Contents

[Installation 4](#_Toc73543198)

[Configure and run the debugger 4](#_Toc73543199)

[Install and use packages# 5](#_Toc73543200)

[Python REPL 7](#_Toc73543201)

[Python Identifiers and Keywords 9](#_Toc73543202)

[Keywords 9](#_Toc73543203)

[Identifiers 11](#_Toc73543204)

[Python Simple and Compound Statements 15](#_Toc73543205)

[Simple Statements 15](#_Toc73543206)

[Compound Statements 15](#_Toc73543207)

[Python Values, Types and Variables 18](#_Toc73543208)

[Values and types 18](#_Toc73543209)

[Python Built-in Data Types 19](#_Toc73543210)

[Getting the Data Type 19](#_Toc73543211)

[Setting the Data Type 19](#_Toc73543212)

[Python Statements 21](#_Toc73543213)

[Multi-line statement 21](#_Toc73543214)

[Python : Indentation 22](#_Toc73543215)

[Python Operators and Expressions 23](#_Toc73543216)

[Arithmetic Operators 23](#_Toc73543217)

[Relational Operators 23](#_Toc73543218)

[Bitwise Operators 24](#_Toc73543219)

[Assignment operators 25](#_Toc73543220)

[Identity operators 25](#_Toc73543221)

[Membership Operators 25](#_Toc73543222)

[Expressions 26](#_Toc73543223)

[Order of operations 26](#_Toc73543224)

[String operations: 26](#_Toc73543225)

[Take Input from User in Python 27](#_Toc73543226)

[Take an integer as an Input 27](#_Toc73543227)

[Python Comments 28](#_Toc73543228)

[Multi-line comments 28](#_Toc73543229)

[Docstring in Python 28](#_Toc73543230)

[Declaration of docstrings 28](#_Toc73543231)

[Python Conditional Execution 30](#_Toc73543232)

[if Statement 30](#_Toc73543233)

[Alternative execution: if-else Statement 30](#_Toc73543234)

[Chained conditionals: if…elif…else 30](#_Toc73543235)

[Nested conditionals 30](#_Toc73543236)

[Catching exceptions using try and except 31](#_Toc73543237)

[Python Functions 32](#_Toc73543238)

[Define function and function call 32](#_Toc73543239)

[Parameters and arguments 32](#_Toc73543240)

[Built-in functions in Python 32](#_Toc73543241)

[Type conversion functions 33](#_Toc73543242)

[Random numbers 33](#_Toc73543243)

[randint() Function in Python 33](#_Toc73543244)

[choice() Function in Python 33](#_Toc73543245)

[Math functions in Python 33](#_Toc73543246)

[Python Loops 34](#_Toc73543247)

[Python While Loop 34](#_Toc73543248)

[Python “Infinite loops” and break 34](#_Toc73543249)

[Finishing iterations with continue in Python 35](#_Toc73543250)

[Python for Loop 35](#_Toc73543251)

[The range() function in Python 36](#_Toc73543252)

[Bonus Example: Counting and summing loops 36](#_Toc73543253)

[Bonus Example: Maximum and minimum loops 36](#_Toc73543254)

[Python Strings 37](#_Toc73543255)

[Getting the length of a string using len() function 37](#_Toc73543256)

[Traversing a  string with a loop 37](#_Toc73543257)

[String slices 37](#_Toc73543258)

[Strings are immutable: 37](#_Toc73543259)

[Looping and counting 38](#_Toc73543260)

[The in operator in Python 38](#_Toc73543261)

[String comparison 38](#_Toc73543262)

[Python Lists 39](#_Toc73543263)

[How to create a list? 39](#_Toc73543264)

[Lists are mutable 39](#_Toc73543265)

[How to access elements from a list? 39](#_Toc73543266)

[List Index 40](#_Toc73543267)

[Negative indexing 40](#_Toc73543268)

[Traversing a list 40](#_Toc73543269)

[List operations 40](#_Toc73543270)

[How to slice lists in Python? 40](#_Toc73543271)

[List methods 41](#_Toc73543272)

[How to delete or remove elements from a list? 41](#_Toc73543273)

[***pop*** *operator* 41](#_Toc73543274)

[***del*** operator 41](#_Toc73543275)

[remove() Function 42](#_Toc73543276)

[Lists and functions 42](#_Toc73543277)

[Lists and strings 43](#_Toc73543278)

[List arguments 43](#_Toc73543279)

[Python Dictionaries 45](#_Toc73543280)

[How to create a dictionary? 45](#_Toc73543281)

[How to access elements from a dictionary? 45](#_Toc73543282)

[How to change or add elements in a dictionary? 46](#_Toc73543283)

[Dictionary Membership Test 46](#_Toc73543284)

[How to delete or remove elements from a dictionary? 46](#_Toc73543285)

[Python Dictionary Methods 47](#_Toc73543286)

[Referecnces 49](#_Toc73543287)

# Installation

* Install VS Code (<https://code.visualstudio.com/>)
* Install Python (<https://www.python.org/downloads>)
  + abcd
* Install Python Extension for VS Code
  + Open VS Code and install the extension from:

<https://marketplace.visualstudio.com/items?itemName=ms-python.python>



* + Once you have a version of Python installed, activate it using the **Python: Select Interpreter** command.
  + If VS Code doesn't automatically locate the interpreter you're looking for, refer to [Environments - Manually specify an interpreter](https://code.visualstudio.com/docs/python/environments#_manually-specify-an-interpreter).
  + You can configure the Python extension through settings. Learn more in the [Python Settings reference](https://code.visualstudio.com/docs/python/settings-reference).
* Verify Python installation
  + python –version
  + py –version
* From the terminal window, create a folder and navigate to that folder
* Open that folder in VS Code
* Create a file named hello.py and add the following code to it and save the file:

msg = "Hello World"

print(msg)

* The Python extension then provides shortcuts to run Python code in the currently selected interpreter (**Python: Select Interpreter** in the Command Palette):
  + In the text editor: right-click anywhere in the editor and select **Run Python File in Terminal**. If invoked on a selection, only that selection is run.
  + In Explorer: right-click a Python file and select **Run Python File in Terminal**.

