Python Documentation V1.0

# List of Contents

1.0 - Data Types

1.1 - [Integers](#_1timsq4lrc4i) 1

1.2 - [Floats](#_lu44nqdnekww) 1.0

1.3 - [Boolean](#_6bmyl67w7gsy) True

1.4 - [Strings](#_y9k07ne0lhqh) “ “

1.4.1 - [Formatting](#_zg6iimnh4xcp)

1.4.2 - [*f* strings](#_u0q3x678engb)

1.4.3 - [String Methods](#_u0rq6226d26e)

1.5 - [Lists](#_uwv28e8xq9k0) [\*]

1.5.1 - [List Methods](#_m7tot6rkmwdo)

1.5.1 - List Comprehension

1.6 - Tuples (\*)

1.6.1 - Tuple Methods

1.7 - Sets {\*}

1.7.1 - Set Methods

1.8 - Dictionary {\*:\*}

1.8.1 - Dictionary Methods

1.8 - Class

2.0 - Math  
 2.1 - [Math Operators](#_cqdlden23qdm)

2.2 - [Assignment Operators](#_5e1oened1qv7)

2.3 - [Comparison Operators](#_jgpd2bq66c0k)

2.4 - [Logical Operators](#_lpzdzqc87vif)

2.5 - [“in” Operator](#_iftf03rebvr)

2.6 - [Operator Order](#_85st28ejs8cb)

3.0 - Conditional Statements\

3.1 - [Indentation](#_g3bcssdjzl82)

3.2 - [Else](#_3njxa3o7tck5)

3.3 - [Elif](#_3418zcjom6fk)

4.0 - Functions

4.1 - [Arguments and Parameters](#_4kmylr9qln49)

4.2 - [Return Statements](#_dgmaeqky73jk)

4.3 - Default Parameter Values

4.4 - Lambda Functions

5.0 - Loops

5.1 - [For Loops](#_ju7hrbv9kts0)

5.2 - While Loops

5.3 - Break and Continue

5.4 - Pass

6.0 - Built-In Functions

6.1 - [File I/O](#_kfcjp5ehy54j)

6.2 - [Character Recognition](#_878u5rim5ls0)

6.3 - [Math Functions](#_amxjh6i22f6p)

6.4 - [Type Functions](#_y7sfeff799xx)

6.5 - [Common Miscellaneous Functions](#_7zy74zabhv24)

7.0 - External Libraries

7.1 - [Regex](#_795kidxkezrp) (--> [*re* Methods](#_u7gadc5pllly))

7.2 - [Random](#_84wcosx4ama3)

7.3 - NumPy

7.4 - Pandas

7.5 - MatPlotLib

# 1.0 Data Types

*We all start somewhere*

## 1.1 Integers

Integers are whole numbers

Examples of integers: 5, 6, 10, 1111

To set a variable to integer, use int()

E.g.

| Num1 = int(input("Enter number: ")) Num2 = int(999) String = "1234" Integers = int(String)  print(Num1,Num2,Integers) |
| --- |

Output: (inputs are underlined)

| >>> Enter number: 123 >>> 123,999,1234 |
| --- |

## 1.2 Floats

Floats are numbers with decimal points. Whole numbers can also be floats, indicated with a .0 behind them.

Examples of floats: 5.663, 9.0, 1.1111

To set a variable to float, use float()

E.g.

| Number = float(12.0) String = "999.999" Float = float(String) |
| --- |

## 1.3 Boolean (bool)

Boolean is either True or False. When any expression/conditional is evaluated, it will return a boolean

Examples of boolean: True, False

Examples of expressions/conditionals that return a boolean (True/False): 2 == 2, 4 > 5, variable x == variable, len(“hello”) > 5

—--------------------------------------------------------------------

Integers, Lists and strings can also have an associated boolean value.

