print ("Value of a using nonlocal is : ",end="")

def outer():

    a = 5

    def inner():

        nonlocal a

        a = 10

    inner()

    print (a)

outer()

Output:

Value of a using nonlocal is : 10

**# demonstrating without non local**

# inner loop not changing the value of outer a

# prints 5

print ("Value of a without using nonlocal is : ",end="")

def outer():

    a = 5

    def inner():

        a = 10

    inner()

    print (a)

outer()

Output:

Value of a without using nonlocal is : 5

**3. PYTHON IDENTIFIERS**

* A Python identifier is a **name** used to identify a variable, function, class, module or other object.
* Python **does not allow** punctuation characters such as **@, $, and %** within identifiers. Python is a case sensitive programming language. Thus, **Manpower**and **manpower** are two different identifiers in Python.
* Here are naming conventions for Python identifiers –

1. **Class names** start with an uppercase letter. All **other identifiers** start with a lowercase letter.

2. Starting an identifier with a single leading underscore indicates that the **identifier is private.**

3. Starting an identifier with two leading underscores indicates a **strongly private identifier.**

* If the identifier also ends with two trailing underscores, the identifier is a **language-defined special name**.
* Avoid using names with only one character. Instead, make meaningful names.

**For example –** While i = 1 is valid, but writing iter = 1 or index = 1 would make more sense.

* You can use underscore to combine multiple words to form a sensible name.

**For example –** count\_no\_of\_letters

**3.1 Rules to be followed while creating Identifiers:**

**1.** To form an identifier, use a **sequence of letters** either in **lowercase** **(a to z)** or **uppercase** **(A to Z)**. However, you can also mix up **digits (0 to 9)** or an **underscore (\_)** while writing an identifier.

**For example –** Names like **shorpClass**, **style\_1**, and **update\_shape\_to\_db** are all valid identifiers.

**2.** You **can’t use digits to begin** an identifier name. It’ll lead to the syntax error.

**For example –** The name, **0Shape** is incorrect, but **shape1** is a valid identifier.

**3.** Also, the Keywords are reserved, so you should not use them as identifiers.

>>> for=5

SyntaxError: invalid syntax

>>> True=1

SyntaxError: can't assign to keyword

>>> While=9

SyntaxError: can't assign to keyword

**4.** Python Identifiers can also not have special characters **[‘.’, ‘!’, ‘@’, ‘#’, ‘$’, ‘%’]** in their formation.

>>> @index=0

SyntaxError: invalid syntax

>>> isPython?=True

SyntaxError: invalid syntax

>>> $dollar=7

SyntaxError: invalid syntax

### 3.2 Testing If An Identifier Is Valid or Invalid:

* You can test whether a Python identifier is valid or not by using the **keyword.iskeyword()** function. It returns **“True”** if the keyword is correct or **“False”** otherwise.

Please refer below

>>> import keyword

>>> keyword.iskeyword("world")

False

>>> keyword.iskeyword("try")

True

* Another useful method to check if an identifier is valid or not is by calling the **str.isidentifier()** function. But it is only available in **Python 3.0** and onwards.

>>> 'technology'.isidentifier()

True

>>> '1hyderabad'.isidentifier()

False

>>> 'technology\_com'.isidentifier()

True

**4. PYTHON VARIABLES AND OPERATORS**

**Variables:**

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

**4.1 Python Variables:**

* Variables are nothing but reserved **memory locations** to store values. This means that when you create a variable you reserve some space in memory.
* Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables
* Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when we assign a value to a variable. The equal sign (=) is used to assign values to variables.

Example code:

distance = 100  # Integer Assignment

cost = 99.6       # Floating Point Assignment

item = "Mango"  # String Assignment

print(distance)

print(cost)

print(item)

OUTPUT:

100

99.6

Mango

**Multiple Assignment of Variables:**

Python allows you to assign a single value to several variables simultaneously. For example −

                                x = y = z = 5

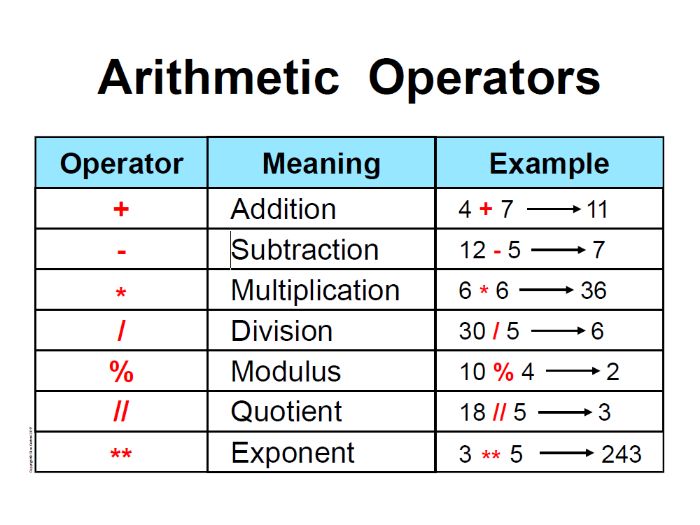
 Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example

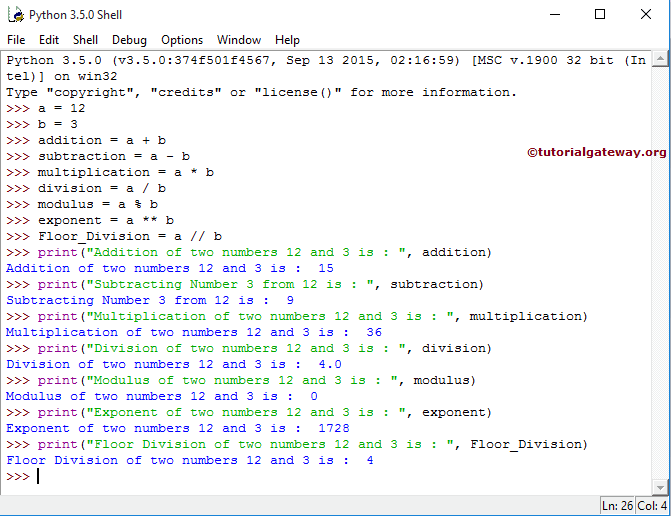
                       a, b, c = 1, 2, "Kaushal"

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "Kaushal" is assigned to the variable c.