## Configure and run the debugger

First, set a breakpoint on line 2 of hello.py by placing the cursor on the print call and pressing F9. Alternately, just click in the editor's left gutter, next to the line numbers. When you set a breakpoint, a red circle appears in the gutter.

Next, to initialize the debugger, press F5. Since this is your first time debugging this file, a configuration menu will open from the Command Palette allowing you to select the type of debug configuration you would like for the opened file.

Just select **Python File**, which is the configuration that runs the current file shown in the editor using the currently selected Python interpreter.

**Note**: VS Code uses JSON files for all of its various configurations; launch.json is the standard name for a file containing debugging configurations.

Select the Debug menu from the left panel, from the dropdown at the top, select Add Configuration. This will create a launch,json in the .vscode folder. Once this file is created, you will not be required to select the Python from the dropdown to run the code.

You can also work with variables in the **Debug Console** (*If you don't see it, select****Debug Console****in the lower right area of VS Code, or select it from the****...****menu*). Then try entering the following lines, one by one, at the **>** prompt at the bottom of the console:

msg

msg.capitalize()

msg.split()

## Install and use packages[#](https://code.visualstudio.com/docs/python/python-tutorial#_install-and-use-packages)

Let's now run an example that's a little more interesting. In Python, packages are how you obtain any number of useful code libraries, typically from [PyPI](https://pypi.org/). For this example, you use the matplotlib and numpy packages to create a graphical plot as is commonly done with data science. (Note that matplotlib cannot show graphs when running in the [Windows Subsystem for Linux](https://docs.microsoft.com/windows/wsl/about) as it lacks the necessary UI support.)

Return to the **Explorer** view (the top-most icon on the left side, which shows files), create a new file called standardplot.py, and paste in the following source code:

import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(0, 20, 100) # Create a list of evenly-spaced numbers over the range

plt.plot(x, np.sin(x)) # Plot the sine of each x point

plt.show() # Display the plot

Next, try running the file in the debugger using the "Python: Current file" configuration as described in the last section.

Unless you're using an Anaconda distribution or have previously installed the matplotlib package, you should see the message, **"ModuleNotFoundError: No module named 'matplotlib'"**. Such a message indicates that the required package isn't available in your system.

To install the matplotlib package (which also installs numpy as a dependency), stop the debugger and use the Command Palette to run **Terminal: Create New Integrated Terminal** (Ctrl+Shift+`). This command opens a command prompt for your selected interpreter.

A best practice among Python developers is to avoid installing packages into a global interpreter environment. You instead use a project-specific virtual environment that contains a copy of a global interpreter. Once you activate that environment, any packages you then install are isolated from other environments. Such isolation reduces many complications that can arise from conflicting package versions. To create a virtual environment and install the required packages, enter the following commands as appropriate for your operating system:

1. Create and activate the virtual environment

**Note**: When you create a new virtual environment, you should be prompted by VS Code to set it as the default for your workspace folder. If selected, the environment will automatically be activated when you open a new terminal.



**For Windows**

py -3 -m venv .venv

.venv\scripts\activate

If the activate command generates the message "Activate.ps1 is not digitally signed. You cannot run this script on the current system.", then you need to temporarily change the PowerShell execution policy to allow scripts to run (see [About Execution Policies](https://go.microsoft.com/fwlink/?LinkID=135170) in the PowerShell documentation):

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process

**For macOS/Linux**

python3 -m venv .venv

source .venv/bin/activate

1. Select your new environment by using the **Python: Select Interpreter** command from the **Command Palette**.
2. Install the packages

# Don't use with Anaconda distributions because they include matplotlib already.

# macOS

python3 -m pip install matplotlib

# Windows (may require elevation)

python -m pip install matplotlib

# Linux (Debian)

apt-get install python3-tk

python3 -m pip install matplotlib

1. Rerun the program now (with or without the debugger) and after a few moments a plot window appears with the output:



1. Once you are finished, type deactivate in the terminal window to deactivate the virtual environment.

## Python REPL

REPL = Red-Evaluate-Print Loop

The process is:

1. **Read:** take user input.
2. **Eval:** evaluate the input.
3. **Print:** shows the output to the user.
4. **Loop:** repeat.

REPL is an interactive read-evaluate-print loop (REPL) window for each of your Python environments, which improves upon the REPL you get with python.exe on the command line.

Just open a command prompt and run “python” to open the REPL interactive environment:



In the window, start entering python code, which will be executed one line at a time, like an interpreter:



Some more examples:

>>> "hello world"  
'hello world'  
>>>

>>> 128 / 8  
16.0  
>>> 256 \* 4  
1024  
>>>

>>> width = 10  
>>> height = 20  
>>> size = width\*height  
>>> print(size)  
200  
>>>

To quit:

>>> exit()

# Python Identifiers and Keywords

## Keywords

* The keywords are some predefined and reserved words in python that have special meaning. Keywords are used to define the syntax of the coding.
* The keyword cannot be used as an identifier, function, and variable name.
* All the keywords in python are written in lower case expect True and False.
* There are 33 keywords in Python 3.7, let’s go through all of them one by one.