* For integers, the value 0 returns False, any other number returns True
* An empty list ([]) returns False, a non-empty one returns True
* An empty string (“”/’’/’’’’’’) returns False, a non-empty one returns True

Here are some practical examples:

Check if a list is empty:

| list = [1,2,3]  if list:  print("This list is not empty")  else:  print("This list is empty") |
| --- |

Check if the user enters 0

| inp = int(input("Enter the number of apples you collected today: ")) if inp:  print("good job!") else:  print("you suck.") |
| --- |

## 1.4 Strings

Strings can be used to contain text and just about everything.

They are denoted by single quotes ('), double quotes (") or triple quotes (''')

Examples of strings: "5.0", "4", 'Hello!', '''False''', 'True', " "

—----------------------------------------------------------------

Strings have a special character, the backslash (\)

It can be used for whitespace characters,

\n -> newline

\t -> tab

And to add quotes to a string, when you normally would not be allowed:

\" -> double quote

\' -> single quote

Examples:

| print("Hello\nWorld!")  >>> Hello  >>> World!   print("Hello\tWorld!")  >>> Hello World!   print("Hello \"World!\"")  >>> Hello "World!" |
| --- |

## 1.4.1 Formatting

*Gotta make it look good.*

### Placeholders

SYNTAX

| string = "My name is {}, I'm {}"  string.format(*value1*,*value2*...) |
| --- |

PARAMETERS

{} - where the placeholder text is

value1 - what text or whatever you want in the 1st {}

value2 - another text or something that will be in the next {}

E.g.

| print("My name is {}, I'm {}".format("John",36)) |
| --- |

Output:

| >>> My name is John, I'm 36 |
| --- |

Placeholders also can be indexed or named

E.g.

| print("My name is {1}, I'm {0}".format(36,"John")) print("My name is {name}, I'm {age}".format(name = "Tom", age = 18)) |
| --- |

Output:

| >>> My name is John, I'm 36  >>> My name is Tom, I'm 18 |
| --- |

To format, use a format like this '{:.f}'

E.g.

| '{:.2f}'.format(23.456) |
| --- |

It also can be assigned to a var, such as

| f = '{:.2f},{:>15}'  print(f.format(23.456, "text")) |
| --- |

Output:

| >>> 23.46, text |
| --- |

List of formats

| :< - aligns to the left :> - aligns to the right :^ - aligns to the center := - aligns to the left most position :+ - places a plus sign to indicate whether it is positive or negative :- - places a minus sign for negative values : - Uses a space to insert an extra space before positive numbers (minus for -tive values) :, - Uses a comma as a thousand separator :\_ - Uses a underscore as a thousand separator :b - Binary format :c - converts into unicode characters :d - Decimal format :e - Scientific format, lowercase e :E - Scientific format, uppercase E :f - fix point number format  :.(num)f - round to (num) decimal places :g - general format :G - general format; uses an uppercase E for scientific notations :o - Octal format :x - Hex format, lowercase :X - Hex format, uppercase :n - Number format :% - Percentage format :(num) - Number of characters (uses spaces for extra characters) |
| --- |

## 1.4.2 *f* strings

*Formatting made easy.*

SYNTAX

Example 1:

| *# Python3 program introducing f-string* val = 'Geeks' print(f"{val}for{val} is a portal for {val}.") *#the letter f precedes the string to denote a f string*  name = 'Tushar' age = 23 print(f"Hello, My name is {name} and I'm {age} years old.") |
| --- |

Output:

| >>> GeeksforGeeks is a portal for Geeks. >>> Hello, My name is Tushar and I'm 23 years old. |
| --- |

Example 2:

| *# Prints today's date with help* *# of datetime library* import datetime   today = datetime.datetime.today() print(f"{today:%B %d, %Y}") |
| --- |

Output:

| >>> April 04, 2018 |
| --- |

LIMITATIONS

Example 3: Backslash Cannot be used in format string directly.

| f"newline: {ord('\n')}" |
| --- |

Output: Syntax Error

| >>> Traceback (most recent call last):  Python Shell, prompt 29, line 1  Syntax Error: f-string expression part cannot include a backslash: ,  line 1, pos 0 |
| --- |

However, there is a workaround by putting the ‘/n’ in a separate variable and calling the variable through the *f* string to create the same effect.