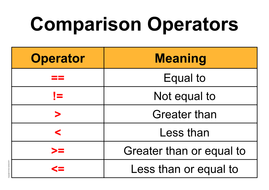
**4.2 Python Basic Operators:**

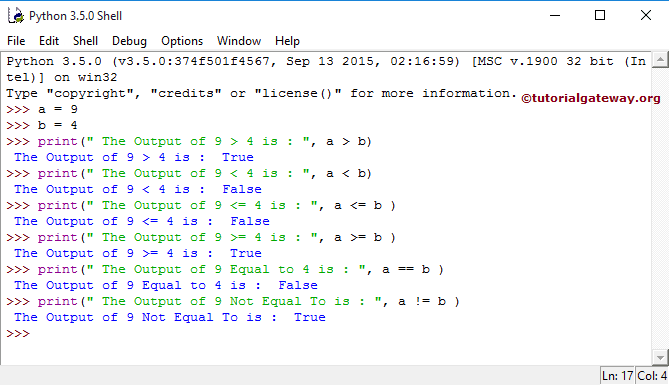
**1. Python Arithmetic operators:**



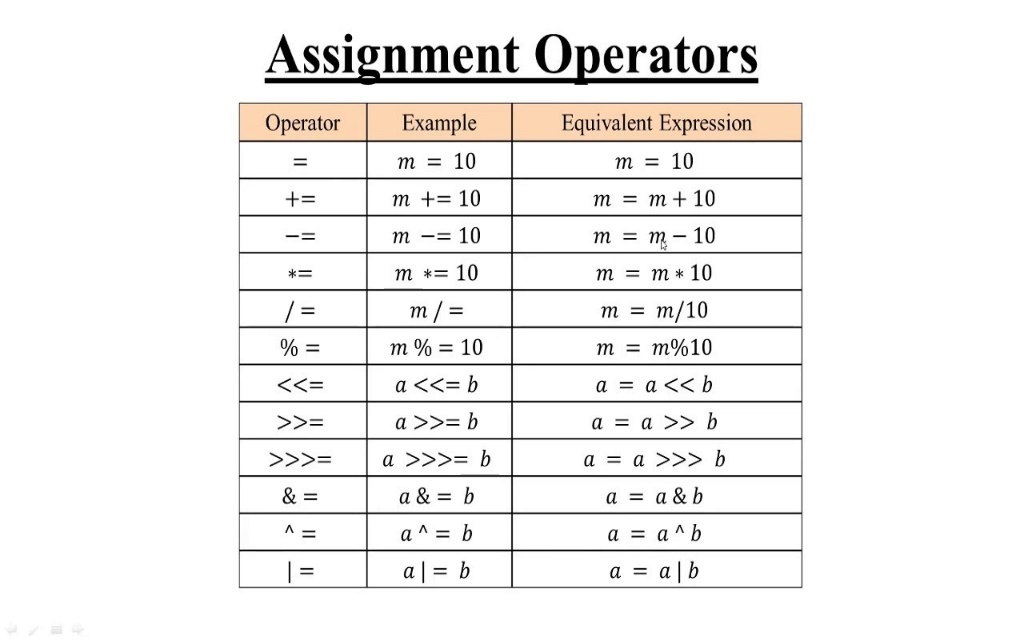


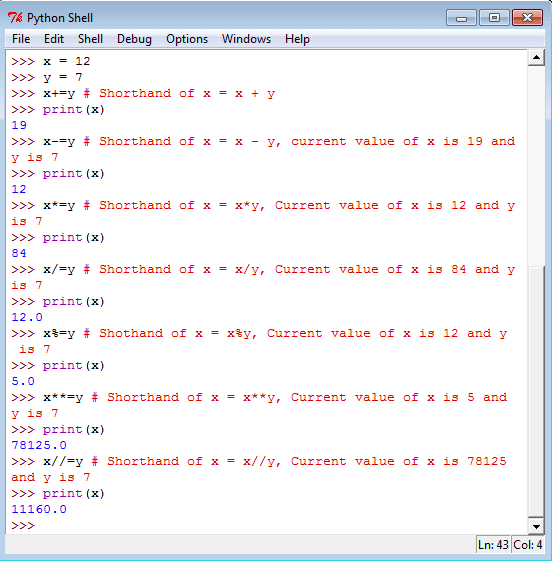
**2. Python Comparison Operators:**



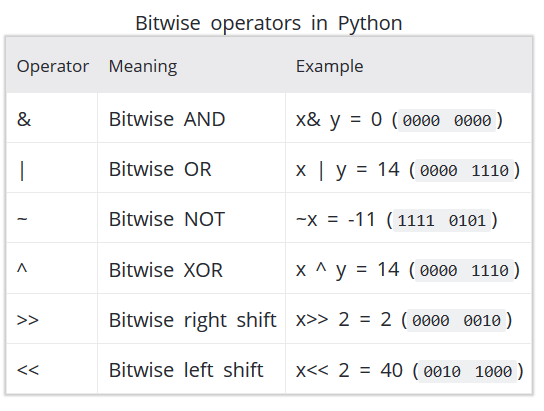


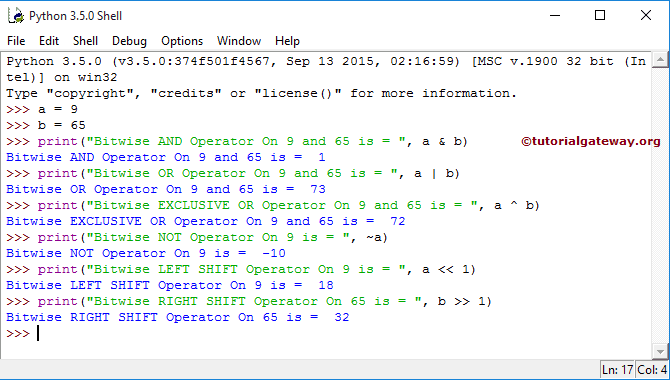
**3. Python Assignment Operators:**



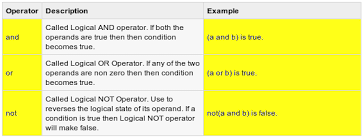


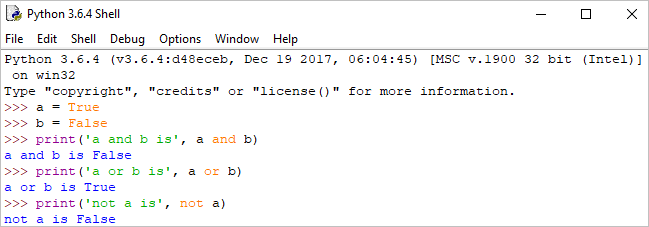
**4. Python Bit wise Operators:**



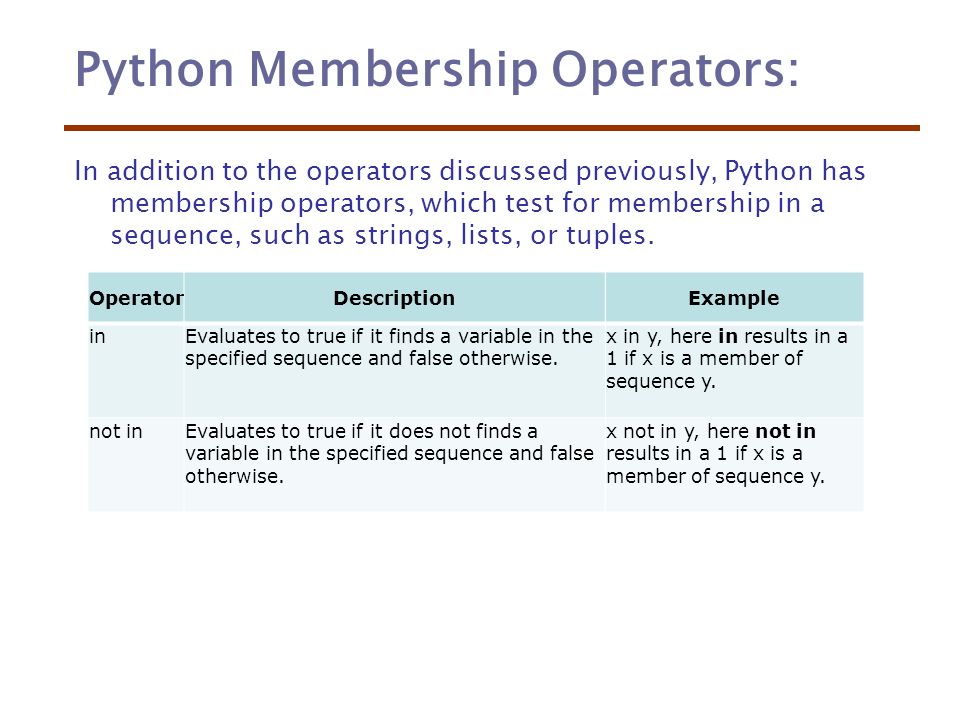


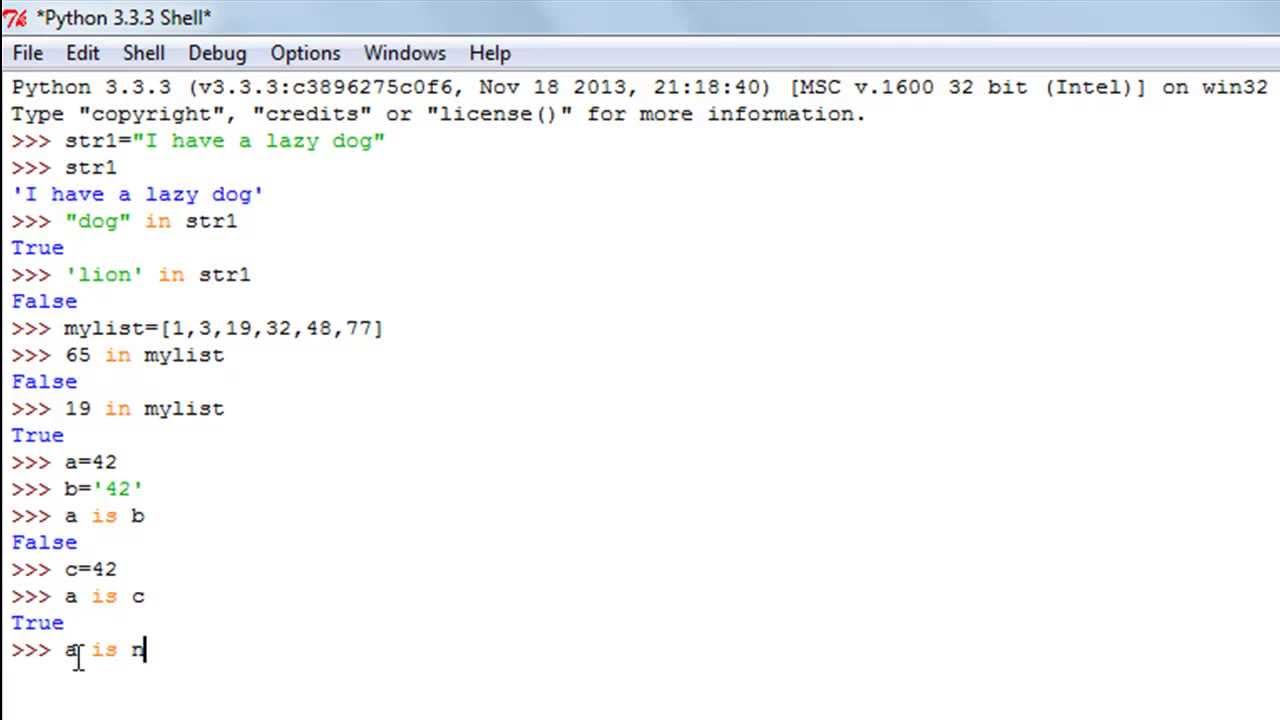
**5. Python Logical Operators:**



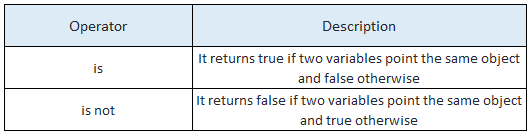


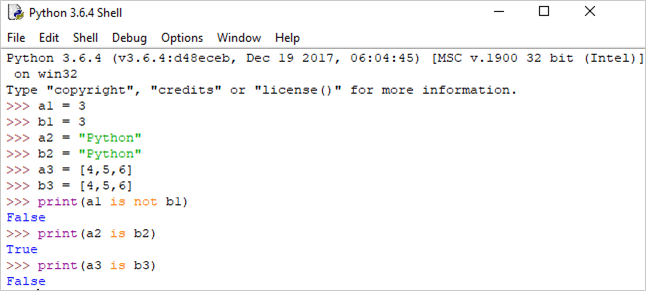
**6. Python Membership Operators:**



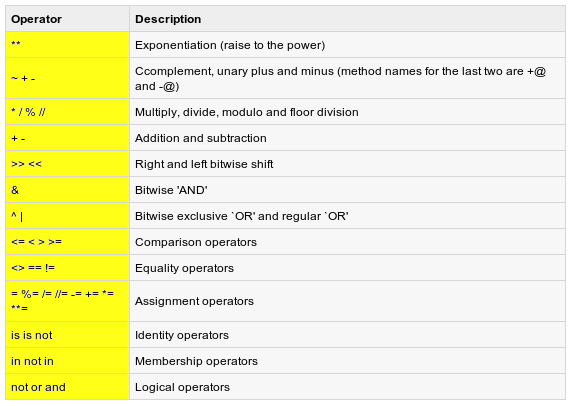


**7. Python Identity Operators:**





**Python Operators Precedence:**



#### 5. [LINES AND INDENTATION, MULTIPLE LINE STATEMENTS, QUOTATION AND COMMENTS IN PYTHON](https://w3-connections.ibm.com/blogs/navredd1/entry/7_LINES_AND_INDENTATION_MULTIPLE_LINE_STATEMENTS_QUOTATION_AND_COMMENTS_IN_PYTHON?lang=en_us)

**5.1. Lines and Indentation:**

* Python doesn't use braces({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.
* The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

For example −

if True:

print ("True")

else:

print ("False")

However, the following block generates an error −

if True:

      print ("Answer")

      print ("True")

else:

print "(Answer")

print ("False")

Thus, in Python all the continuous lines indented with same number of spaces would form a block.

**5.2. Multi Line Statements:**

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue.

For example −

total = item\_one + \

           item\_two + \

           item\_three

Statements contained within the **[], {}, or ()** brackets**do not need to use the line continuation character.**

For example −

days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

**Multiple Statements on a Single Line:**

The **semicolon ( ; )** allows multiple statements on the single line given that neither statement starts a new code block.

Here is a sample snip using the semicolon −

import sys; x = 'foo'; sys.stdout.write(x + '\n')

**Multiple Statement Groups as Suites:**

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

**Header lines** begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite.

For example −

if expression :

   suite

elif expression :

   suite

else :

   suite

**5.3. Quotation in Python:**

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines.

For example, all the following are legal −

1. word = 'word'

2. sentence = "This is a sentence."

3. paragraph = """This is a paragraph. It is

                         made up of multiple lines and sentences."""

**5.4. Comments in Python:**

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

# First comment

print ("Hello, Python!")  # second comment

This produces the following result −

Hello, Python!

You can type a comment on the same line after a statement or expression −

name = “Hello World" # This is again comment

**Python doesn't have multiple-line commenting feature**.

You should comment each line individually as follows −

# This is a comment.

# This is a comment, too.

# This is a comment, too.

# I said that already.

#### 6. PYTHON DATA TYPES

## 6.1 Data Types in Python:

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has **five** standard data types −

•Python Numbers

•Python String

•Python List

•Python Tuple

•Python Dictionary

## 6.2 Python Numbers:

Integers, Floating point numbers and Complex numbers falls under Python Numbers Category.

They are defined as **int**, **float** and **complex** class in Python.

Number objects are created when you assign a value to them.

**For example :**

var1 = 1

var2 = 7

We can delete the reference to a number object by using the **del** statement.

The syntax of the **del**statement is:

**del** var1[,var2[,var3[....,varN]]]]

We can use the **type()**function to know which class a variable or a value belongs to and the **isinstance()** function to check if an object belongs to a particular class.

**Example Program:**

1. a = 5

2. print(a, "is of type", type(a))

3. a = 2.0

4. print(a, "is of type", type(a))

5. a = 1+2j

6. print(a, "is complex number?", isinstance(1+2j,complex))

**Output:**

5, is of type , Integer

2.0, is of type , floating point

1+2j, is complex number?, true

### 6.2.1. Number Type Conversion:

Python converts numbers internally in an expression containing mixed types to a common type for evaluation.

1. Type int(x) to convert x to a plain integer.

2. Type long(x) to convert x to a long integer.

3. Type float(x) to convert x to a floating-point number.

4. Type complex(x) to convert x to a complex number with real part x and imaginary part zero.

5. Type complex(x, y) to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions.

**KEY POINTS:**

1. **Class** is not a Data Type.

2. **//**is integer operation in python 3.0 and **int(..)** is a type cast operator.

3. **NameError** occurs when we execute apple = mango

4. **List** data type can store any value within it.

5. In order to store values in terms of key and value, we use **dictionary** data type.

6. K = 2 + 3l is not complex number because l(or L) stands for Long.

7. **Infinity** is a special case of floating point numbers. It can be obtained by **float(‘inf’)**.

8. ~x is equivalent to -(x+1).

9. Numbers starting with a 0 are octal numbers but 9 isn’t allowed in octal numbers.

10. cmp(x, y) returns 1 if x > y, 0 if x == y and -1 if x < y.

11. ‘+’ cannot be converted to a float.

          Ex: float('24+56') is INVALID.

12. Python rounds off numbers away from 0 when the number to be rounded off is exactly halfway through. round(0.5) is 1 and round(-0.5) is -1.

13. ^ is the Binary XOR operator.

## 6.3 Python Strings:

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

Multi-line strings can be denoted using triple quotes '''or """.

>>> x = "This is first string."

>>> s = ' ' ' a multiple line ' ' '

Python allows for either pairs of single or double quotes.

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.

The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator.

**For example:**

str = 'Hello World!'

1. print (str)          # Prints complete string

2. print (str[0])       # Prints first character of the string

3. print (str[2:5])     # Prints characters starting from 3rd to 5th

4. print (str[2:])      # Prints string starting from 3rd character

5. print (str \* 2)      # Prints string two times

6. print (str + "TEST") # Prints concatenated string

This will produce the following result −

1. Hello World!

2. H

3. llo

4. llo World!