| **No.** | **Keywords** | **Description** |
| --- | --- | --- |
| 1 | **and** | This is a logical operator it returns true if both the operands are true else return false. |
| 2 | **Or** | This is also a logical operator it returns true if anyone operand is true else return false. |
| 3 | **not** | This is again a logical operator it returns True if the operand is false else return false. |
| 4 | **if** | This is used to make a conditional statement. |
| 5 | **elif** | Elif is a condition statement used with if statement the elif statement is executed if the previous conditions were not true |
| 6 | **else** | Else is used with if and elif conditional statement the else block is executed if the given condition is not true. |
| 7 | **for** | This is created for a loop. |
| 8 | **while** | This keyword is used to create a while loop. |
| 9 | **break** | This is used to terminate the loop. |
| 10 | **as** | This is used to create an alternative. |
| 11 | **def** | It helps us to define functions. |
| 12 | **lambda** | It used to define the anonymous function. |
| 13 | **pass** | This is a null statement that means it will do nothing. |
| 14 | **return** | It will return a value and exit the function. |
| 15 | **True** | This is a boolean value. |
| 16 | **False** | This is also a boolean value. |
| 17 | **try** | It makes a try-except statement. |
| 18 | **with** | The with keyword is used to simplify exception handling. |
| 19 | **assert** | This function is used for debugging purposes. Usually used to check the correctness of code |
| 20 | **class** | It helps us to define a class. |
| 21 | **continue** | It continues to the next iteration of a loop |
| 22 | **del** | It deletes a reference to an object. |
| 23 | **except** | Used with exceptions, what to do when an exception occurs |
| 24 | **finally** | Finally is use with exceptions, a block of code that will be executed no matter if there is an exception or not. |
| 25 | **from** | The form is used to import specific parts of any module. |
| 26 | **global** | This declares a global variable. |
| 27 | **import** | This is used to import a module. |
| 28 | **in** | It’s used to check if a value is present in a list, tuple, etc, or not. |
| 29 | **is** | This is used to check if the two variables are equal or not. |
| 30 | **None** | This is a special constant used to denote a null value or avoid. It’s important to remember, 0, any empty container(e.g empty list) do not compute to None |
| 31 | **nonlocal** | It’s declared a non-local variable. |
| 32 | **raise** | This raises an exception |
| 33 | **yield** | It’s ends a function and returns a generator. |

## Identifiers

An identifier is a name used to identify a variable, function, class, module, etc. The identifier is a combination of character digits and underscore. The identifier should start with a character or Underscore then use digit. The characters are A-Z or a-z,a UnderScore ( \_ ) and digit (0-9). we should not use special characters ( #, @, $, %, ! ) in identifiers.

**Examples of valid identifiers:**

var1

\_var1

\_1\_var

var\_1

**Examples of invalid identifiers:**

!var1

1var

1\_var

var#1

**Example of and, or, not, True, False keywords:**

print("example of True, False, and, or not keywords")

#  compare two operands using and operator

print(True and True)

# compare two operands using or operator

print(True or False)

# use of not operator

print(not False)

**Example of a break, continue.**

# execute for loop

for i in range(1, 11):

    # print the value of i

    print(i)

    # check the value of i is less then 5

    # if i lessthen 5 then continue loop

    if i < 5:

        continue

    # if i greather then 5 then break loop

    else:

        break

**Example of for, in, if, elif and else keyword:**

# run for loop

for t in range(1, 5):

  # print one of t ==1

    if t == 1:

        print('One')

   # print two if t ==2

    elif t == 2:

        print('Two')

    else:

        print('else block execute')

**Example of def, if and else keywords:**

# define GFG() function using def keyword

def GFG():

    i=20

    # check i is odd or not

    # using if and else keyword

    if(i % 2 == 0):

        print("given number is even")

    else:

        print("given number is odd")

# call GFG() function

GFG()

**Example try, except, raise:**

def fun(num):

    try:

        r = 1/num

    except:

        print('Exception raised.')

        return

    return r

print(fun(10))

print(fun(0))

**Example of a lambda keyword:**

# define a anonymous using lambda keyword

# this labda function increment the value of b

a = lambda b: b+1

# run a for loop

for i in range(1, 6):

    print(a(i))

**Use of return keyword:**

# define a function

def fun():

  # declare a variable

    a = 5

    # return the value of a

    return a

# call fun method and store

# it's return value in a variable

t = fun()

# print the value of t

print(t)

**Use of a del keyword:**

# create a list

l = ['a', 'b', 'c', 'd', 'e']

# print list before using del keyword

print(l)

del l[2]

# print list after using del keyword

print(l)

**Use of global keyword:**

# declare a variable

gvar = 10

# create a function

def fun1():

  # print the value of gvar

    print(gvar)

# declare fun2()

def fun2():

  # declare global value gvar

    global gvar

    gvar = 100

# call fun1()

fun1()

# call fun2()

fun2()

**Example of yield keyword:**

def Generator():

    for i in range(6):

        yield i+1

t = Generator()

for i in t:

    print(i)

**Example of assert keyword:**

def sumOfMoney(money):

    assert len(money) != 0,"List is empty."

    return sum(money)

money = []

print("sum of money:",sumOfMoney(money))

# Python Simple and Compound Statements

We write code blocks in Python and each code block contains sequence of statements. We classified these statements as simple and compound statements. Python program contains collection of these statements; assignments, expressions, computations, functions, loops etc.

## Simple Statements

The statements which are meant for simple operations and mostly written in a single logical line of code.

**For example**, assignment statements are simple statements.

x = 10

which means, we are assigning a value “10” to the variable “x”. This we call as simple statement.

The computation statements (expression statements) also we call simple statements; these statements will compute or calculate some expressions and return the results.

**For example**, x = (10 + 15) is an expression statement.

Other than Assignment and Expression statements; the statements below also we called as Simple Statements: These are the statements formed with Python keyword(s); some of them are break, continue, return and import.

* **break** Statement – We use ***break*** statement, to bypass the execution of the statements which are defined after the break statement. The execution control will go to end of the Compound Statement. Usually we use this statement, within the Compound Statements.
* **continue** statement – **continue** statement is used to skip the statements execution which are defined after this statement. The execution control will go to the beginning of the Compound Statement. These statements also usually use with the Compound Statements.
* Have you noticed the difference between break & continue statements? Control execution will go to the beginning of the Compound Statement when we use continue; where as for break, the control execution will go to end of the Compound Statement.
* **return** statement -We use **return** statements within the function to return from the function with or without a value.
* **import** statement – To import code modules to current namespace, we use **import** statement. Usually, we write these statements at the beginning of the Program code.

## Compound Statements

A compound statement is a statement comprise of group of statements. The compound statements are usually executed, when a condition satisfies or a code block is called directly or through a function call. Compound Statements are spread into multiple logical lines; but aligned them into a particular group.