Example 4:

| newline = ord('\n')  print(f"newline: {newline}") |
| --- |

Output:

| >>> newline: 10 |
| --- |

## 1.4.3 String Methods

### **str.capitalize()**

*The capitalize() method converts the first character of a string to an uppercase letter and all other alphabets to lowercase.*

SYNTAX

**Example 1:** *Python capitalize()*

| sentence = "i love PYTHON"  *# converts first character to uppercase and others to lowercase* capitalized\_string = sentence.capitalize()  print(capitalized\_string) |
| --- |

Output:

| >>> I love python |
| --- |

In the above example, we have used the **capitalize()** method to convert the **first character** of the sentence string to uppercase and the **other characters** to lowercase.

Here, sentence.**capitalize()** returns "Python is awesome" which is **assigned to capitalized\_string**.

The **capitalize()** method returns a new string and **doesn't modify the original string**. For example:

**Example 2:** *str.capitalize() doesn’t change the original string*

| sentence = "i am learning PYTHON." *# capitalize the first character*  capitalized\_string = sentence.capitalize() *# the sentence string is not modified*  print('Before capitalize():', sentence) print('After capitalize():', capitalized\_string) |
| --- |

Output:

| >>> Before capitalize(): i am learning PYTHON. >>> After capitalize(): I am learning python. |
| --- |

Here, the **capitalize()** method doesn't modify the original sentence string.

RETURN VALUE

The **capitalize()** method returns:

* A string with the first letter capitalized and all other characters in lowercase.

——————————————————————————————————————————————————————————————————————

### **\*str.center()**

*The center() method returns a new centered string after padding it with the specified character.*

Example:

| sentence = "Python is awesome"  *# returns the centered padded string of length 24*  new\_string = sentence.center(24, '\*')  print(new\_string) |
| --- |

Output:

| >>> \*\*\*python is awesome\*\*\* |
| --- |

SYNTAX

| str.center(width, [fillchar]) *#where str is a string* |
| --- |

PARAMETERS

The **center()** method takes two parameters:

* **width** - length of the string with padded characters
* **fillchar** (optional) - padding character

Note: If **fillchar** is not provided, whitespace (normal space) is taken as the **default** argument.

RETURN VALUE

The **center()** method returns:

* a string padded with specified **fillchar**

Note: The center() method **doesn't modify** the original string.

EXAMPLES

**Example 1:**

| sentence = "Python is awesome"  *# returns the centered padded string of length 20*  new\_string = sentence.center(20, '$')  print(new\_string) |
| --- |

Output:

| >>> $Python is awesome$$ |
| --- |

In the above example, we have used the **center()** method with the sentence string as sentence.**center(**20,'$'**)**.

The method returns a **new centered string** after padding the sentence with '$' up to length **20**.

**Example 2:**

| text = "Python is awesome"  *# returns padded string by adding whitespace up to length 24* new\_text = text.center(24)  print("Centered String: ", new\_text) |
| --- |

Output:

| >>> Centered String: Python is awesome |
| --- |

Here, we have **not** passed the **fillchar** parameter in the **center()** method. The method pads **whitespace** to text making the length of the centered string **24**.

——————————————————————————————————————————————————————————————————————

### **\*str.casefold()**

*The casefold() method converts all characters of the string into lowercase letters and returns a new string.* ***It is a more aggressive version of str.lower()***

| text = "pYtHon"  *# convert all characters to lowercase* lowercased\_string = text.casefold()  print(lowercased\_string) |
| --- |

Output:

| >>> python |
| --- |

SYNTAX

| str.casefold() |
| --- |

Note: Here, str is a string.

PARAMETERS

* The **casefold()** method doesn't take any parameters.