5. Hello World!Hello World!

6. Hello World!TEST

|  |
| --- |
| Python String Methods |
| [**capitalize()**](https://www.programiz.com/python-programming/methods/string/capitalize) - Returns the string with first letter capitalized and the rest lowercased. |
| [**casefold()**](https://www.programiz.com/python-programming/methods/string/casefold) - Returns a lowercase string, generally used for caseless matching. This is more aggressive than the lower() method. |
| [**center()**](https://www.programiz.com/python-programming/methods/string/center) - Center the string within the specified width with optional fill character. |
| [**count()**](https://www.programiz.com/python-programming/methods/string/count) - Count the non-overlapping occurrence of supplied substring in the string. |
| [**encode()**](https://www.programiz.com/python-programming/methods/string/encode) - Return the encoded version of the string as a bytes object. |
| [**endswith()**](https://www.programiz.com/python-programming/methods/string/endswith) - Returns ture if the string ends with the supplied substring. |
| [**expandtabs()**](https://www.programiz.com/python-programming/methods/string/expandtabs) - Return a string where all the tab characters are replaced by the supplied number of spaces. |
| **find()** - Return the index of the first occurrence of supplied substring in the string. Return -1 if not found. |
| **format()**- Format the given string. |
| [**format\_map()**](https://www.programiz.com/python-programming/methods/string/format_map)- Format the given string. |
| [**index()**](https://www.programiz.com/python-programming/methods/string/index) - Return the index of the first occurrence of supplied substring in the string. Raise ValueError if not found. |
| [**isalnum()**](https://www.programiz.com/python-programming/methods/string/isalnum)- Return true if the string is non-empty and all characters are alphanumeric. |
| [**isalpha()**](https://www.programiz.com/python-programming/methods/string/isalpha) - Return true if the string is non-empty and all characters are alphabetic. |
| [**isdecimal()**](https://www.programiz.com/python-programming/methods/string/isdecimal) - Return true if the string is non-empty and all characters are decimal characters. |
| [**isdigit()**](https://www.programiz.com/python-programming/methods/string/isdigit) - Return true if the string is non-empty and all characters are digits. |
| [**isidentifier()**](https://www.programiz.com/python-programming/methods/string/isidentifier) - Return true if the string is a valid identifier. |
| [**islower()**](https://www.programiz.com/python-programming/methods/string/islower) - Return true if the string has all lowercased characters and at least one is cased character. |
| [**isnumeric()**](https://www.programiz.com/python-programming/methods/string/isnumeric) - Return true if the string is non-empty and all characters are numeric. |
| [**isprintable()**](https://www.programiz.com/python-programming/methods/string/isprintable) - Return true if the string is empty or all characters are printable. |
| [**isspace()**](https://www.programiz.com/python-programming/methods/string/isspace)- Return true if the string is non-empty and all characters are whitespaces. |
| [**istitle()**](https://www.programiz.com/python-programming/methods/string/istitle)- Return true if the string is non-empty and titlecased. |
| [**isupper()**](https://www.programiz.com/python-programming/methods/string/isupper) - Return true if the string has all uppercased characters and at least one is cased character. |
| [**join()**](https://www.programiz.com/python-programming/methods/string/join)- Concatenate strings in the provided iterable with separator between them being the string providing this method. |
| [**ljust()**](https://www.programiz.com/python-programming/methods/string/ljust) - Left justify the string in the provided width with optional fill characters. |
| [**lower()**](https://www.programiz.com/python-programming/methods/string/lower) - Return a copy of all lowercased string. |
| [**lstrip()** -](https://www.programiz.com/python-programming/methods/string/lstrip) Return a string with provided leading characters removed. |
| [**maketrans()**](https://www.programiz.com/python-programming/methods/string/maketrans) - Return a translation table. |
| **partition()** - Partition the string at first occurrence of substring (separator) and return a 3-tuple with part before separator, the separator and part after separator. |
| [**replace()**](https://www.programiz.com/python-programming/methods/string/replace)- Replace all old substrings with new substrings. |
| **rfind()** - Return the index of the last occurrence of supplied substring in the string. Return -1 if not found. |
| [**rindex()**](https://www.programiz.com/python-programming/methods/string/rindex)- Return the index of the last occurrence of supplied substring in the string. Raise ValueError if not found. |
| [**rjust()**](https://www.programiz.com/python-programming/methods/string/rjust)- Right justify the string in the provided width with optional fill characters. |
| [**rpartition()**](https://www.programiz.com/python-programming/methods/string/rpartition) - Partition the string at last occurrence of substring (separator) and return a 3-tuple with part before separator, the separator and part after separator. |
| [**rsplit()**](https://www.programiz.com/python-programming/methods/string/rsplit) - Return a list of words delimited by the provided subtring. If maximum number of split is specified, it is done from the right. |
| [**rstrip()**](https://www.programiz.com/python-programming/methods/string/rstrip)- Return a string with provided trailing characters removed. |
| [**split()**](https://www.programiz.com/python-programming/methods/string/split) - Return a list of words delimited by the provided subtring. If maximum number of split is specified, it is done from the left. |
| [**splitlines()**](https://www.programiz.com/python-programming/methods/string/splitlines) - Return a list of lines in the string. |
| [**startswith()**](https://www.programiz.com/python-programming/methods/string/startswith) - Return true if the string starts with the provided substring. |
| [**strip()**](https://www.programiz.com/python-programming/methods/string/strip) - Return a string with provided leading and trailing characters removed. |
| [**swapcase()**](https://www.programiz.com/python-programming/methods/string/swapcase) - Return a string with lowercase characters converted to uppercase and vice versa. |
| [**title()**](https://www.programiz.com/python-programming/methods/string/title) - Return a title (first character of each word capitalized, others lowercased) cased string. |
| [**translate()**](https://www.programiz.com/python-programming/methods/string/translate) - Return a copy of string that has been mapped according to the provided map. |
| [**upper()**](https://www.programiz.com/python-programming/methods/string/upper) - Return a copy of all uppercased string. |
| [**zfill()**](https://www.programiz.com/python-programming/methods/string/zfill) - Return a numeric string left filled with zeros in the provided width. |

## 6. 4 Python Lists:

**Lists** are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]).

To some extent, **lists are similar to arrays in C**. One difference between them is that all the items belonging to a **list can be of different data type**i.e., all the items in a list do not need to be of the same type.

# Empty List

my\_list = []

# list of integers

my\_list = [1,2,3,4]

# list with mixed data types

my\_list = [5,"Hey",15.4]

Also, a list can even have another list as an item. This is called **nested list.**

# nested list

my\_list = [ "micky", [2,0,9], [ 'n' ]]

The values stored in a list can be accessed using the **slice operator ([ ] and [:])** with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (\*) is the repetition operator.

For example:

**my\_list = [ 'p', 'y', 't', 'h', 'o', 'n' ]**

1. print(my\_list[0])  #output : p

2. print(my\_list[3])  #output : h

3. my\_list[4.0]  # **Error !** Only integer can be used for indexing

**nested\_list = [ "boss", [ 2, 0, 1,9]]**

1. print(nested\_list[0][1])  #output : o

2. print(nested\_list([1],[3])  #output : 9

Python allows **negative indexing**for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

For example:

**my\_list = [ 'p', 'y', 't', 'h', 'o', 'n' ]**

1. print(my\_list[-1])  #output : n

2. print(my\_list[-5])  #output : y

### 6.4.1 How to slice lists in Python?

      We can access a range of items in a list by using the slicing operator (colon).

For example:

**my\_list = [ 'p', 'y', 't', 'h', 'o', 'n', 1, 2, 3 ]**

1. print(my\_list[2:5])   #output : t h o       #elements from 3rd to 5th

2. print(my\_list[ : -5])  #output : p y t h   #elements from beginning to 4th

3. print(my\_list[5 : ]   #output : n 1 2 3    #elements from 6th to end

4. print(my\_list[ : ]     #output : p y t h o n 1 2 3  # elements from beginning to end

### 6.4.2 How to change or add elements to a list?

List are mutable, meaning, their elements can be changed unlike [string](https://www.programiz.com/python-programming/string) or [tuple](https://www.programiz.com/python-programming/tuple).

We can use assignment operator (=) to change an item or a range of items.

For Example:

**Even = [ 2, 4, 6, 8 ]**

1. Even[0] = 1   #change the 1st item

2. print(Even)   #output : [ 1, 4, 6, 8 ]

3. Even[1:4] = [ 3, 5, 7 ]  # change 2nd to 4th items

4. print(Even)   #output : [ 1, 3, 5, 7 ]

We can add one item to a list using **append()** method or add several items using **extend()**method.

For Example:

**odd = [1, 3, 5]**

odd.append(7)

print(odd) #Output: [1, 3, 5, 7]

odd.extend([9, 11, 13])

print(odd) #Output: [1, 3, 5, 7, 9, 11, 13]

We can also use **+** operator to combine two lists. This is also called **concatenation.** The **\*** operator repeats a list for the given number of times.

For Example:

**odd = [1, 3, 5]**

print(odd + [9, 7, 5]) #Output: [1, 3, 5, 9, 7, 5]

print(["re"] \* 3) #Output: ["re", "re", "re"]

we can insert one item at a desired location by using the method **insert()** or insert multiple items by squeezing it into an empty slice of a list.

For Example:

**odd = [1, 9]**

odd.insert(1,3)

print(odd) #Output: [1, 3, 9]

odd[2:2] = [5, 7]

print(odd) #Output: [1, 3, 5, 7, 9]

## 6.4.3 How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword **del**. It can even delete the list entirely.

For example:

**my\_list = ['p','r','o','b','l','e','m']**

# delete one item

**del** my\_list[2]

print(my\_list) #Output : ['p', 'r', 'b', 'l', 'e', 'm']

# delete multiple items

**del** my\_list[1:5]

print(my\_list) #Output: ['p', 'm']

# delete entire list

**del** my\_list

print(my\_list) #Error: List not defined

We can use **remove()** method to remove the given item or**pop()** method to remove an item at the given index.

The **pop()**method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the **clear()**method to empty a list.

For Example:

**my\_list = ['p','r','o','b','l','e','m']**

my\_list.remove('p')

print(my\_list) #Output: ['r', 'o', 'b', 'l', 'e', 'm']

print(my\_list.pop(1)) #Output: 'o'

print(my\_list) #Output: ['r', 'b', 'l', 'e', 'm']

print(my\_list.pop(-1)) #Output: 'm'

print(my\_list) #Output: ['r', 'b', 'l', 'e']

my\_list.clear()

print(my\_list) #Output: []

Finally, we can also delete items in a list by **assigning an empty list** to a slice of elements.

>>> **my\_list = ['p','r','o','b','l','e','m']** >>> my\_list[2:3] = [] >>> my\_list

['p', 'r', 'b', 'l', 'e', 'm'] >>> my\_list[2:5] = [] >>> my\_list

['p', 'r', 'm']

### 6.4.4 Python List Methods:

Methods that are available with list object in Python programming are tabulated below.

They are accessed as **list.method()**. Some of the methods have already been used above.

|  |
| --- |
| Python List Methods |
| **append()** - Add an element to the end of the list |
| **extend()** - Add all elements of a list to the another list |
| **insert()** - Insert an item at the defined index |
| **remove()** - Removes an item from the list |
| **pop()** - Removes and returns an element at the given index |
| **clear()** - Removes all items from the list |
| **index()** - Returns the index of the first matched item |
| **count()** - Returns the count of number of items passed as an argument |
| **sort()** - Sort items in a list in ascending order |
| **reverse()** - Reverse the order of items in the list |
| **copy()** - Returns a shallow copy of the list |

For Example:

**my\_list = [3, 8, 1, 6, 0, 8, 4]**

print(my\_list.index(8)) #Output : 1

print(my\_list.count(8)) #Output : 2

my\_list.sort()

print(my\_list) #Output : [0, 1, 3, 4, 6, 8, 8]

my\_list.reverse()

print(my\_list) #Output : [8, 8, 6, 4, 3, 1, 0]

**6.5 PYTHON TUPLES:**

A **tuple is another sequence data type** that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, **tuples are enclosed within parentheses.**

The main differences between lists and tuples are:

1.**Lists**are enclosed in brackets ( [ ] ) and their elements and size can be changed.

2. **Tuples**are enclosed in parentheses ( ( ) ) and cannot be updated.

Tuples can be thought of as **read-only** lists.

For example:

**tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )**

tinytuple = (123, 'john')

1.print (tuple)               # Prints complete tuple

2. print (tuple[0])         # Prints first element of the tuple

3. print (tuple[1:3])      # Prints elements starting from 2nd till 3rd

4. print (tuple[2:])        # Prints elements starting from 3rd element

5. print (tinytuple \* 2)  # Prints tuple two times

6. print (tuple + tinytuple) # Prints concatenated tuple

This produce the following result −

1.('abcd', 786, 2.23, 'john', 70.200000000000003)

2. abcd

3. (786, 2.23)

4. (2.23, 'john', 70.200000000000003)

5. (123, 'john', 123, 'john')

6. ('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')

**Creating a tuple with one element** is a bit tricky.

Having one element within parentheses is not enough. We will **need a trailing comma** to indicate that it is in fact a tuple.

For Example:

# only parentheses is not enough

my\_tuple = ("hello")

print(type(my\_tuple))  #Output: <class 'str'>

# need a comma at the end

my\_tuple = ("hello",)

print(type(my\_tuple))    #Output: <class 'tuple'>

# parentheses is optional

my\_tuple = "hello",

print(type(my\_tuple))   #Output: <class 'tuple'>

**6.5.1 Accessing Elements in a Tuple:**

There are various ways in which we can access the elements of a tuple.

**1. Indexing**

We can use the index operator [] to access an item in a tuple where the index starts from 0.So, a tuple having 6 elements will have index from 0 to 5. Trying to access an element other that (6, 7,...) will raise an **IndexError.**

The index must be an integer, so we cannot use float or other types. This will result into **TypeError.**

Likewise, nested tuple are accessed using **nested indexing.**

For Example:

**my\_tuple = ('p','e','r','m','i','t')**

print(my\_tuple[0])    #Output: 'p'

print(my\_tuple[5])    #Output: 't'

# index must be in range

# If you uncomment line 14, you will get an error.

print(my\_tuple[6])    #**IndexError**: list index out of range

# index must be an integer

my\_tuple[2.0]      # TypeError: list indices must be integers, not float

# nested tuple

**n\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))**

# nested index

print(n\_tuple[0][3])   # Output: 's'

print(n\_tuple[1][1])   # Output: 4

**2. Negative Indexing**

Python allows negative indexing for its sequences.The index of -1 refers to the last item, -2 to the second last item and so on.

For example:

**my\_tuple = ('p','e','r','m','i','t')**

print(my\_tuple[-1])   # Output: 't'

print(my\_tuple[-6])   # Output: 'p'

**3. Slicing**

We can access a range of items in a tuple by using the slicing operator -**colon ":".**

For Example:

**my\_tuple = ('p','y','t','h','o','n','1','2','3')**

# elements 2nd to 4th

print(my\_tuple[1:4])     # Output: ('y', 't', 'h')

# elements beginning to 2nd

print(my\_tuple[:-7])     # Output: ('p', 'y')

# elements 8th to end

print(my\_tuple[7:])     # Output: ('1', '2')

# elements beginning to end

print(my\_tuple[:])      # Output: ('p', 'y', 't', 'h', 'o', 'n', '1', '2', '3')

### 6.5.2 Changing a Tuple:

Unlike lists, tuples are immutable. This means that **elements of a tuple cannot be changed once it** **has been assigned**. But, if the element is itself a mutable datatype like list, its nested items can be changed. We can also assign a tuple to different values (reassignment).