Class definitions and Function definitions are Compound Statements

Other Compound Statements are:

* The conditional statement – The if statement
* The statements which are grouped with in the Conditional Compound Statement (**The if statement**) are going to execute when the particular condition is satisfied.
* Condition Loop Statements – The for statement AND the while statement
* **for** statement is used to iterate through the elements of a sequence; whereas the statements within the **while** statement are going to execute when the condition is satisfied.
* Using **while** statement also we can iterate through the elements of a sequence; but we need to write additional code to do this; whereas **for**statement syntax by default supports this.
* An Exception Handler – The try statement
* The group of statements with-in **try** are block are going to execute when an exception occurs.

Putting all together the statements; the complete code looks like below:

|  |
| --- |
| #stmts\_example.py |
|  |
| # import statement |
| import math |
|  |
| x = 100 |
| index = 1 |
|  |
| # Display PI value |
| print("PI Value:\n", math.pi) |
|  |
| # conditional statement - The if statement |
| if ( x == 100 ): |
| x = x / 4 |
| print("\nThe result of (100/4) is:\n", x) |
|  |
|  |
| # The for statement |
| print("\n-- The for statement --\n") |
| print("Elements in the sequence are:") |
| sequence = [1, 2, 3, 4, 5] |
| for element in sequence: |
| print(element) |
|  |
|  |
| # The while statement |
| print("\n-- The while statement --\n") |
| print("Print only EVEN numbers:") |
| while(index < x): |
| if( ( index % 2 ) == 0 ): |
| print(index) |
|  |
| index = index + 1 |
|  |
| # The break & continue statements |
| print("\n-- The break & continue statements --\n") |
| print("Enter any value (0 - exit):") |
| while(1): |
| n = int(input()) |
| if ( n == 0 ): |
| break |
|  |
| # skip EVEN numbers to print |
| if( ( n % 2 ) == 0 ): |
| continue |
|  |
| print("You ENTERED the NUMBER : ", n) |
|  |
|  |
| # The try statement |
| print("-- The try statement --") |
| try: |
| div\_by\_0 = (1 / 0) |
| except: |
| print("Hurray!!! we caught, Divide / 0 Error") |

# Python Values, Types and Variables

## Values and types

* A value is one of the most basic things in any program works with.
* A value may be characters i.e., ‘Hello, World!’ or a number like 1,2.2 ,3.5 etc.
* Values belong to different types: 1 is an integer, 2 is a float and ‘Hello, World!’ is a string etc.

**Numbers:**

Python supports 3 types of numbers: integers, float and complex number. If you want to know  what type a value has you can use type() function. Paste the following code and click the run button to check the output.

print(type(1))  
print(type(2.2))  
print(type(complex(2,3)))

**Strings:**

Strings are defined either with a single quote or a double quotes. The difference between the two is that using double quotes makes it easy to include apostrophes.

print(type('Hello World'))  
print(type("Today's News Paper"))

**Variables:**

A variable is nothing but a name that refers to a value. An assignment statement creates new variables and gives them values.

name="Mr. XYZ"  
id=123  
height=165.5

print(name)  
print(id)  
print(height)

The type of a variable means the type of the value it refers to.

print(type(name))  
print(type(id))  
print(type(height))

You can do assignments on more than one variable “simultaneously” on the same line like the following code.

a, b, c = "Make", "Me", "Analyst"  
d=a+b+c  
print(d)

## Python Built-in Data Types

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

|  |  |
| --- | --- |
| **Text Type:** | str |
| **Numeric Types:** | int, float, complex |
| **Sequence Types:** | list, tuple, range |
| **Mapping Type:** | dict |
| **Set Types:** | set, frozenset |
| **Boolean Type:** | bool |
| **Binary Types:** | bytes, bytearray, memoryview |

### Getting the Data Type

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

Example: Print the data type of the variable x:

x = 5

print(type(x))

### Setting the Data Type

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

|  |  |
| --- | --- |
| **Example** | **Data Type** |
| x = "Hello World" | str |
| x = 20 | int |
| x = 20.5 | float |
| x = 1j | complex |
| x = ["apple", "banana", "cherry"] | list |
| x = ("apple", "banana", "cherry") | tuple |
| x = range(6) | range |
| x = {"name" : "John", "age" : 36} | dict |
| x = {"apple", "banana", "cherry"} | set |
| x = frozenset({"apple", "banana", "cherry"}) | frozenset |
| x = True | bool |
| x = b"Hello" | bytes |
| x = bytearray(5) | bytearray |
| x = memoryview(bytes(5)) | memoryview |

# Python Statements

Statements are instructions or piece of codes that Python interpreter can execute. We have already seen two kinds of statements: print and assignment. There are other kinds of statements like if statement, for statement, while statement etc.

When you type a statement, the interpreter executes it and displays the result, if something is there. If you write a script it usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute one by one.

print(100)  
x = 200  
y=400  
z=x+y  
print(z)

## Multi-line statement

In Python, end of a statement is marked by a newline character. But You can write a statement with multiple lines using character (\). Check the following example.

st = "I " + "am" + " Mr." + \  
" X."+" I live in " \  
"city Y."

print(st)

Line continuation is implied inside parentheses ( ), brackets [ ] and braces { } in Python. This is called explicit line continuation. For example, you can write the above multi-line statement as the following code.

st = ("I " + "am" + " Mr." +  
" X."+" I live in "  
"city Y.")

print(st)

In Python, end of a statement is marked by a newline character. But You can write a statement with multiple lines using character (\). Check the following example. You can use [ ] and { } for the same purpose described above.

st = ["I " + "am " + "Mr. " +  
" X."+" I live in "+  
"city Y"]  
print(st)

You can write multiple statements in a single line using semicolons, as following example.

x= 100; y = 200; c = x\*y

print(c)

## **Python : Indentation**

One of the most distinctive features of Python is its use of certain indentation style to mark blocks of code. Once you are wrting python code just be careful of few things:

* In Python white spaces are important!
* The indentation is important!
* If you write program that is not correctly indented, it shows either errors or does not give result what you want!
* Python is case sensitive!
* You can’t safely mix tabs and spaces in Python

Normally, we use tabs or four whitespaces for indentation.

smallest\_so\_far = 50

for the\_num in [9, 41, 12, 3, 74, 15] :

if the\_num < smallest\_so\_far :

smallest\_so\_far = the\_num

print (smallest\_so\_far)

# Python Operators and Expressions

Operators are special symbols that are useful for doing computations like addition, subtraction, multiplication, division, and exponentiation etc. The operators are always applied to some values which are called operands.