RETURN VALUE

The **casefold()** method returns:

* a lowercase string

EXAMPLES

**Example 1:**

| text = "PYTHON IS FUN"  *# converts text to lowercase* print(text.casefold()) |
| --- |

Output:

| >>> python is fun |
| --- |

In the above example, we have used the casefold() method to convert all the characters of text to **lowercase**.

Here, text.casefold() modifies the **value** of string1 and returns 'python is fun'.

**Example 2:** *casefold() as an Aggressive str.lower() Method*

The casefold() method is **similar** to the lower() method but it is **more aggressive**. This means the **casefold()** method **converts more characters into lower case** compared to **lower()** .

For example, the German letter **ß** is already lowercase so, the **lower()** method **doesn't** make the conversion.

But the **casefold()** method **will** convert **ß** to its equivalent character **ss**.

| text = 'groß'  *# convert text to lowercase using casefold()* print('Using casefold():', text.casefold())   *# convert text to lowercase using lower()* print('Using lower():', text.lower()) |
| --- |

Output:

| >>> Using casefold(): gross >>> Using lower(): groß |
| --- |

In the above example, we have used the **casefold()** and **lower()** methods to convert 'groß'.

The casefold() method also converts 'ß' to lowercase whereas **lower() doesn't**.

——————————————————————————————————————————————————————————————————————

### **str.count()**

*The count() method returns the number of occurrences of a substring in the given string.*

Example:

| message = 'python is popular programming language'  *# number of occurrence of 'p'* print('Number of occurrence of p:', message.count('p')) |
| --- |

Output:

| >>> Number of occurrence of p: 4 |
| --- |

SYNTAX

| string.count(substring, start=..., end=...) |
| --- |

PARAMETERS

**count()** method only requires a single parameter for execution. However, it also has two optional parameters:

* substring - string whose count is to be found.
* start (Optional) - starting index within the string where search starts.
* end (Optional) - ending index within the string where search ends.

Note: Index in Python starts from 0, not 1.

RETURN VALUE

**count()** method returns the number of occurrences of the substring in the given string.

EXAMPLES

Example 1: *Count the number of occurrences of a given substring*

| *# define string* string = "Python is awesome, isn't it?" substring = "is"  count = string.count(substring)  *# print count* print("The count is:", count) |
| --- |

Output:

| >>> The count is: 2 |
| --- |

Example 2: *Count number of occurrences of a given substring using* ***start*** *and* ***end***

| *# define string* string = "Python is awesome, isn't it?" substring = "i"  *# count after first 'i' and before the last 'i'* count = string.count(substring, 8, 25)  *# print count* print("The count is:", count) |
| --- |

Output:

| >>> The count is: 1 |
| --- |

Here, the counting **starts** after the first ***i*** has been encountered, i.e. 7th *index* position.

And, it **ends** before the last ***i***, i.e. 25th *index* position.

——————————————————————————————————————————————————————————————————————

### **str.endswith()**

*The endswith() method returns True if a string ends with the specified suffix. If not, it returns False.*

Example:

| message = 'Python is fun'  *# check if the message ends with fun* print(message.endswith('fun')) |
| --- |

Output:

| >>> True |
| --- |

SYNTAX

| str.endswith(suffix[, start[, end]]) |
| --- |

PARAMETERS

The **endswith()** takes **three** parameters:

* suffix - **String or tuple** of suffixes to be checked
* start (optional) - Beginning position where suffix is to be checked within the string.
* end (optional) - Ending position where suffix is to be checked within the string.

RETURN VALUE

The endswith() method returns a **boolean**.

* It returns **True** if a string ends with the specified suffix.
* It returns **False** if a string doesn't end with the specified suffix.