For Example:

**my\_tuple = (4, 2, 3, [6, 5])**

# we cannot change an element

my\_tuple[1] = 9   # **TypeError**: 'tuple' object does not support item assignment

# but item of mutable element can be changed

my\_tuple[3][0] = 9

print(my\_tuple)     # Output: (4, 2, 3, [9, 5])

# tuples can be reassigned

my\_tuple = ('p','y','t','h','o','n','1','2','3')

print(my\_tuple)     # Output: ('p', 'y', 't', 'h', 'o', 'n', '1', '2', '3')

We can use **+ operator** to combine two tuples. This is called **concatenation**.

We can also **repeat**the elements in a tuple for a given number of times using the **\* operator**. Both + and \* operations result into a new tuple.

For Example:

# Concatenation

print((1, 2, 3) + (4, 5, 6))  # Output: (1, 2, 3, 4, 5, 6)

# Repeat

print(("Repeat",) \* 3)    # Output: ('Repeat', 'Repeat', 'Repeat')

**6.5.3 Deleting a Tuple:**

We cannot change the elements in a tuple. That means we **cannot delete or remove items from a tuple**. But deleting atuple entirely is possible using the keyword **del**.

For Example:

**my\_tuple = ('p','y','t','h','o','n','1','2','3')**

# can't delete items

del my\_tuple[3]    # TypeError: 'tuple' object doesn't support item deletion

del my\_tuple   # can delete entire tuple

my\_tuple      # NameError: name 'my\_tuple' is not defined

### 6.5.4 Python Tuple Methods:

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

|  |  |
| --- | --- |
| Method | Description |
| **count(x)** | Return the number of items that is equal to **x** |
| **index(x)** | Return index of first item that is equal to **x** |

For Example:

**my\_tuple = ('a','p','p','l','e',)**

# Count

print(my\_tuple.count('p'))    # Output: 2

# Index

print(my\_tuple.index('l'))    # Output: 3

**Built-in Functions with Tuple**

|  |  |
| --- | --- |
| Function | Description |
| **all()** | Return True if all elements of the tuple are true (or if the tuple is empty). |
| **any()** | Return True if any element of the tuple is true. If the tuple is empty, return False. |
| **enumerate()** | Return an enumerate object. It contains the index and value of all the items of tuple as pairs. |
| **len()** | Return the length (the number of items) in the tuple. |
| **max()** | Return the largest item in the tuple. |
| **min()** | Return the smallest item in the tuple |
| **sorted()** | Take elements in the tuple and return a new sorted list (does not sort the tuple itself). |
| **sum()** | Retrun the sum of all elements in the tuple. |
| **tuple()** | Convert an iterable (list, string, set, dictionary) to a tuple. |

#### 6.6 PYTHON DICTIONARY:

Python dictionary is **an unordered collection of items**. Python's dictionaries are kind of hash table type.

While other compound data types have only value as an element, a dictionary has a**key : value**pair. Dictionaries are optimized to retrieve values when the key is known.

A **dictionary key** can be almost any Python type, but **are** usually **numbers or strings**. **Values**, on the other hand, can be **any arbitrary Python object.**

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

**For Example:**

# empty dictionary

my\_dict = {}

# dictionary with integer keys

my\_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys

my\_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()

my\_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair

my\_dict = dict([(1,'apple'), (2,'ball')])

### 6.6.1 Accessing elements from a dictionary:

While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the **get()**method.

The difference while using **get()** is that it returns **None** instead of **KeyError**, if the key is not found.

For Example:

**my\_dict = {'name':'Jack', 'age': 26}**

print(my\_dict['name'])    # Output: Jack

print(my\_dict.get('age'))  # Output: 26

# Trying to access keys which doesn't exist throws error

# my\_dict.get('address') # Output: None

my\_dict['address']     # Error

## ****6.6.2 Change or Add elements in a dictionary:****

**Dictionary are mutable**. We can add new items or change the value of existing items using assignment operator**(=)**.

If the key is already present, value gets updated, else a new **key: value** pair is added to the dictionary.

For Example:

**my\_dict = {'name':'Jack', 'age': 26}**

# update value

my\_dict['age'] = 27

print(my\_dict)     #Output: {'age': 27, 'name': 'Jack'}

# add item

my\_dict['address'] = 'Downtown'

print(my\_dict)   # Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}

## 6.6.3 Delete or Remove elements from a dictionary:

We can remove **a particular item** in a dictionary by using the method **pop()**. This method removes as item with the provided key and returns the value.

The method, **popitem()** can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the **clear()**method.

We can also use the **del**keyword to remove individual items or the entire dictionary itself.

For Example:

# create a dictionary

**squares = {1:1, 2:4, 3:9, 4:16, 5:25}**

# remove a particular item

print(squares.pop(4))   # Output: 16

print(squares)          # Output: {1: 1, 2: 4, 3: 9, 5: 25}

# remove an arbitrary item

print(squares.popitem())    # Output: (1, 1)

print(squares)    # Output: {2: 4, 3: 9, 5: 25}

# delete a particular item

**del squares[5]**

print(squares)    # Output: {2: 4, 3: 9}

# remove all items

**squares.clear()**

print(squares)    # Output: {}

# delete the dictionary itself

**del squares**

print(squares)   # Throws Error

## 6.6.4 Python Dictionary Methods:

|  |  |
| --- | --- |
| Method | Description |
| **clear()** | Remove all items form the dictionary. |
| **copy()** | Return a shallow copy of the dictionary. |
| **fromkeys(seq[, v])** | Return a new dictionary with keys from seq and value equal to v(defaults to None). |
| **get(key[,d])** | Return the value of key. If key doesnot exit, return d (defaults to None). |
| **items()** | Return a new view of the dictionary's items (key, value). |
| **keys()** | Return a new view of the dictionary's keys. |
| **pop(key[,d])** | Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError. |
| **popitem()** | Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty. |
| **setdefault(key[,d])** | If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None). |
| **update([other])** | Update the dictionary with the key/value pairs from other, overwriting existing keys. |
| **values()** | Return a new view of the dictionary's values |

For Example:

**marks = {}.fromkeys(['Math','English','Science'], 0)**

print(marks)  # Output: {'English': 0, 'Math': 0, 'Science': 0}

for item in marks.items():

    print(item)

list(sorted(marks.keys()))  # Output: ['English', 'Math', 'Science']

**squares = {x: x\*x for x in range(6)}**

print(squares)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

**odd\_squares = {x: x\*x for x in range(11) if x%2 == 1}**

print(odd\_squares)  # Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

### Built-in Functions with Dictionary:

|  |  |
| --- | --- |
| Function | Description |
| **all()** | Return True if all keys of the dictionary are true (or if the dictionary is empty). |
| **any()** | Return True if any key of the dictionary is true. If the dictionary is empty, return False. |
| **len()** | Return the length (the number of items) in the dictionary. |
| **cmp()** | Compares items of two dictionaries. |
| **sorted()** | Return a new sorted list of keys in the dictionary. |

For Example:

**squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}**

print(len(squares))   # Output: 5

print(sorted(squares))  # Output: [1, 3, 5, 7, 9]

#### 7. PYTHON IF….ELSE STATEMENT:

**Decision making** is required when we want to execute a code only if a certain condition is satisfied.

The **if…elif…else** statement is used in Python for decision making.

### 7.1 Python if Statement:

### Syntax

if test expression:

statement(s)

Here, the program evaluates the **test expression**and will execute statement(s) only if the text expression is **True**.

If the text expression is **False**, the statement(s) is not executed.

In Python, the body of the **if statement** is indicated by the **indentation**. Body starts with an indentation and the first unindented line marks the end.

Python interprets non-zero values as **True**. **None** and **0** are interpreted as **False.**

### Example: Python if Statement:

# If the number is positive, we print an appropriate message

num = 3

if num > 0:

    print(num, "is a positive number.")    # indented

print("This is always printed.")              # Unindented

num = -1

if num > 0:

    print(num, "is a positive number.")    # indented

print("This is also always printed.")      # Unindented

**Output:**

3, is a positive number.

This is always printed.

This is also always printed.

## ****7.2 Python if...else Statement:****

### **Syntax of if...else**

if test expression:

Body of if

else:

    Body of else

The**if..else** statement evaluates **test expression** and will execute body of**if**only when test condition is **True**.

If the condition is **False**, **body of else** is executed. Indentation is used to separate the blocks.

### Example of if...else:

# Program checks if the number is positive or negative

# And displays an appropriate message

num = -3

if num >= 0:

    print("Positive or Zero")

else:

    print("Negative number")

**Output:**

Negative Numer

## 7.3 Python if...elif...else Statement:

### Syntax of if...elif...else

if test expression:

      Body of if

elif test expression:

       Body of elif

else:

      Body of else

The **elif** is short for else if. It allows us to check for multiple expressions.

If the condition for **if** is **False**, it checks the condition of the next **elif** block and so on.

If all the conditions are **False**, body of else is executed.

Only one block among the several **if...elif...else** blocks is executed according to the condition.

The **if**block can have only one **else**block. But it can have multiple**elif** blocks.

### Example of if...elif...else:

# In this program, we check if the number is positive or negative or zero and

# display an appropriate message

num = 3.4

if num > 0:

    print("Positive number")

elif num == 0:

    print("Zero")

else:

    print("Negative number")

Output:

Positive Number

## 7.4 Python Nested if statements:

We can have a **if...elif...else** statement inside another **if...elif...else**statement. This is called **Nesting**in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

### Python Nested if Example:

# In this program, we input a number

# check if the number is positive or

# negative or zero and display an appropriate message

# This time we use nested if

num = float(input("Enter a number: "))

if num >= 0:

  if num == 0:

  print("Zero")

  else:

  print("Positive number")

else:

  print("Negative number")

**Output 1**

Enter a number: 5

Positive number

**Output 2**

Enter a number: -1

Negative number

**Output 3**

Enter a number: 0

Zero

#### 8. PYTHON FOR LOOP

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called **traversal.**

### Syntax of for Loop:

for val in sequence:

Body of for

Here, **val** is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

### Example: Python for Loop

# Program to find the sum of all numbers stored in a list

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]   # List of numbers

sum = 0   # variable to store the sum

# iterate over the list

for val in numbers:

        sum = sum+val

print("The sum is", sum)    # Output: The sum is 48

## 8.1 The range() function:

We can generate a sequence of numbers using **range()**function.

**Examaple:**

**range(10)** will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as **range(start,stop,step size)**. step size defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function **list()**.

For Example:

1. print(range(10)) # Output: range(0, 10)

2. print(list(range(10))) # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

3. print(list(range(2, 8))) # Output: [2, 3, 4, 5, 6, 7]

4. print(list(range(2, 20, 3)) # Output: [2, 5, 8, 11, 14, 17]

We can use the **range()** function in for loops to iterate through a sequence of numbers.

It can be combined with the**len()** function to iterate though a sequence using indexing.

For Example:

# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index

for i in range(len(genre)):

          print("I like", genre[i])

Output:

I like pop

I like rock

​I like jazz

## 8.2 for loop with else:

**A for loop can have an optional else block as well.** The **else** part is executed if the items in the sequence used in for loop exhausts.

**Break statement** can be used to stop for loop. In such case, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

For Example:

digits = [0, 1, 5]

for i in digits:

    print(i)

else:

    print("No items left.")

Output:

0

1

5

No items left.

#### 9. PYTHON WHILE LOOP AND PASS STATEMENT:

## 9.1 Python While Loop:

The **while loop** in Python is used to **iterate over a block of code** as long as the **test expression** (condition) is **true.**

We generally use this loop when we **don't know beforehand**, the number of times to iterate.

### Syntax of while Loop in Python:

while test\_expression:

     Body of while

In while loop, test expression is checked first. The body of the loop is entered only if the **test\_expression** evaluates to **True**. After one iteration, the test expression is checked again. This process continues until the **test\_expression** evaluates to **False**.

In Python, the body of the while loop is determined through indentation.

Body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as **True**.**None** and **0** are interpreted as **False**.

### Example: Python while Loop

# Program to add natural numbers upto sum = 1+2+3+...+n

# To take input from the user,

n = int(input("Enter n: "))

# n = 10

# initialize sum and counter

sum = 0

i = 1

while i <= n:

    sum = sum + i

    i = i+1    # update counter

# print the sum

print("The sum is", sum)

Output:

Enter n: 10

The sum is 55

**9.1.1 While loop with else:**

The **else** part is executed if the condition in the while loop evaluates to **False**.

The while loop can be terminated with a **break statement**. In such case, the **else** part is ignored. Hence, a while loop's **else** part runs if no break occurs and the condition is **false.**

**For  Example:**

# Example to illustrate the use of else statement

# with the while loop

counter = 0

while counter < 3:

    print("Inside loop")

    counter = counter + 1

else:

    print("Inside else")

Output:

Inside loop

Inside loop

Inside loop

Inside else

**9.2 Python Pass Statement:**

In Python programming, **pass** is a null statement. The difference between a comment and **pass** statement in Python is that, while the interpreter ignores a comment entirely, **pass** is not ignored.

However, **nothing happens when pass is executed**. It results into no operation (NOP).