Python has so many built-in operators to perform different arithmetic and logical operations. There are main 7 types of operators in Python.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment operators
6. Identity operators
7. Membership operators

## Arithmetic Operators



**Examples:**

print(35/6)

print(3.14\*10)

print(10+41)

print(10%4)

print(5\*\*2)

(5+9)\*(15-7)

## Relational Operators

Below table shows the relational operators in Python.These operators are used to compare values.



**Examples:**

a=10

b=10

print(a<b)

print(a>b)

print(a==b)

print(a<=b)

print(a>=b)

## Bitwise Operators

Bitwise operators act on operands bit by bit as if they are string of binary digits.



**Examples:**

a=1

b=2

print(a&b)

print(a|b)

print(a^b)

print(~a)

print(a<<b)

print(a>>b)

## Assignment operators

In Python, we use Assignment operators to assign values to variables. Following table covers all assignment operators available in Python.



## Identity operators

There two identity operators in Python are is and is not. we use Identity Operators to compare the memory location of two objects.



**Examples:**

a = 2

b=7

print(a is not b)

print(a is b)

x=[1,2,3]

y=[1,2,3]

print(x is y)

## Membership Operators

In Python, there are operators that are mainly useful to test for membership in a sequence such as lists, strings or tuples.operators test for membership in a sequence such as lists, strings, tuples, set and dictionary.



**Examples:**

a=[1,2,3,4]

b=3

print(b in a)

x="Make Me Analyst"

y="Analyst"

print(y in x)

print(y not in x)

## Expressions

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

## Order of operations

If more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

1. Parentheses have the highest precedence. It can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60, even if it doesn’t change the result.
2. Exponentiation has the next highest precedence, so 2\*\*1+1 is 3, not 4, and 3\*1\*\*3 is 3, not 27.
3. Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precence. So 2\*3-1 is 5, not 4, and 6+4/2 is 8, not 5.
4. Operators with the same precedence are evaluated from left to right. So the expression 5-3-1 is 1, not 3, because the 5-3 happens first and thened 1 is subtracted from 2.

When you have doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

## String operations:

The + operator works perfectly with strings, but keep in mind that it is not addition in the mathematical sense.Actuallly, it performs concatenation, which means joining the strings by linking them end to end. For example:

name="Mr. X"

age="30"

s="I am "+ name + "."+ "My age is "+ age

print(s)

# Take Input from User in Python

Sometimes you would like to take input for a particular variable from the user via keyboard. In python, **input()** function is a built-in function for taking input from user. When this function is called, the program stops and waits for receiving a input. When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.

i = input()  
print(i)

**Output:**

>>> input = input()  
MakeMeAnalyst  
>>> print(input)  
MakeMeAnalyst  
>>>

It is a better to print a prompt telling the user what is the input they should enter . You can pass a string to input to be displayed to the user before pausing for input:

>>> name = input('Enter your name?\n')  
Enter your name?  
Mr. X  
>>> print(name)  
Mr. X  
>>>

The sequence \n at the end of the prompt represents a newline, which is a special character that causes a line break. That’s why the user’s input appears below the prompt.

## Take an integer as an Input

If you expect the user to type an integer, you can try to convert the return value to int using the int() function:

prompt = 'What is your age?\n'  
i=input(prompt)  
print(i)  
print(type(i)) #It returns a string  
i=int(i) #Convert it to integer.  
print(i)

**Output:**

>>>  
What is your age?  
30  
30  
<class ‘str’>  
30  
>>>

# Python Comments

In Python, comments start with the # symbol.

x = 10 # assign 10 to x

print("Value of x is:",x) #print value of x

## Multi-line comments

If you want comments that extend multiple lines, one way of doing it is to put hash (#) in the beginning of each line. For example:

#Here is an example of  
#multi-lines comments  
#in python.

Other way of doing the same multi-line comments just putting triple quotes, either ''' or """.

"""Here is an example of  
multi-lines comments  
in python."""

OR

'''Here is an example of  
multi-lines comments  
in python.'''

## Docstring in Python

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. An object’s docstring is defined by including a string constant as the first statement in the object’s definition. For example, the following function defines a docstring:

def my\_fun():

"""Take two numbers as input

and print the sum of the two numbers.

"""

a=10

b=20

c=a+b

print(c)

print(my\_fun.\_\_doc\_\_)

my\_fun()

## Declaration of docstrings

The following Python file shows the declaration of docstrings within a python source file:

"""

Assuming this is file called test.py, then this string is

first statement in the file. This will become the "test" module's

docstring when the file is imported.

"""

class TestClass(object):

"""The test class's docstring"""

def test\_method(self):

"""The test method's docstring"""

def test\_function():

"""The test function's docstring"""

import test  
help(test)  
help(test.TestClass)  
help(test.TestClass.test\_method)  
help(test.test\_function)

**Output:**

>>> import test  
>>> help(test)  
Help on module test:

NAME  
test

DESCRIPTION  
Assuming this is file called test.py, then this string is  
first statement in the file. This will become the “test” module’s  
docstring when the file is imported.