——————————————————————————————————————————————————————————————————————

### **\*str.expandtabs()**

*The* ***expandtabs()*** *method returns a copy of the string with all tab characters '\t' replaced with whitespace characters until the next multiple of tabsize parameter.*

SYNTAX

| string.expandtabs(tabsize) |
| --- |

PARAMETERS

The **expandtabs()** takes an integer *tabsize* argument. The default *tabsize* is 8.

RETURN VALUE

The **expandtabs()** returns a string where all '\t' characters are replaced with whitespace characters until the next multiple of *tabsize* parameter.

EXAMPLES

Example 1: *expandtabs() With no Argument*

| str = 'xyz\t12345\tabc'  *# no argument is passed* *# default tabsize is 8* result = str.expandtabs()  print(result) |
| --- |

Output:

| >>> xyz 12345 abc |
| --- |

**How does expandtabs() work in Python?**

The **expandtabs()** method keeps track of the current cursor position.

The position of the **first** '**\t**' character in the above program is **3**. And, the **tabsize** is **8** (if argument is not passed).

The expandtabs() character replaces the '**\t**' with whitespace until the next **tabstop**. The position of '**\t**' is 3 and the first **tabstop** is 8. Hence, the number of spaces after 'xyz' is 5.

The next tab stops are the multiples of tabsize. The next **tabstops** are 16, 24, 32 and so on.

Now, the position of the **second** '**\t**' character is **13**. And, the **next** tab stop is **16**. Hence, there are **3 spaces** after '12345'.

Example 2: *expandtabs() With Different Argument*

| str = "xyz\t12345\tabc" print('Original String:', str)  *# tabsize is set to 2* print('Tabsize 2:', str.expandtabs(2))  *# tabsize is set to 3* print('Tabsize 3:', str.expandtabs(3))  *# tabsize is set to 4* print('Tabsize 4:', str.expandtabs(4))  *# tabsize is set to 5* print('Tabsize 5:', str.expandtabs(5))  *# tabsize is set to 6* print('Tabsize 6:', str.expandtabs(6)) |
| --- |

Output:

| >>> Original String: xyz 12345 abc >>> Tabsize 2: xyz 12345 abc >>> Tabsize 3: xyz 12345 abc >>> Tabsize 4: xyz 12345 abc >>> Tabsize 5: xyz 12345 abc >>> Tabsize 6: xyz 12345 abc |
| --- |

**Explanation**

* The default **tabsize** is **8**. The tab stops are 8, 16 and so on. Hence, there are **5** spaces after 'xyz' and 3 after '12345' when you print the original string.
* When you set the **tabsize** to **2**. The tab stops are 2, 4, 6, 8 and so on. For 'xyz', the tab stop is 4, and for '12345', the tab stop is 10. Hence, there is **1** space after 'xyz' and 1 space after '12345'.
* When you set the **tabsize** to **3**. The tab stops are 3, 6, 9 and so on. For 'xyz', the tab stop is 6, and for '12345', the tab stop is 12. Hence, there are **3** spaces after 'xyz' and 1 space after '12345'.
* When you set the **tabsize** to **4**. The tab stops are 4, 8, 12 and so on. For 'xyz', the tab stop is 4 and for '12345', the tab stop is 12. Hence, there is **1** space after 'xyz' and 3 spaces after '12345'.
* When you set the **tabsize** to **5**. The tab stops are 5, 10, 15 and so on. For 'xyz', the tab stop is 5 and for '12345', the tab stop is 15. Hence, there are 2 spaces after 'xyz' and 5 spaces after '12345'.
* When you set the **tabsize** to **6**. The tab stops are 6, 12, 18 and so on. For 'xyz', the tab stop is 6 and for '12345', the tab stop is 12. Hence, there are **3** spaces after 'xyz' and 1 space after '12345'.

——————————————————————————————————————————————————————————————————————

### **\*str.encode()**

*The* ***encode()*** *method returns an encoded version of the given string.*

Example:

| title = 'Python Programming'  *# change encoding to utf-8* print(title.encode()) |
| --- |

Output:

| >>> b'Python Programming' |
| --- |

SYNTAX

| string.encode(encoding='UTF-8',errors='strict') |
| --- |

PARAMETERS

By default, the **encode()** method doesn't require any parameters.