**Syntax of Pass:**

pass

We generally use it as a placeholder.

Suppose we have a **loop** or a **function** that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the **pass** statement to construct a body that does nothing.

**For Example:**

# pass is just a placeholder for functionality to be added later.

sequence = {'p', 'a', 's', 's'}

for val in sequence:

    pass

We can do the same thing in an empty function or **class** as well.

def function(args):       pass

class example:       pass

#### 10. PYTHON BREAK AND CONTINUE STATEMENTS:

In Python, break and continue statements can **alter the flow**of a normal loop.

Loops iterate over a block of code until test expression is false, but sometimes we wish to **terminate the current iteration** or even the whole loop without checking test expression.

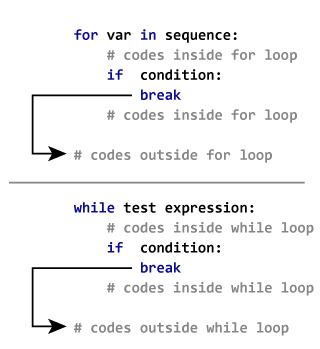
## 10.1 Python break statement

**The break statement terminates the loop containing it.** Control of the program flows to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

**Syntax of Break:**

break



#### Example: Python break

# Use of break statement inside loop

for val in "string":

    if val == "i":

        break

    print(val)

print("The end")

Output:

s

t

r

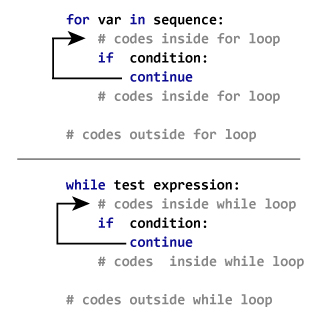
The end

## 10.2 Python continue statement

The continue statement is used **to skip the rest of the code**inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

### Syntax of Continue

continue



### Example: Python continue

# Program to show the use of continue statement inside loops

for val in "string":

    if val == "i":

        continue

    print(val)

print("The end")

Output:

s

t

r

n

g

The end

#### 11. PYTHON FUNCTIONS:

A function is a reusable block of code which performs operations specified in the function.They let you break down tasks and allow you to reuse your code in different programs.

We can define functions to provide the required functionality. Here are simple rules to define a function in Python

* Functions blocks begin **def** followed by the function **name** and parentheses ().
* There are input parameters or arguments that should be placed within these parentheses.
* You can also define parameters inside these parentheses.
* There is a body within every function that starts with a colon (:) and is indented.
* You can also place documentation before the body
* The statement **return** exits a function, optionally passing back a value.

**Syntax:**

def function\_name(parameters):

           """ Doc String """

           statement(s)

**Example of a function:**

def greet(name):

 """This function greets to

 the person passed in as

 parameter"""

        print("Hello, " + name + ". Good morning!")

To call a function we simply type the function name with appropriate

parameters.

>>> greet(pavan kalyan)

Hello, pavan kalyan. Good morning!

**Syntax of return:**

return [expression\_list]

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the **return** statement itself is not present inside a function, then the function will return the **None** object.

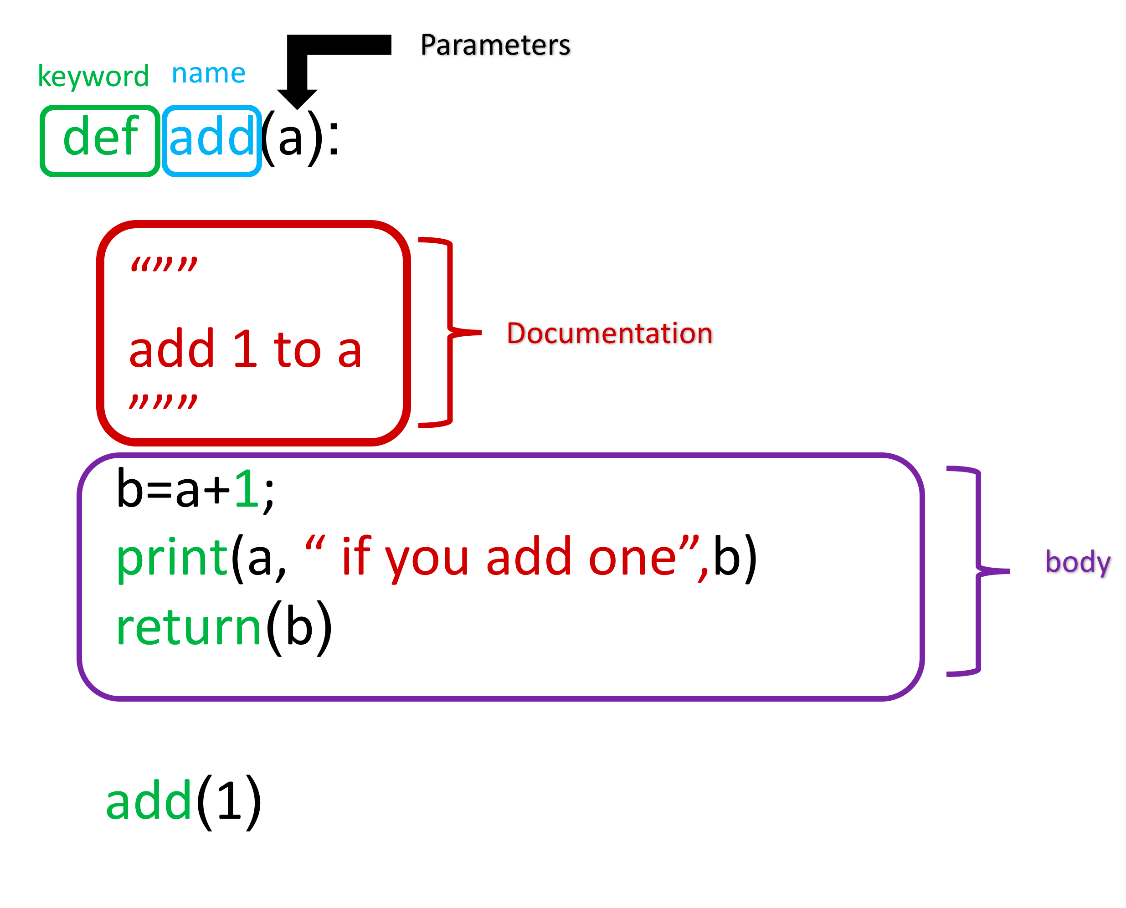
>>> print(greet("Feb"))

Hello, Feb. Good morning!

None

Here, **None** is the returned value.

**Example of a function:**



**EXAMPLE:**

def absolute\_value(num):

 """This function returns the absolute

 value of the entered number"""

      if num >= 0:

             return num

      else:

            return -num

print(absolute\_value(2))         # Output: 2

print(absolute\_value(-4))       # Output: -4

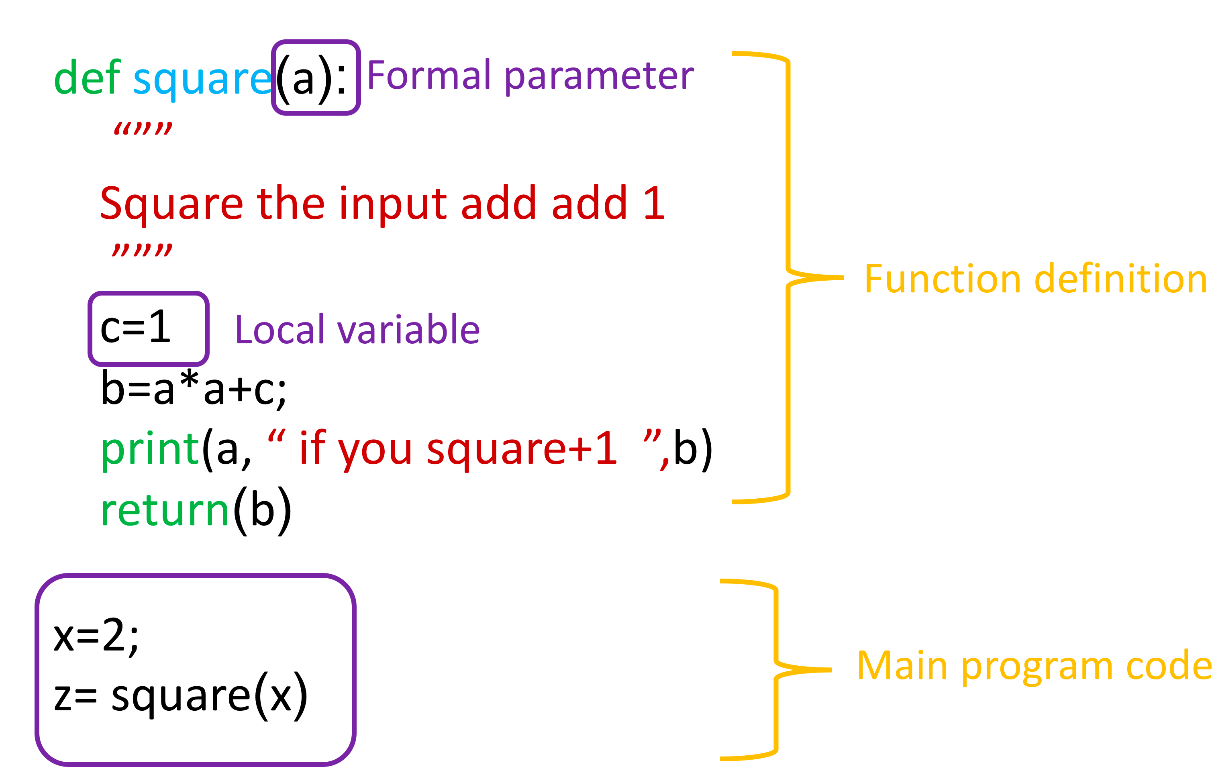
**Variables:**

The input to a function is called a **formal parameter**.

A variable that is declared inside a function is called a **local variable**. The parameter only exists within the function (i.e. the point where the function starts and stops).

A variable that is declared outside a function definition is a **global variable**, and its value is accessible and modifiable throughout the program.

**For Example:**



**Scope and Lifetime of variables:**

**Scope of a variable** is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

**Lifetime of a variable** is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

def my\_func():

      x = 10

      print("Value inside function:",x)

x = 20

my\_func()

print("Value outside function:",x)

OUTPUT:

Value inside function: 10

Value outside function: 20

## Types of Functions

Basically, we can divide functions into the following two types:

1. [Built-in functions](https://www.programiz.com/python-programming/built-in-function) - Functions that are built into Python.
2. [User-defined functions](https://www.programiz.com/python-programming/user-defined-function) - Functions defined by the users themselves.

#### 12. PYTHON FILES:

**File** is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to **read from** or **write to a file** we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

**12.1 How to open a file?**

Python has a built-in function **open()** to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

>>> f = open("filename.txt")   # open file in current directory.

>>> f = open("C:/Python33/README.txt") # specifying full path

We can specify the mode while opening a file. In mode, we specify whether we want to read **'r'**, write **'w'** or append **'a'**to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

|  |  |
| --- | --- |
| Python File Modes | |
| Mode | Description |
| 'r' | Open a file for reading. (default) |
| 'w' | Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists. |
| 'x' | Open a file for exclusive creation. If the file already exists, the operation fails. |
| 'a' | Open for appending at the end of the file without truncating it. Creates a new file if it does not exist. |
| 't' | Open in text mode. (default) |
| 'b' | Open in binary mode. |
| '+' | Open a file for updating (reading and writing) |

>>> f = open("test.txt") # equivalent to 'r' or 'rt'

>>> f = open("test.txt",'w') # write in text mode

>>> f = open("img.bmp",'r+b') # read and write in binary mode

when working with files in text mode, it is highly recommended to specify the encoding type.

>>> f = open("test.txt",mode = 'r', encoding = 'utf-8')

**12.2 How to close a file Using Python?**

When we are done with operations to the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file and is done using Python **close()** method.

f = open("test.txt",encoding = 'utf-8')

# perform file operations

f.close()

**12.3 How to write to File Using Python?**

In order to write into a file in Python, we need to open it in write **'w'**, append **'a'** or exclusive creation **'x'**mode.

We need to be careful with the **'w'** mode as it will overwrite into the file if it already exists. All previous data is erased.

Writing a string or sequence of bytes (for binary files) is done using **write()** method. This method returns the number of characters written to the file.

with open("Firstfile.txt",'w',encoding = 'utf-8') as f:

       f.write("my first file\n")

This program will create a new file named **'Firstfile.txt'**if it does not exist. If it does exist, it is overwritten.

We must include the newline characters ourselves to distinguish different lines.

## 12.4 How to ****read**** files in Python?

To read a file in Python, we must open the file in reading mode.

There are various methods available for this purpose. We can use the **read(size)** method to read in **size** number of data. If **size** parameter is not specified, it reads and returns up to the end of the file.

>>> f = open("Firstfile.txt",'r',encoding = 'utf-8')

>>> f.read(4) # read the first 4 data

'This'

>>> f.read(4) # read the next 4 data

' is '

>>> f.read()# read in the rest till end of file

'my first file\n'

 >>> f.read() # further reading returns empty sting

 ''

We can change our current file cursor (position) using the **seek()** method. Similarly, the **tell()** method returns our current position (in number of bytes).

>>> f.tell() # get the current file position 22

>>> f.seek(0) # bring file cursor to initial position 0

 >>> print(f.read()) # read the entire file This is my first file

We can read a file line-by-line using a **for loop**. This is both efficient and fast.