>>> help(test.TestClass.test\_method)  
Help on class TestClass in module test:

>>> help(test.test\_function)  
Help on function test\_method in module test:

# Python Conditional Execution

## if Statement

When we write programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. The simplest form is the if statement. The boolean expression after the if statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the if statement are indented. Check the following example.

x=10  
if x > 0 :

print('x is positive')

## Alternative execution: if-else Statement

A second form of the if statement is alternative execution.,In this case there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

x=11

if x%2 == 0 :

    print('x is even')

else :

    print('x is odd')

## Chained conditionals: if…elif…else

Sometimes there are more than two possibilities. Therefore you need more than two branches. One way to express a computation like that is a chained conditional like below:

a=10

b=20

if a < b:

print('a is less than b')

elif a > b:

print('a is greater than b')

else:

print('a and b are equal')

There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one.

a=10

b=20

if a < b:

print('a is less than b')

elif a > b:

print('a is greater than b')

elif a==b:

print('a and b are equal')

## Nested conditionals

One conditional can also be nested within another. You could have written the three-branch example like this:

a=20  
b=10  
if a == b:

print('a and b are equal')

else:

if a < b:

print('a is less than b')

else:

print('a is greater than b')

## Catching exceptions using try and except

There is a conditional execution structure built into Python to handle certain types of expected and unexpected errors called “try / except”. The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error. Lets consider the following the example:

i=int(input("Enter a number\n"))  
print(i)

For the above code if you don’t enter any number just hit enter without giving any input then you will get an error like this:

Traceback (most recent call last):  
File “<pyshell#4>”, line 1, in <module>  
i=int(input(“Enter a number\n”))  
ValueError: invalid literal for int() with base 10: ”

Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds. If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

try:

i=int(input("Enter a number\n"))

print(i)

except:

print("Please enter a number")

**Output:**

>>>

Enter a number

Please enter a number  
>>>

# Python Functions

Functions can reduce the program smaller by eliminating repetitive code. Any point of time, if you make a change, you just change it in one place. SO, creating function allows to name a group of statements, which makes your program easier to read, understand, and debug.  Basically, when you define a function, you specify the name and the sequence of statements. Later, you can “call” the function by name. We have already seen one example of a function call:

>> type(10)  
<class 'int'>

Here the name of the function is type. The expression in parentheses is called the argument of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the type function, is the type of the argument. A function may “returns” a result. The result is called the return value.

## Define function and function call

def is a keyword that indicates that this is a function definition. A function definition specifies the name of a new function and the sequence of statements that execute when the function is called. Here is an example:

def print\_myname():

print("I'm Mr. K.")

print('I am from city Y.')

Here is the syntax for calling them:

print(print\_myname)  
print(type(print\_myname))  
print\_myname()

## Parameters and arguments

Inside the function, the arguments are assigned to variables called parameters. Here is an example of a user-defined function that takes an argument:

def add(a, b):

add1 = a + b

return add1

x = add(3, 5)

print(x)

## Built-in functions in Python

Python provides a number of important built-in functions. Those built-in functions can be used without providing the function definition. Few examples are given below.

print(max(1,2,3,4,5))  
print(min(1,2,3,4,5))  
print(len("Hi! I am Mr. K"))

## Type conversion functions

Sometimes you need to convert values from one type to another. Python also provides built-in functions for that. For example, The int function takes any value and converts it to an integer, if it can, or give errors otherwise:

print(int('10'))

print(int("Hello! I am Mr. K")) # You will get ValueError for this.

print(int(1.99999))

float converts integers and strings to floating-point numbers:

print(float(12))

print(float('2.190'))

Similarly, str converts its argument to a string:

print(str(12))

print(str(2.1))

## Random numbers

To create random numbers in python you can use random() function which returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series.

Check the following example to produce 5 random numbers.

import random

for i in range(5):

x = random.random()

print(x)

## randint() Function in Python

There is another function called randint which takes the parameters low and high, and returns an integer between low and high (including both).

random.randint(5, 10)

random.randint(15, 20)

## choice() Function in Python

To choose an element from a sequence at random, you can use choice:

t = [1, 2, 3,4,5]

random.choice(t)

## Math functions in Python

Python provides math module for mathematical functions. Before you can use the module, you have to import it:

import math  
print(math.pi)  
print(math.sqrt(16) / 4.0)  
print(math.sin(90))

# Python Loops

## Python While Loop

Iteration is very common in any programming language. Python provides several features to make it easier. One form of iteration in Python is the while statement.  
Flow of execution for a while statement:

* 1. Evaluate the condition is True or False.
  2. If the condition is false, exit the while statement and continue execution atthe next statement.
  3. If the condition is true, execute the body and then go back to step 1.

Here is a simple program:

# To take input from the user.

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

n = 10

while n <15 :

print(n)

n = n + 1

print('STOP!!!')

For the above loop, we would say, “It had five iterations”, which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that the condition becomes false and the loop terminates. The variable which helps to finish the loop is called iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

## Python “Infinite loops” and break

You can write an infinite loop on purpose and then use the break statement to jump out of the loop.

n = 10

while True :

print(n)

n = n + 1

print('STOP!!!')

If you mistakenly run the above code then you will see that it will run forever. While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using **break** when we have reached the exit condition. For example, suppose you want to take input from the user until they type done.  
You could write:

while True:

line = input('Enter "STOP" to stop the loop\n')

if line == 'STOP':

break

print(line)

print('STOP!')

Here, the loop condition is True, which is always true, so the loop runs repeatedly until it hits the break statement.

## Finishing iterations with continue in Python

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types “STOP”, but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

while True:

line = input('> ')

if line[0] == '#':

continue

if line == 'done':

break

print(line)

print('Done!')

**Example 2:**

for i in "Make Me Analyst":

if i == "M":

continue

print(i)

print("STOP")

## Python for Loop

Sometimes You want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When you have a list of things to loop through, you can construct a definite loop using a for statement. You call the while statement an indefinite loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set. The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body:

emp = ['Seba', 'Kattula', 'Mohan']

for e in emp:

print('Hello:', e)

print('Done!')

**Example 2:**

arr=[1,2,3,4,5]

for i in arr:

print(i)

## The range() function in Python

You can generate a sequence of numbers using range() function. range(5) will generate numbers from 0 to 4 (5 numbers). You can also define the start, stop and step size as **range(start,stop,step size)**. step size defaults to 1 if not provided. You can use this function in a list() to output all the items in it.