It returns an utf-8 encoded version of the string. In case of failure, it raises a ***UnicodeDecodeError*** exception.

However, it takes two parameters:

* **encoding** - the encoding type a string has to be encoded to
* **errors** - response when encoding fails. There are six types of error response
  + strict - default response which raises a UnicodeDecodeError exception on failure
  + ignore - ignores the unencodable unicode from the result
  + replace - replaces the unencodable unicode to a question mark ?
  + xmlcharrefreplace - inserts XML character reference instead of unencodable unicode
  + backslashreplace - inserts a \uNNNN escape sequence instead of unencodable unicode
  + namereplace - inserts a \N{...} escape sequence instead of unencodable unicode

EXAMPLES

Example 1: *Encode to Default Utf-8 Encoding*

| *# unicode string* string = 'pythön!'  *# print string* print('The string is:', string)  *# default encoding to utf-8* string\_utf = string.encode()  *# print result* print('The encoded version is:', string\_utf) |
| --- |

Output:

| >>> The string is: pythön! >>> The encoded version is: b'pyth\xc3\xb6n!' |
| --- |

Example 2: *Encoding with error parameter*

| *# unicode string* string = 'pythön!'  *# print string* print('The string is:', string)  *# ignore error* print('The encoded version (with ignore) is:', string.encode("ascii", "ignore"))  *# replace error* print('The encoded version (with replace) is:', string.encode("ascii", "replace")) |
| --- |

Output:

| >>> The string is: pythön! >>> The encoded version (with ignore) is: b'pythn!' >>> The encoded version (with replace) is: b'pyth?n!' |
| --- |

**String Encoding**

Since Python 3.0, strings are stored as Unicode, i.e. each character in the string is represented by a code point. So, each string is just a sequence of Unicode code points.

For efficient storage of these strings, the sequence of code points is converted into a set of bytes. The process is known as encoding.

There are various encodings present which treat a string differently. The popular encodings being utf-8, ascii, etc.

Using the string **encode()** method, you can convert unicode strings into any encodings supported by Python. By default, Python uses utf-8 encoding.

——————————————————————————————————————————————————————————————————————

### **str.find() / str.rfind()**

*The* ***find()*** *method returns the* ***index*** *of the* ***first occurrence*** *of the substring (if* ***found****). If* ***not found****, it returns* ***-1****.*

*The* ***rfind()*** *method does the same thing but with the* ***last occurrence*** *instead.*

Example:

| message = 'Python is a fun programming language'  *# check the index of 'fun'* print(message.find('fun')) |
| --- |

Output:

| >>> 12 |
| --- |

SYNTAX

| str.find(sub[, start[, end]] )  str.rfind(sub[, start[, end]] ) |
| --- |

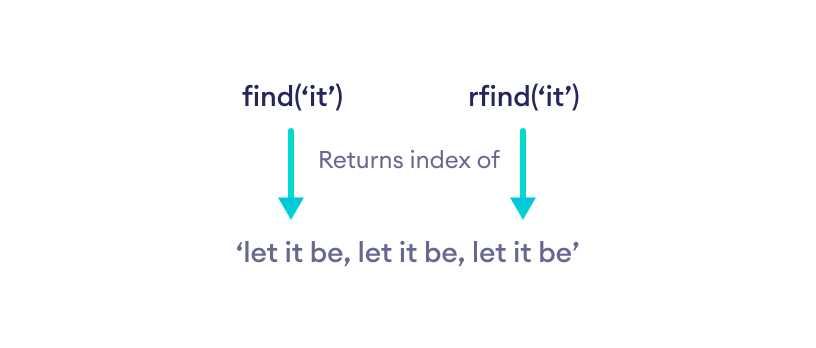
PARAMETERS

Both the **find()** and **rfind()** methods takes at **maximum three** parameters:

* **sub** - It is the substring to be searched in the *str* string.
* **start** and **end** (optional) - The range *str*[start:end] within which substring is searched.