>>> for line in f:

print(line, end = '')

This is my first file

Alternately, we can use **readline()** method to read individual lines of a file. This method reads a file till the newline, including the newline character.

>>> f.readline()

'This is my first file\n'

>>> f.readline()

''

Lastly, the **readlines()** method returns a list of remaining lines of the entire file. All these reading method return empty values when end of file (EOF) is reached.

>>> f.readlines()

['This is my first file\n']

## 12.5 Python File Methods:

|  |  |
| --- | --- |
| Method | Description |
| close() | Close an open file. It has no effect if the file is already closed. |
| detach() | Separate the underlying binary buffer from the TextIOBaseand return it. |
| fileno() | Return an integer number (file descriptor) of the file. |
| flush() | Flush the write buffer of the file stream. |
| isatty() | Return True if the file stream is interactive. |
| read(n) | Read atmost n characters form the file. Reads till end of file if it is negative or None. |
| readable() | Returns True if the file stream can be read from. |
| readline(n=-1) | Read and return one line from the file. Reads in at most nbytes if specified. |
| readlines(n=-1) | Read and return a list of lines from the file. Reads in at most n bytes/characters if specified. |
| seek(offset,from=SEEK\_SET) | Change the file position to offset bytes, in reference to from (start, current, end). |
| seekable() | Returns True if the file stream supports random access. |
| tell() | Returns the current file location. |
| truncate(size=None) | Resize the file stream to size bytes. If size is not specified, resize to current location. |
| writable() | Returns True if the file stream can be written to. |
| write(s) | Write string s to the file and return the number of characters written. |
| writelines(lines) | Write a list of lines to the file. |

#### 13. PYTHON DIRECTORY AND FILES MANAGEMENT:

If there are a large number of **files to handle**in your Python program, you can arrange your code within different directories to make things more manageable.

A directory or folder is a collection of files and sub directories. Python has the **os**module, which provides us with many useful methods to work with directories (and files as well).

## ****13.1 Get Current Directory****

We can get the present working directory using the **getcwd()** method.This method returns the current working directory in the form of a string.

We can also use the**getcwdb()** method to get it as bytes object.

>>> import os

>>> os.getcwd()

'C:\\Program Files\\PyScripter'

>>> os.getcwdb()

b'C:\\Program Files\\PyScripter'

The extra backslash implies escape sequence. The **print()** function will render this properly.

>>> print(os.getcwd()) C:\Program Files\PyScripter

## 13.2 Changing Directory

We can change the current working directory using the **chdir()** method.

The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash (\) to separate path elements.

It is safer to use escape sequence when using the backward slash.

>>> os.chdir('C:\\Python33')

>>> print(os.getcwd())

C:\Python33

## 13.3 List Directories and Files

All files and sub directories inside a directory can be known using the **listdir()** method.

This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

>>> print(os.getcwd())

C:\Python33

>>> os.listdir()

['DLLs', 'Doc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'python.exe', 'pythonw.exe', 'README.txt', 'Scripts', 'tcl', 'Tools']

>>> os.listdir('G:\\')

['$RECYCLE.BIN', 'Movies', 'Music', 'Photos', 'Series', 'System Volume Information']

## 13.4 Making a New Directory

We can make a new directory using the **mkdir()** method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

>>> os.mkdir('test')

>>> os.listdir()

['test']

## 13.5 Renaming a Directory or a File

The **rename()** method can rename a directory or a file.

The first argument is the old name and the new name must be supplies as the second argument.

>>> os.listdir()

['test']

>>> os.rename('test','new\_one')

>>> os.listdir()

['new\_one']

## 13.6 Removing Directory or File

A file can be removed (deleted) using the **remove()** method.

Similarly, the **rmdir()** method removes an empty directory.

>>> os.listdir()

['new\_one', 'old.txt']

>>> os.remove('old.txt')

>>> os.listdir()

['new\_one']

>>> os.rmdir('new\_one')

>>> os.listdir()

[]

However, note that **rmdir()**method can only remove empty directories.

In order to **remove a non-empty directory** we can use the **rmtree()** method inside the **shutil** module.

>>> os.listdir()

['test']

>>> os.rmdir('test')

Traceback (most recent call last):

 ... OSError: [WinError 145] The directory is not empty: 'test'

>>> import shutil

>>> shutil.rmtree('test')

>>> os.listdir()

[]

#### 14. PYTHON EXCEPTIONS:

## 14. 1 Python Errors and Built-in Exceptions:

When writing a program, we will encounter errors.

Error caused by not following the proper structure (syntax) of the language is called **syntax error**or **parsing error.**

>>> if a < 3

File "<interactive input>",

line 1 if a < 3

SyntaxError: invalid syntax

We can notice here that a colon is missing in the **if** statement.

Errors can also occur at runtime and these are called **Exceptions**.

**For Example:**

When a file we try to open does not exist- **FileNotFoundError**,

Dividing a number by zero ----**ZeroDivisionError**,

Module we try to import is not found ----**ImportError** etc

Whenever these type of runtime error occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

>>> 1 / 0

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

ZeroDivisionError: division by zero

>>> open("imaginary.txt")

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'

## 14.2 Python Built-in Exceptions

We can view all the built-in exceptions using the **local()** built-in functions as follows.

>>> locals()['\_\_builtins\_\_']

|  |  |
| --- | --- |
| Exception | Cause of Error |
| **AssertionError** | Raised when **assert** statement fails. |
| **AttributeError** | Raised when attribute assignment or reference fails. |
| **EOFError** | Raised when the **input()** functions hits end-of-file condition. |
| **FloatingPointError** | Raised when a floating point operation fails. |
| **GeneratorExit** | Raise when a generator's **close()** method is called. |
| **ImportError** | Raised when the imported module is not found. |
| **IndexError** | Raised when index of a sequence is out of range. |
| **KeyError** | Raised when a key is not found in a dictionary. |
| **KeyboardInterrupt** | Raised when the user hits interrupt key (Ctrl+c or delete). |
| **MemoryError** | Raised when an operation runs out of memory. |
| **NameError** | Raised when a variable is not found in local or global scope. |
| **NotImplementedError** | Raised by abstract methods. |
| **OSError** | Raised when system operation causes system related error. |
| **OverflowError** | Raised when result of an arithmetic operation is too large to be represented. |
| **ReferenceError** | Raised when a weak reference proxy is used to access a garbage collected referent. |
| **RuntimeError** | Raised when an error does not fall under any other category. |
| **StopIteration** | Raised by **next()** function to indicate that there is no further item to be returned by iterator. |
| **SyntaxError** | Raised by parser when syntax error is encountered. |
| **IndentationError** | Raised when there is incorrect indentation. |
| **TabError** | Raised when indentation consists of inconsistent tabs and spaces. |
| **SystemError** | Raised when interpreter detects internal error. |
| **SystemExit** | Raised by **sys.exit()** function. |
| **TypeError** | Raised when a function or operation is applied to an object of incorrect type. |
| **UnboundLocalError** | Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. |
| **UnicodeError** | Raised when a Unicode-related encoding or decoding error occurs. |
| **UnicodeEncodeError** | Raised when a Unicode-related error occurs during encoding. |
| **UnicodeDecodeError** | Raised when a Unicode-related error occurs during decoding. |
| **UnicodeTranslateError** | Raised when a Unicode-related error occurs during translating. |
| **ValueError** | Raised when a function gets argument of correct type but improper value. |
| **ZeroDivisionError** | Raised when second operand of division or modulo operation is zero. |

We can also define our own exception in Python (if required). Visit this page to learn more about [user-defined exceptions](https://www.programiz.com/python-programming/user-defined-exception).

#### 14.3 Exception Handling

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.

When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

**For example:**

if function **A** calls function **B** which in turn calls function **C** and an exception occurs in function **C**. If it is not handled in **C**, the exception passes to **B** and then to **A**.

If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

## 14.3.1 Catching Exceptions in Python

In Python, **exceptions can be handled using a** try statement.

A critical operation which can raise exception is placed inside the **try clause** and the code that handles exception is written in **except clause**.

It is up to us, what operations we perform once we have caught the exception.

**For example:**

# import module **sys** to get the type of exception

import sys

randomList = ['a', 0, 2]

for entry in randomList:

    try:

        print("The entry is", entry)

        r = 1/int(entry)

        break

    except:

        print("Oops!",sys.exc\_info()[0],"occured.")

        print("Next entry.")

        print()

print("The reciprocal of",entry,"is",r)

**OUTPUT:**

The entry is a

Oops! <class 'ValueError'> occured.

Next entry.

The entry is 0

Oops! <class 'ZeroDivisionError' > occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5

## 14.3.2 Catching Specific Exceptions in Python

In the above example, we did not mention any exception in the **except** clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch.

A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an except clause.

try:

  # do something

  pass

except ValueError:

  # handle ValueError exception

  pass

except (TypeError, ZeroDivisionError):

# handle multiple exceptions

# TypeError and ZeroDivisionError

  pass

except:

  # handle all other exceptions

  pass

## 14.3.3 Raising Exceptions

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword raise.

We can also optionally pass in value to the exception to clarify why that exception was raised.

>>> raise KeyboardInterrupt

Traceback (most recent call last): ... KeyboardInterrupt

>>> raise MemoryError("This is an argument")

Traceback (most recent call last): ...

MemoryError: This is an argument

>>> try:

  a = int(input("Enter a positive integer: "))

  if a <= 0: ...

  raise ValueError("That is not a positive number!")

  except ValueError as ve:

  print(ve)

Output:

Enter a positive integer: -2

That is not a positive number!

## 14.3.4 try...finally

The try statement in Python can have an optional **finally**clause. This clause is executed no matter what, and is generally used to release external resources.

**For example:**

we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).

try:

f = open("test.txt",encoding = 'utf-8')

finally:

f.close()

This type of construct makes sure the file is closed even if an exception occurs.

#### 15. PYTHON OBJECT ORIENTED PROGRAMMING

Python is a multi-paradigm programming language i.e., it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects, known as **Object-Oriented Programming (OOP).**

An object has two characteristics:

* attributes
* behavior

For Example:

Parrot is an object,

* name, age, color are attributes
* singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as **DRY (Don't Repeat Yourself).**

In Python, the concept of OOP follows some basic principles:

|  |  |
| --- | --- |
| **Inheritance** | A process of using details from a new class without modifying existing class. |
| **Encapsulation** | Hiding the private details of a class from other objects. |
| **Polymorphism** | A concept of using common operation in different ways for different data input. |

**15.1 Class**

**A class is a blueprint for the object.**

We can think of class as an sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The **example** for class of parrot can be :

class Parrot:

pass

Here, we use **class**keyword to define an empty class **Parrot**. From class, we construct instances. An instance is a specific object created from a particular class.

**15.2 Object**

**An object (instance) is an instantiation of a class.** When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The**example for object**of parrot class can be:

obj = Parrot()

Here, **obj** is object of class **Parrot**.

Example 1: Creating Class and Object in Python

class Parrot:

    # class attribute

    species = "bird"

    # instance attribute

    def \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

# instantiate the Parrot class

blu = Parrot("Blu", 10)

woo = Parrot("Woo", 15)

# access the class attributes

print("Blu is a {}".format(blu.\_\_class\_\_.species))

print("Woo is also a {}".format(woo.\_\_class\_\_.species))

# access the instance attributes

print("{} is {} years old".format( blu.name, blu.age))

print("{} is {} years old".format( woo.name, woo.age))

OUTPUT:

Blu is a bird

Woo is also a bird

Blu is 10 years old

Woo is 15 years old

**15.3 Methods**

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python

class Parrot:

    # instance attributes

    def \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

    # instance method

    def sing(self, song):

        return "{} sings {}".format(self.name, song)

    def dance(self):

        return "{} is now dancing".format(self.name)

# instantiate the object

blu = Parrot("Blu", 10)

# call our instance methods

print(blu.sing("'Happy'"))

print(blu.dance())

OUTPUT:

Blu sings 'Happy' Blu is now dancing

## 15.4 Inheritance

**Inheritance is a way of creating new class for using details of existing class without modifying it**. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

### Example 1: Use of Inheritance in Python

# parent class

class Bird:

    def \_\_init\_\_(self):

        print("Bird is ready")

    def whoisThis(self):

        print("Bird")

    def swim(self):

        print("Swim faster")

# child class

class Penguin(Bird):

    def \_\_init\_\_(self):

        # call super() function

        super().\_\_init\_\_()

        print("Penguin is ready")

    def whoisThis(self):

        print("Penguin")

    def run(self):

        print("Run faster")

peggy = Penguin()

peggy.whoisThis()

peggy.swim()

peggy.run()

OUTPUT:

Bird is ready Penguin is ready Penguin Swim faster Run faster

In the above program, we created two classes i.e. **Bird** (parent class) and **Penguin** (child class). The child class inherits the functions of parent class. We can see this from **swim()**method. Again, the child class modified the behavior of parent class. We can see this from **whoisThis()** method. Furthermore, we extend the functions of parent class, by creating a new **run()** method.

Additionally, we use **super()** function before**\_\_init\_\_()** method. This is because we want to pull the content of **\_\_init\_\_()** method from the parent class into the child class.

## 15.5 Encapsulation

Using OOP in Python, we can **restrict access to methods and variables**. This prevent data from direct modification which is called **Encapsulation**. In Python, we denote private attribute using underscore as prefix i.e single “ \_ “ or double “ \_\_“.