# Program to iterate through a list using indexing

arr = [1,2,3,4,5]

# iterate over the list using index

for i in range(len(arr)):

print(arr[i])

**Example 2:**

# Program to iterate through a list using indexing

arr = ["A","B","C","D"]

# iterate over the list using index

for i in range(len(arr)):

print(arr[i])

## Bonus Example: Counting and summing loops

count = 0

for i in [1,2,3,4,5]:

count = count + 1

print('Count: ', count)

## Bonus Example: Maximum and minimum loops

largest = None

print('Before:', largest)

for i in [3, 4, 12, 90, 44, 150]:

if largest is None or i > largest :

largest = i

print('Loop:', i, largest)

print('Largest:', largest)

smallest = None

print('Before:', smallest)

for i in [3, 4, 12, 90, 44, 150]:

if smallest is None or i < smallest :

smallest = i

print('Loop:', i, smallest)

print(Smallest:', smallest)

# Python Strings

A string is a sequence of characters. You can access the characters one at a time with the bracket operator. The expression in brackets is called an index. The index indicates which character in the sequence you want to print.

name="Mr. X"  
l = name[0]  
print(l)

## Getting the length of a string using len() function

print(len(name))

To get the last letter of a string, you might try this:

print(l[len(l)-1])

Alternatively, you can use negative indices, which count backward from the end of the string. The expression l[-1] yields the last letter, l[-2] yields the second to last, and so on.

print(name[-1])  
print(name[-2])

## Traversing a  string with a loop

One way to write a traversal is with a **while loop**:

i = 0

while i < len(name):

letter = name[i]

print(letter)

i = i + 1

One way to write a traversal is with a **for loop**:

for char in name:

print(char)

## String slices

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

s = 'Make Me Analyst'

print(s[0:4])

print(s[8:len(s)])

print(s[:4])

print(s[:len(s)])

## Strings are immutable:

Strings are immutable in Python. It means you can’t change an existing string. Let’s try the below example:

str = 'Make Me Analyst'  
str[0]='T'

If you run the above code, you will get an error like this: TypeError: ‘str’ object does not support item assignment  
The reason for the error is that strings are immutable. The best you can do is create a new string that is a variation on the original:

str = 'Make Me Analyst'  
new\_str='Hi! '+ str[8:len(str)]

This example concatenates a new first word onto a slice of the string and it has no effect on the original string.

## Looping and counting

The following program counts the number of times the letter “M” appears in a string:

str = 'Make Me Analyst'

count = 0

for letter in str:

if letter == 'M':

count = count + 1

print(count)

## The in operator in Python

str = 'Make Me Analyst'

a='Analyst' in str

print(a)

b='x' in str

print(b)

## String comparison

The comparison operators work on strings.  Following code checks if two strings are equal:

word='Analyst'

if word=='Analyst':

print('Both are same!')

Some comparison operations are useful for putting words in alphabetical order:

word='Orange'

if word < 'Apple':

print('Your word, ' + word + ', comes before Apple')

elif word > 'Apple':

print('Your word, ' + word + ', comes after Apple.')

else:

print('All right, Orange!!!')

**Note:**Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters

# Python Lists

Like a [**string**](http://makemeanalyst.com/python-programming/strings/), a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called elements or sometimes items.

## How to create a list?

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

This is an example of a list of five integers.

numbers = [10, 20, 30, 40, 50]  
print(numbers)

Here is an empty list.

empty = []

Below example is a list of three strings.

food = ['Hot dog','Sandwich', 'Hamburger']  
print(food)

You can also create a list with mixed datatypes

mixed\_list = [1, "Python", 1.5]  
print(mixed\_list)

The following list contains a string, a float, an integer, and another list:

nested\_list = ['Python', 2.0, 5, [10, 20]]  
print(nested\_list)

## Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a [**string**](http://makemeanalyst.com/python-programming/strings/): the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

food = ['Hot dog','Sandwich', 'Hamburger']  
print(food[0])  
print(food[1])

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

numbers = [10, 20]  
numbers[0] = 100  
numbers[1] = 200  
print(numbers)

The in operator also works on lists.

food = ['Hot dog','Sandwich', 'Hamburger']  
print('Hot dog' in food)  
print('French fries' in food)

## How to access elements from a list?

You have already seen in the above example that we can use the index operator [] to access an item in a list. Index starts from 0.  So, a list having 3 elements will have index from 0 to 2.

## List Index

food=['Hot dog','Sandwich', 'Hamburger']  
print(food[0])  
print(food[1])

If you try to read or write an element that does not exist, you get an IndexError

print(food[3])

## Negative indexing

If an index has a negative value, it counts backward from the end of the list.

food=['Hot dog','Sandwich', 'Hamburger']  
print(food[-1])  
print(food[-2])

## Traversing a list

The most common way to traverse the elements of a list is with a for loop.

food=['Hot dog','Sandwich', 'Hamburger']

for i in food:

print(i)

Above method works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to traverse the list is to combine the functions range and len:

food=['Hot dog','Sandwich', 'Hamburger']

for i in range(len(food)):

print(food[i])

## List operations

The + operator concatenates lists:

a = [1, 2, 3]  
b = [4, 5, 6]  
c = a + b  
print(c)

Similarly, the operator repeats a list a given number of times:

a=[0]\*4  
print(a)  
b=[1,2,3]\*3  
print(b)

The first example repeats four times. The second example repeats the list three times.

## How to slice lists in Python?

The slice operator also works on lists. You can access a range of items in a list by using the slicing operator (colon).

l = ['make','me', 'analyst']  
# get elements 2nd to 3rd  
print(l[1:3])  
# get elements beginning to 2nd  
print(l[:-1])  
# get elements 2nd to end  
print(l[1:])  
# elements beginning to end  
print(l[:])

Since lists are mutable, it is often useful to make a copy before performing operations  
that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

t = ['a', 'b', 'c', 'd', 'e', 'f']  
t[1:3] = ['x', 'y']  
print(t)

## List methods

Python provides methods that operate on lists. For example, **append** adds a new element to the end of a list:

x = ['a', 'b', 'c']  
x.append('d')  
print(x)

**extend** takes a list as an argument and appends all of the elements:

x1 = ['a', 'b', 'c']  
x2 = ['d', 'e']  
x1.extend(x2)  
print(x1)

This example leaves x2 unmodified.