RETURN VALUES  
Both the **find()** and **rfind()** methods returns an integer value:

* If the substring exists inside the string, it returns the index of the first occurrence of the substring.
* If a substring doesn't exist inside the string, it returns -1.

WORKINGS OF **find()** and ***rfind()*** FORMAT

*Return value from find() and rfind()*

EXAMPLES

Example 1: ***find()*** *With No* ***start*** *and* ***end*** *Argument*

| quote = 'Let it be, let it be, let it be'  *# first occurance of 'let it'(case sensitive)* result = quote.find('let it') print("Substring 'let it':", result)  *# find returns -1 if substring not found* result = quote.find('small') print("Substring 'small ':", result)  *# How to use find()* if (quote.find('be,') != -1):  print("Contains substring 'be,'") else:  print("Doesn't contain substring") |
| --- |

Output:

| >>> Substring 'let it': 11 >>> Substring 'small ': -1 >>> Contains substring 'be,' |
| --- |

Example 2: ***find()*** *With start and end Arguments*

| quote = 'Do small things with great love'  *# Substring is searched in 'hings with great love'* print(quote.find('small things', 10))  *# Substring is searched in ' small things with great love'*  print(quote.find('small things', 2))  *# Substring is searched in 'hings with great lov'* print(quote.find('o small ', 10, -1))  *# Substring is searched in 'll things with'* print(quote.find('things ', 6, 20)) |
| --- |

Output:

| >>> -1 >>> 3 >>> -1 >>> 9 |
| --- |

Note: **find()** returns -1 if the substring cannot be found within the string or the given length.

——————————————————————————————————————————————————————————————————————

### **str.index() / str.rindex()**

The **index()** method returns the **lowest index** while **rindex()** returns the **highest index** of a substring contained within the string (**if** **found**). If the substring is **not found**, it raises an **exception**.

Example:

| text = 'Python is fun'  *# find the index of is* result = text.index('is') print(result) |
| --- |

Output:

| >>> 7 |
| --- |

SYNTAX

| str.index(sub[, start[, end]] )  str.rindex(sub[, start[, end]] ) |
| --- |

PARAMETERS

Both the index() and rindex() methods take three parameters:

* **sub** - substring to be searched in the string *str*.
* **start** and **end**(optional) - substring is searched within *str*[start:end]

RETURN VALUES

* If the substring **exists** within the string, it returns the **lowest index [for index()] or highest index [for rindex()]** in the string where the substring is found.
* If the substring doesn't exist inside the string, it raises a ***ValueError*** exception.

The **index()** method is **similar** to the **find()** method for strings.

The only difference is that **find() method returns -1** if the substring is not found, whereas **index() throws an exception**.

EXAMPLES

Example 1: ***index()*** *With Substring argument Only*

| sentence = 'Python programming is fun.'  result = sentence.index('is fun') print("Substring 'is fun':", result)  result = sentence.index('Java') print("Substring 'Java':", result) |
| --- |

Output:

| >>> Substring 'is fun': 19  >>> Traceback (most recent call last):  File "<string>", line 6, in   result = sentence.index('Java')  ValueError: substring not found |
| --- |

Note: Index in Python starts from 0 and not 1. So the occurrence is 19 and not 20.