### Example 2: Data Encapsulation in Python

class Computer:

    def \_\_init\_\_(self):

        self.\_\_maxprice = 900

    def sell(self):

        print("Selling Price: {}".format(self.\_\_maxprice))

    def setMaxPrice(self, price):

        self.\_\_maxprice = price

c = Computer()

c.sell()

# change the price

c.\_\_maxprice = 1000

c.sell()

# using setter function

c.setMaxPrice(1000)

c.sell()

**OUTPUT**:

Selling Price: 900

Selling Price: 900

Selling Price: 1000

In the above program, we defined a class **Computer**. We use **\_\_init\_\_()** method to store the maximum selling price of computer. We tried to modify the price. However, we can’t change it because Python treats the **\_\_maxprice** as private attributes. To change the value, we used a setter function i.e **setMaxPrice()** which takes price as parameter.

## 15.6 Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

### Example 3: Using Polymorphism in Python

class Parrot:

    def fly(self):

        print("Parrot can fly")

    def swim(self):

        print("Parrot can't swim")

class Penguin:

    def fly(self):

        print("Penguin can't fly")

    def swim(self):

        print("Penguin can swim")

# common interface

def flying\_test(bird):

    bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

flying\_test(peggy)

OUTPUT:

Parrot can fly

Penguin can't fly

**15.7 Difference between a Method in Class and a Normal function in Python**

        In Methods, we need to **add an extra parameter** to the beginning of the parameter list.

        In Methods, the first parameter of parameter list refers to the object itself and we use word **self** for this parameter which is highly recommended by the Programmers.

        In Methods, Parameters after self are Optional.

**Syntax**:

**class** <classname>:

**def** <nameofmethod>(self,[parameters]):

**Ex**:

          # Creating a class with Methods

**class** person:

**def** \_\_init\_\_(self,name): # **\_\_init\_\_(self,[parameters])** is used to initialize the object

                       self.name = name

                       name = "John "

                       print(name)

**def** display(self):

                       print("Hello!!!! ",self.name)

          p1 = person("Python") # Creating a object called **p1**

          p1.display()   # Calling Method with object

          person("Cameroon").display()  # Calling method with calss

**Output:**

John

              Hello!!!! Python

              Hello!!!! Cameroon

**15.7.1 Use of self parameter in every method:**

**self** parameter differentiate between local variables and instance variables.

**Local Variable, Instance Variable and Class Variable:**

**1. Local Variable**:

The variable which does not belongs to an object is called local variable. In the above example, **name** is local variable.

**2. Instance Variable**:

The variable which is belongs to an object is called instance variable.

In the above example, **self.name** is a instance variable because we know that self refers to the object.

**3.** **Class Variable**:

The variable which is belong to all objects in the Class is called class variable.

**Syntax to access class variable**: <classname>.<variablename>

Example:

**class** Student:

        clg = 'svce'   # Class Variable

**def** \_\_init\_\_(self,rollno,name):

                   self.rollno = rollno     # self.rollno is Instance variable and rollno is Local Variable

                   self.name = name       # self.name is Instance variable and name is Local Variable

**def** display(self):

                    print("Student Name: ",self.name)

                    print("Student Roll No: ",self.rollno)

                    print("College: ",student.clg)

student1 = student('svce001', "Ram Charan")

student1.display()

student2 = student('svce002', "Komaram Bheem")

student2.display()

**Output**:

   Student Name: Ram Charan

   Student Roll no: svce001

   College: svce

   Student Name: Komaram bheem

   Student Roll no: svce002

   College: svce

**What if we don’t use self parameter in methods in class?**

**Code**:

**class** person:

**def** display(): # creating display method without self parameter

                 print("Hello!!!!")

       p=person()

       p.display()

**Output**:

        Traceback (most recent call last):

        p.display()

             TypeError: display() takes 0 positional arguments but 1 was given

       By using **@Staticmethod**we can avoid **self** parameter in methods in class

**Ex**:

**class** person:

**@Staticmethod**

**def** display(): # creating display method without self parameter

                 print("Hello!!!!")

       p=person()

       p.display()

**Output**:

     Hello!!!!

#### 16. PYTHON TIME AND CALENDAR MODULE:

**16.1 Time Module**:

       Using the methods which are belongs to Time module, we can get current date, time, day, year, month.

       To use those methods, we need to **import** **time** module.

**Methods to get current time**:

       >>> import time

       >>> time.time()    # To get Current time in seconds we use **time()** method

        1575561356.1108036

       >>> time.localtime(time.time())   # To get Current time in appropriate format we use **localtime(time.time())** method

       time.struct\_time(tm\_year=2019, tm\_mon=12, tm\_mday=5, tm\_hour=21, tm\_min=28, tm\_sec=40, tm\_wday=3, tm\_yday=339, tm\_isdst=0)

       >>> time.asctime()  # To get time in another format

       'Thu Dec  5 21:32:54 2019'

**Methods to get information about Upcoming days or Previous days**: For this use tuples.

       >>> tuple1 = (1997,12,25,7,30,59,0,0,0)

       >>> time.mktime(tuple1)

       883015259.0

       >>> time.localtime(time.mktime(tuple1))

       time.struct\_time(tm\_year=1997, tm\_mon=12, tm\_mday=25, tm\_hour=7, tm\_min=30, tm\_sec=59, tm\_wday=3, tm\_yday=359, tm\_isdst=0)

**Sleep() Method in time module**:

       Using **sleep()** method, we can **delay the execution** of the instruction.

**Ex**:

       import time

       time.sleep(10)     # Output will be displayed after 10 seconds

       print("Hello")

**Output:**

Hello      # Output will be displayed after 10 seconds because of sleep(10) method before print statement

**16.2 Calendar Module**:

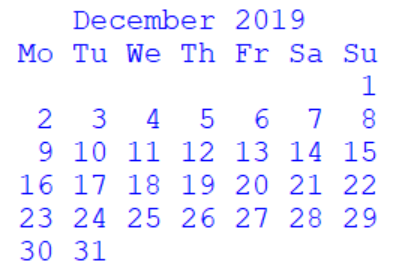
       By using Calendar Module Methods, we can print the calendar of a month or a whole year.

       To use those methods, we need to **import** **calendar** module.

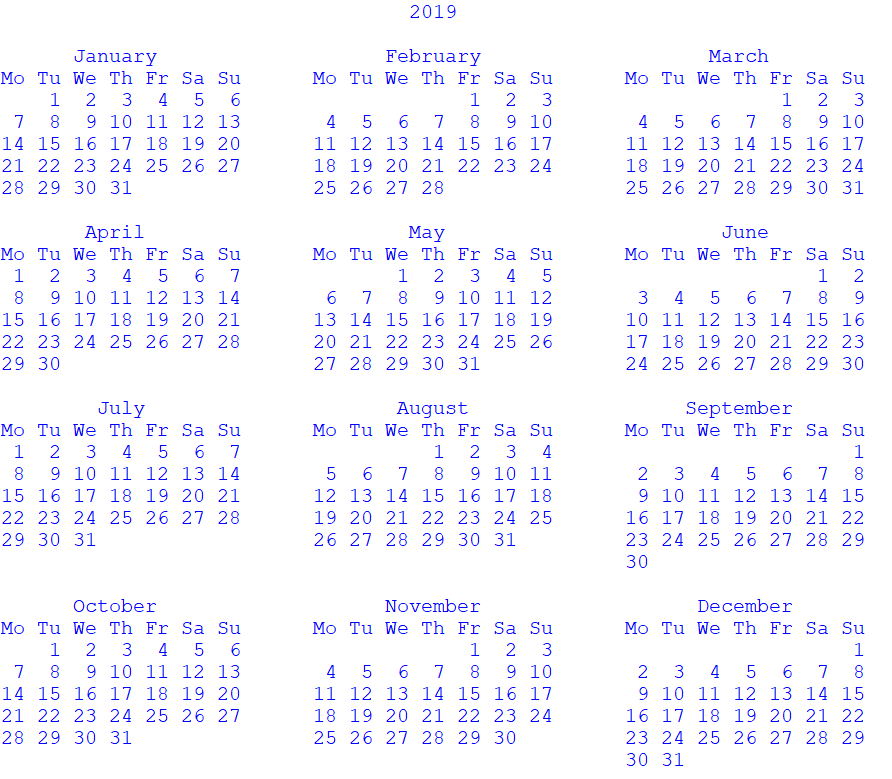
**Method to print the calendar of a month**:

        >>> import calendar

        >>> print(calendar.month(2019,12))   # To print calendar of a month we use **month(year,month,[width,length])** method of calendar module.



        >>> print(calendar.calendar(2019))  # To print calendar of a whole year we use **calendar(year,[width,length,distance b/w two months,no.of months per column])** method of calendar module.



**calendar.weekday(year, month, date)** method:

     Index of Monday = 0, Tuesday = 1, Wednesday = 2, Thursday = 3, Friday = 4, Saturday = 5, Sunday = 6

      >>> import calendar

      >>> calendar.weekday(2019,12,5)

      3

**calendar.isleap(year)**Method:

     By using **isleap(year)**, we can find out whether the year is Leap Year or Not.

     >>> calendar.isleap(2000)

     True

     >>> calendar.isleap(2019)

     False

**calendar.leapdays(year1,year2)**Method:

     By using **leapdays(year1,year2)**, We can find out number of leap years are present between year1 and year2.

     >>> calendar.leapdays(2000,2019)

     5

**To get more information about calendar module, use help(calendar) method.**

#### 17. PYTHON ITERATORS AND ITERABLLES:

An **Iterator** is an Object that contains a Countable Number of Values and iterator can be iterated upon all the Values.

       In Python, an iterator is an object which implements the iterator Protocol, which consist of the methods **\_\_iter()\_\_** and **\_\_next()\_\_** .

**Strings, Lists, Tuples, Dictionaries and sets** are all **Iterable** **Objects**. They are **Iterable Containers** which we can get an iterator from.

       Example:

       mystr = "HEAD"

       myit = iter(mystr)

       print(next(myit))

       print(next(myit))

       print(next(myit))

       print(next(myit))

**OUTPUT:**

H

E

A

D

**17.1 Looping through an Iterator**:

        The **for** loop actually creates an iterator object and executes the next method for each loop.

       Example:

       mytuple = ("rat","dog","kitten")

       for x in mytuple:

             print(x)

       OUTPUT:

       rat

       dog

       kitten

**How for loop actually works?**

**for element in iterable:**

is actually implemented as:

# create an iterator object from that iterable

**iter\_obj = iter(iterable)**

# infinite loop

**while true:**

**try**:

              # get next item

**element = next(iter\_obj)**

**except stopIteration**:

              # if stopIteration is raised, break from loop

**Break**

So internally, **for** loop creates an iterator object, **iter\_obj** by calling **iter()** on the iterable.

This **for** loop is actually an **infinite while loop**.

Inside the loop, it calls **next()** to get the next element and executes the body of **for** loop with this value. After all the items exhaust, **stopIteration** is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

**17.2 Create an Iterator**:

To Create an object/class as an iterator we have to implement the methods **\_\_iter()\_\_** and **\_\_next()\_\_**to the objects.

The **\_\_iter()\_\_**method allows to do operations but must always **return the iterator object** itself.

The **\_\_next()\_\_**method allows to do operations but must always **return the next item** in the sequence.

 Example:

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

class MyNumbers:  
  def \_\_iter\_\_(self):  
    self.a = 1  
    return self  
  
  def \_\_next\_\_(self):  
    x = self.a  
    self.a += 1  
    return x  
  
myclass = MyNumbers()  
myiter = iter(myclass)  
  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))

OUTPUT:

1

2

3

4

5

Example 2:

Create an Iterator that returns Top Ten numbers starting form 1

class TopTen:

   def  \_\_init\_\_(self):  
    self.num = 1

  def \_\_iter\_\_(self):

       return self  
  
  def \_\_next\_\_(self):

        if  self.num <= 10:

              value = self.num

              self.num += 1  
                         return value  
        else:

        raise stopIteration

values = TopTen()

print(next(values))    # **Prints 1 and doesn’t print 1 again in next for loop i.e., for loop will print values from 2**

for i in values:  
       print(i)

OUTPUT:

1

2

3

4

5

6

7

8

9

10

#### 18. PYTHON GENERATORS:

**Generators are used to fetch one value at a time from more number of values defined in a function using yield keyword.**

**18.1.** **Generator Function**:

A Generator Function is defined like a Normal Function, but whenever it needs to generate a value, it does so with **yield** Keyword rather than **return**.

If the body of a **def function contains yield**, the function automatically becomes a **generator function**.

Generator function generates Iterator.

**Example**:

# A generator function that yields 1 for first time, 2 second time and 3 third time

**def** simpleGeneratorFun():

        yield 1

        yield 2

        yield 3

# Driver code to check above generator function

for value in simpleGeneratorFun():

        print(value)

OUTPUT:

1

2

3

**18.2.** **Generator Object**:

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in **for** loop.

# A Python program to demonstrate use of generator object with next()

# A generator function

**def** simpleGeneratorFun():

        yield 1

        yield 2

        yield 3

# x is a generator object

x = simpleGeneratorFun()

# Iterating over the generator object using next

print(x.\_\_next\_\_()); # In Python 2, next()

print(x.\_\_next\_\_());

print(x.\_\_next\_\_());

OUTPUT:

1

2

3

**18.3 Python Recursive Function**:

If a function calls itself, it is called as **Recursive Function**.