**sort** arranges the elements of the list from low to high:

t = ['d', 'c', 'e', 'b', 'a']  
t.sort()  
print(t)

Most list methods are void; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

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

### ***pop*** *operator*

There are several ways to delete elements from a list. If you know the index of the element you want, you can use **pop**:

t = ['a', 'b', 'c']  
x = t.pop(1)  
print(t)  
print(x)

### ***del*** operator

pop modifies the list and returns the element that was removed. If you don’t provide an index, it deletes and returns the last element.

If you don’t need the removed value, you can use the **del** operator:

t = ['a', 'b', 'c']  
del t[1]  
print(t)

### remove() Function

If you know the element you want to remove (but not the index), you can use **remove**:

t = ['a', 'b', 'c']  
t.remove('b')  
print(t)

The return value from remove is None. To remove more than one element, you can use del with a slice index:

t = ['a', 'b', 'c', 'd', 'e', 'f']  
del t[1:5]  
print(t)

As usual, the slice selects all the elements up to, but not including, the second index.

## Lists and functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

nums = [3, 4, 5, 6, 7, 8]  
print(len(nums))  
print(max(nums))  
print(min(nums))  
print(sum(nums))  
print(sum(nums)/len(nums))

The ***sum()*** function only works when the list elements are numbers. The other functions (***max()***, **len()**, etc.) work with lists of strings and other types that can be comparable.

You could rewrite an earlier program that computed the average of a list of numbers entered by the user using a list. First, the program to compute an average without a list:

total = 0

count = 0

while (True):

inp = input('Enter a number: ')

if inp == 'done': break

value = float(inp)

total = total + value

count = count + 1

average = total / count

print('Average:', average)

In this program, you have count and total variables to keep the number and running total of the user’s numbers as we repeatedly prompt the user for a number. You could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

numlist = list()

while (True):

inp = input('Enter a number: ')

if inp == 'done': break

value = float(inp)

numlist.append(value)

average = sum(numlist) / len(numlist)

print('Average:', average)

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

## Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

l="Make Me Aanlyst"  
t = list(l)  
print(t)

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

s = 'Make Me Aanlyst'  
t = s.split()  
print(t)

Once you have used split to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list. You can call split with an optional argument called a delimiter that specifies which characters to use as word boundaries. The following example uses a hyphen.

s = 'make-me-analyst'  
delimiter = '-'  
s.split(delimiter)

***join*** is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

t = ['Make', 'Me', 'Analyst']  
delimiter = ' '  
delimiter.join(t)

## List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, delete\_head removes the first element from a list:

def delete\_head(t):

del t[0]

Here’s how it is used:

letters = ['a', 'b', 'c']

delete\_head(letters)

print(letters)

The parameter t and the variable letters are aliases for the same object. It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

t1 = [1, 2]

t2 = t1.append(3)

print(t1)

print(t2)

t3 = t1 + [3]

print(t3)

t2 is t3

# Python Dictionaries

A dictionary is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type. Dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value.

The association of a key and a value is called a key-value pair or sometimes an item. As an example, we’ll build a dictionary that maps from English to German words, so the keys and the values are all strings.

## How to create a dictionary?

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

eng2gr = dict()

print(eng2gr)

The curly brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

eng2gr['one'] = 'eins'

This line creates an item that maps from the key ’one’ to the value “eins”. If you print the dictionary again, you see a key-value pair with a colon between the key and value:

print(eng2gr)

This output format is also an input format. For example, you can create a new dictionary with three items. But if you print eng2gr, you might be surprised:

eng2gr = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}

print(eng2gr)

## How to access elements from a dictionary?

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable. But that’s not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

eng2gr = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}

print(eng2gr['two'])

The key ’two’ always maps to the value “zwei” so the order of the items doesn’t matter.If the key isn’t in the dictionary, you get an exception:

>>> print(eng2gr['four'])  
KeyError: 'four'

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.

print(eng2gr.get('two'))

print(eng2gr.get('three'))

The len function works on dictionaries; it returns the number of key-value pairs:

len(eng2gr)

## How to change or add elements in a dictionary?

Dictionaries 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.

eng2gr = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}

eng2gr['four'] = 'four' #Add Element

print(eng2gr)

eng2gr['four'] = 'vier'  #Update Element

print(eng2gr)

## Dictionary Membership Test

The in operator works on dictionaries; it tells you whether something appears as a key in the dictionary.

>>> 'one' in eng2gr  
True  
>>> 'eins' in eng2gr  
False

To see whether something appears as a value in a dictionary, you can use the method values, which returns the values as a list, and then use the**in** operator:

vals = list(eng2gr.values())

>>>'eins' in vals

True

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a hash table that has a remarkable property: the ***in*** operator takes about the same amount of time no matter how many items there are in a  
dictionary. I won’t explain why hash functions are so magical, but you can read more about it at [wikipedia.org/wiki/Hash\_table](http://wikipedia.org/wiki/Hash_table).

## How to delete or remove elements from a dictionary?

You 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.

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

eng2gr = {'one': 'eins', 'two': 'zwei', 'three': 'drei', 'four':'vier'}

# remove a particular item

print(eng2gr.pop('four'))

print(eng2gr)

# remove an arbitrary item

print(eng2gr.popitem())

print(eng2gr)

# delete a particular item

del eng2gr['one']

print(eng2gr)

# remove all items

eng2gr.clear()

## Python Dictionary Methods

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

|  |  |
| --- | --- |
| 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 |

Here are a few example use of these methods.

fruits = {}.fromkeys(['Orange','Apple','Banana'], 0)

print(fruits)

for item in fruits.items():

print(item)

list(sorted(fruits.keys()))

# Referecnces

<https://code.visualstudio.com/docs/languages/python>

<https://code.visualstudio.com/docs/python/python-tutorial>

<https://www.geeksforgeeks.org/python-keywords-and-identifiers/>

<https://codesteps.com/2018/09/02/python-simple-and-compound-statements/>

<http://makemeanalyst.com/python-programming/>