Example:

A Program using Recursive Function to find the Factorial of a Number

def factorial(num):

      if num == 1:

          return 1

      else:

          return num \* **factorial(num -1)** # Here function is calling itself which is called as **Recursive function**

num = 4

print( factorial(num))

OUTPUT:

24

Advantages of Recursive Function:

1. Recursive Functions make the code look clean and simple.

2. A complex task can be broken down into simpler sub-problems using Recursion.

Dis-Advantages of Recursive Function:

1. Some times logic behind recursion is hard to follow through.

2. Recursive calls are more expensive as they take up lot of memory and time.

3. Recursive Functions are hard to debug.

#### 19. PYTHON REGEX:

* A Regular Expression (RegEx) is a sequence of characters that defines a search pattern i.e., RegEx is a tool for matching patterns in text.

Example:

**^a…s$**

The above code defines a RegEx Pattern. The Pattern is : **any five letter word starting with** a **and ending with** s.

* Using RegEx, We can **match**, **find** or **replace** text or word in place in strings.
* Python has a **module named re** to work with RegEx.

Example:

import re

pattern = '^a...s$'

test\_str = input("Enter string to search: ")

if **re.match(pattern,test\_str)**:

print("Search Successful")

else:

print("Search Unsuccessful")

OUTPUT 1:

Enter string to search: alias

Search Successful

OUTPUT 2:

Enter string to search: aliaa

Search Unsuccessful

* **Metacharacters** are used to specify Regular Expressions. Metacharacters are characters that are interpreted in a special way by a RegEx Engine. Here’s a list of metacharacters

**[]** **.** **^** **$** **\*** **+** **?** **{}** **()** **\** **|**

**[] – Square Brackets**

* **Square Brackets specifies a set of characters that we wish to match.**
* **[abc] will match if the string we are trying to match will contain any of the a, b or c.**
* **We can also specify a range of characters using – inside the square brackets.**

1. **[a-e] is same as [abcde]**
2. **[1-4] is same as [1234]**
3. **[0-39] is same as [01239]**

* **We can complement (invert) the character set by using caret ^ symbol at the start of the square bracket.**

1. **[^abc] means any character except a or b or c.**
2. **[^0-9] means any Non-Digit Character.**

**. – Period**

**A Period matches any single Character (it can be any character except newline ‘\n’).**

**^ - Caret**

**The Caret symbol ^ is used to check if a string starts with a certain character.**

**$ - Dollar**

**The Dollar symbol $ is used to check if a string ends with a certain character.**

**\* - Star**

**The Star symbol \* matches Zero or More occurrences of the pattern left(Preceding Expression) to it.**

**Example: 'X\*' = 0 or X or XX or XXX or XXXX etc**

**+ - Plus**

**The Plus symbol + matches One or More occurrences of the pattern left(Preceding Expression) to it.**

**Example: 'X+' = X or XX or XXX or XXXX etc**

**? – Question Mark**

**The Question Mark symbol ? matches Zero or One occurrences of the pattern left(Preceding Expression) to it.**

**Example: 'X?' = 0 or X**

**{} – Braces**

**Consider this code : {m,n}. This means at least m, and at most n repetitions of the pattern left to it.**

**Example:**

**'X{4}' = Matches exactly 4X i.e., XXXX**

**a{2,3} --- Matches at least 2 a’s but not more than 3 a’s i.e., aa, aaa**

**abc daat --- 1 match ( at daat)**

**aabc daaaat --- 2 matches ( at aabc daaaat)**

**[0-9]{2,4} --- Matches at least 2 digits but not more than 4 digits**

**12 and 345673 --- 2 matches ( 12 and 345673)**

**1 and 2 ---- No match**

**| - Alternation**

**Vertical bar | is used for Alternation ( or Operator)**

**a | b match any string that contains either a or b.**

**() – Group**

**Parenthesis () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz.**

**\ - Backslash**

**Backslash \ is used to escape various characters including all metacharacters.**

**Example:**

**\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.**

**\s – Matches the Whitespace**

**\S – Matches the Non- Whitespace**

**\d – Matches the single digit**

**\D – Matches the single Non-Digit**

**Functions and Constants to work with RegEx:**

**Python has a module named re to work with RegEx. To use it, we need to import the module**

**import re**

1. **re.findall(): This method returns a list of strings that containing all matches. If the pattern is not found, re.findall() will return empty list.**

**Example: Program to extract numbers from a string**

**import re**

**string = 'Hello 67 Hi 34 Good 45'**

**pattern = '\d+' # \d – matches any decimal digit equivalent to [0-9]**

**# + – matches one or more occurrences of the pattern left to it.**

**result = re.findall(pattern,string)**

**print(result)**

**OUTPUT:**

**['67', '34', '45']**

1. **re.split(): This method splits the string where there is a match and returns a list of strings where the splits have occurred.**

**Example:**

**import re**

**string = 'Twelve : 12 Eighty Nine : 89.'**

**pattern = '\d+' # \d – matches any decimal digit equivalent to [0-9]**

**# + – matches one or more occurrences of the pattern left to it.**

**result = re.split(pattern,string) # returns a list of strings where the splits have occurred**

**print(result)**

**OUTPUT:**

**['Twelve : ', 'Eighty Nine : ', '. ']**

**If the pattern is not found, re.split() returns a list containing an empty string.**

**We can pass maxsplit argument to the re.split() method. Its’s the maximum number of splits that will occur.**

**The default value of maxsplit is 0; meaning all possible splits.**

**Example:**

**import re**

**string = 'Twelve : 12 Eighty Nine : 89.'**

**pattern = '\d+' # \d – matches any decimal digit equivalent to [0-9]**

**# + – matches one or more occurrences of the pattern left to it.**

**# maxsplit = 1 i.e., split only at first occurrence**

**result = re.split(pattern,string,1) # returns a list of strings where the splits have occurred**

**print(result)**

**OUTPUT:**

**['Twelve : ', 'Eighty Nine : 89. ']**

1. **re.sub():**

**Syntax: re.sub(pattern, replace, string)**

**This method returns a string where matched occurrences are replaced with content of the replace variable. If the pattern is not found, re.sub() will return original string.**

**Example:**

**import re**

**string = 'abc 12\**

**def 23 \n f45 6as'**

**pattern = '\s+' # matches one or more occurrences of the whitespace in the string**

**replace = ' ' # Empty String**

**new\_string = re.sub(pattern,replace,string)**

**print(new\_string)**

**OUTPUT:**

**abc12def23f456as**

**We can pass count as fourth parameter to the re.sub() method.**

**re.subn(): This method returns a tuple of 2 items containing the new string and number of substitutions made.**

**Example:**

**import re**

**string = 'abc 12\**

**def 23 \n f45 6as'**

**pattern = '\s+' # matches one or more occurrences of the whitespace in the string**

**replace = ' ' # Empty String**

**new\_string = re.subn(pattern,replace,string)**

**print(new\_string)**

**OUTPUT:**

**('abc12def23f456as', 4)**

1. **re.search(): This method takes 2 arguments: a Pattern and a String. This method looks for the first location where the RegEx pattern produces a match with the string.**

**This search function will search entire string and returns the first occurrence of the pattern.**

**If the search is successful, re.search() returns a match object; if not, it returns None.**

**Syntax: match = re.search(pattern,string,[flag=0])**

**Example:**

**import re**

**string = "Python is fun"**

**# \APython check if 'Python' is at the beginning**

**match = re.search('\APython', string)**

**if match:**

**print("Pattern found inside the string")**

**else:**

**print("Pattern not found")**

**OUTPUT:**

**Pattern found inside the string**

1. **re.match(): This function is used to match any text or word in the strings.**

**This match function will search only at the Beginning of the string.**

**Syntax: match = re.match(pattern,string,[flag=0])**

**Example:**

**import re**

**string = "Pet:Cat I Hate Cats"**

**match = re.match("pet:\w\w\w",string) # \w matches any Alpha-Numeric Character equivalent to [a-zA-Z0-9\_]. Underscore \_ is also considered as alphanumeric character.**

**print(match)**

**print(match.group(0)) # The group() method returns the part of the string where there is a match.**

**OUTPUT:**

**<re.Match object; span=(0, 7), match='pet:cat'>**

**'Pet:Cat'**

**match.start()** returns the **index of the start** of the matched sub-string.

**match.end()** returns the **end index** of the matched sub-string.

**match.span()** returns a tuple containing **start and end index** of the matched part.

**match.re** attribute of a matched object returns a **regular expression object**.

**match.string** attribute returns the **passed String**.

**Using r prefix before RegEx**:

When **r** or **R** **prefix** is used before a regular expression, it means **Raw String**.

Example:

**'\n' is a new line whereas r'\n' means 2 characters : a backslash \ followed by n**

**Example:**

**import re**

**string = "\n and \r are escape sequences"**

**result = re.findall(r'[\n\r]',string)**

**print(result)**

**OUTPUT:**

**['\n', '\r']**

**20. Instance vs. Static vs. Class Methods in Python: The Important Differences**

There are 3 types of Methods in Python:

1. Instance Methods
2. Static Methods
3. Class Methods

**Understanding Decorator Patterns (Decorators):**

Decorator Patterns, or simply called as Decorators, are Functions that we can write them ourselves, or use those included in libraries, or the Python standard library.

Like any function, Decorators perform a task. The difference here is that **Decorators apply Logic or change the behaviour of other functions**.

Decorators are an excellent way to reuse code and can help to separate logic into individual concerns.

The Decorators Pattern is Python’s preferred way of defining static or class methods.

Decorators have to immediately precede a function or class declaration.

Decorators start with **@** sign, and unlike normal methods, we don’t have to put parenthesis on the end.

Example:

**class** DecoratorExample:  
 """ Example Class """  
 **def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)  
  
 @staticmethod  
 **def** example\_function():  
 """ This method is decorated! """  
 print(**'I\'m a decorated function!'**)

de = DecoratorExample()  
de.example\_function()

**20.1. Instance Methods:**

Instance Methods are most common type of methods in Python Classes because they can access Unique data of their instance.

If we have two objects each created from a Car class, then each object may have different properties like different colours, engine sizes, seats, and so on.

Instance Methods must have **self** as a parameter, but no need to pass this in every time. **self** is another Python special term.

We can use **self** to access any data or methods that may reside in our class. We won’t able to be access them without going through **self**.

Finally, instance methods are the most common, there’s is **NO Decorator needed**. Any method that we create will automatically created as an instance Method.

Example 1:

**class** DecoratorExample:  
 """ Example Class """  
  
 **def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)  
 self.name = **'Decorator\_Example'  
  
 def** example\_function(self):  
 """ This method is an instance method! """  
 print(**'I\'m an instance method!'**)  
 print(**'My name is '** + self.name)  
  
de = DecoratorExample()  
de.example\_function()

OUTPUT:

Hello, World!

I'm an instance method!

My name is Decorator\_Example

**Note**: The **name** variable is accessed through **self**. Notice that when **example\_function** is called, we don’t have to pass **self** as an argument– Python does for us.

Example 2:

**class** DecoratorExample:  
 """ Example Class """  
  
 **def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)  
  
 **def** example\_function(self,name):  
 """ This method is an instance method! """  
 print(**'I\'m an instance method! with 2 parameters'**)  
 self.name = name  
 print(**'My name is '** + self.name)  
  
de = DecoratorExample()  
de.example\_function(**"Decorator"**)

OUTPUT:

Hello, World!

I'm an instance method! with 2 parameters

My name is Decorator

**Note**: When **example\_function** is called, we passed only **name = "Decorator"** as an argument and we don’t have to pass **self** – Python does for us.

**20.2. Static Methods:**

Static Methods are methods that are related to a class in some way, but no need to access any class-specific data and even no need to instantiate an instance, we can simply call our method.

The **@staticmethod** decorator was used to tell Python that this method is a static method.

We may use a static method to add two numbers, or print a given string.

Example:

**class** DecoratorExample:  
 """ Example Class """

**def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)

@staticmethod  
 **def** example\_function():  
 """ This method is a static method! """  
 print(**'I\'m a static method!'**)  
  
de = DecoratorExample.example\_function()

OUTPUT:

I'm a static method!

**20.3. Class Methods**:

Class Methods know about their class. They can’t access specific instance data, but they can call other static methods.

Class Methods do not need **self** as an argument, but they do need a parameter called **cls**. This stands for **class**, and like self, gets automatically passed in Python.

Class Methods are created using **@classmethod** decorator.

Class Methods can manipulate the class itself, which is useful when we are working larger, more complex projects.

Example:

**class** DecoratorExample:  
 """ Example Class """  
  
 **def** \_\_init\_\_(self):  
 """ Example Setup """  
 print(**'Hello, World!'**)  
  
 @classmethod  
 **def** example\_function(cls):  
 """ This method is a class method! """  
 print(**'I\'m a class method!'**)  
 cls.some\_other\_function()  
  
 @staticmethod  
 **def** some\_other\_function():  
 print(**'Hello!'**)  
  
de = DecoratorExample()  
de.example\_function()

OUTPUT:

Hello, World!

I'm a class method!

Hello!

**Summary**:

* **Instance Methods**: The most common method type. Able to access data and properties unique to each instance.
* **Static Methods**: Cannot access anything else in the class. Totally self-contained code.
* **Class Methods**: Can access limited methods in the class. Can modify class specific details.