Example 2: ***index()*** *With start and end Arguments*

| sentence = 'Python programming is fun.'  *# Substring is searched in 'gramming is fun.'* print(sentence.index('ing', 10))  *# Substring is searched in 'gramming is '* print(sentence.index('g is', 10, -4))  *# Substring is searched in 'programming'* print(sentence.index('fun', 7, 18)) |
| --- |

Output:

| >>> 15 >>> 17 >>> Traceback (most recent call last):  File "<string>", line 10, in   print(quote.index('fun', 7, 18))  ValueError: substring not found |
| --- |

——————————————————————————————————————————————————————————————————————

### **GROUP: str.is\_\_\_\_()**

RETURN VALUES

The **is\_\_\_\_()** method returns:

* True - if **all characters** in the string fulfills the condition
* False - if **at least one** character **does not** fulfill the condition

List of **is\_\_\_\_()** methods:

| isalnum - all characters are **alphanumeric** isalpha - all characters are **alphabets** isdecimal - all characters are **decimal characters** isdigit - all characters are **digits (currency signs $ ok)** isidentifier - all the characters are: **uppercase and lowercase**  **letters A through Z, the underscore \_ and except**  **for the first character, the digits 0 through 9** islower - all characters are **lowercase alphabets** isnumeric - all characters are **numbers (or anything numbers**  **related like fractions and currency signs)** isprintable - all characters are **printable in the print() func** isspace - all characters are **whitespace (like space \t \n)** istitle - first character of **each word is capitalized** isupper - all characters are **uppercase alphabets** |
| --- |

——————————————————————————————————————————————————————————————————————

### **str.join()**

*The string* ***join()*** *method returns a string by joining all the elements of an iterable (list, string, tuple), separated by the given separator.*

Example:

| text = ['Python', 'is', 'a', 'fun', 'programming', 'language']  *# join elements of text with space* print(' '.join(text)) |
| --- |

Output:

| >>> Python is a fun programming language |
| --- |

SYNTAX

| string.join(iterable) |
| --- |

PARAMETERS

The **join()** method takes an iterable (objects capable of returning its members one at a time) as its parameter.

Some of the example of iterables are:

* Native data types - List, Tuple, String, Dictionary and Set.
* File objects and objects you define with an *\_\_iter\_\_()* or *\_\_getitem()\_\_* method.

Note: The **join()** method provides a flexible way to create strings from iterable objects. It joins each element of an iterable (such as list, string, and tuple) by a string separator (the string on which the **join()** method is called) and returns the concatenated string.

RETURN VALUES

The **join()** method returns a string created by joining the elements of an iterable by the given string separator.

If the iterable contains any non-string values, it raises the ***TypeError*** exception.

EXAMPLES

Example 1: *Working of the* ***join()*** *method*

| *# .join() with lists* numList = ['1', '2', '3', '4'] separator = ', ' print(separator.join(numList))  *# .join() with tuples* numTuple = ('1', '2', '3', '4') print(separator.join(numTuple))  s1 = 'abc' s2 = '123'  *# each element of s2 is separated by s1* *# '1'+ 'abc'+ '2'+ 'abc'+ '3'* print('s1.join(s2):', s1.join(s2))  *# each element of s1 is separated by s2* *# 'a'+ '123'+ 'b'+ '123'+ 'b'* print('s2.join(s1):', s2.join(s1)) |
| --- |

Output:

| >>> 1, 2, 3, 4 >>> 1, 2, 3, 4 >>> s1.join(s2): 1abc2abc3 >>> s2.join(s1): a123b123c |
| --- |

Example 2: *The* ***join()*** *method with sets*

| *# .join() with sets* test = {'2', '1', '3'} s = ', ' print(s.join(test))  test = {'Python', 'Java', 'Ruby'} s = '->->' print(s.join(test)) |
| --- |

Output:

| >>> 2, 3, 1 >>> Python->->Ruby->->Java |
| --- |

Note: A set is an unordered collection of items, so you may get different output (order is random).

Example 3: *The* ***join()*** *method with dictionaries*

| *# .join() with dictionaries* test = {'mat': 1, 'that': 2} s = '->'  *# joins the keys only* print(s.join(test))  test = {1: 'mat', 2: 'that'} s = ', '  *# this gives error since key isn't string* print(s.join(test)) |
| --- |

Output:

| >>> mat->that >>> Traceback (most recent call last):  File "...", line 12, in <module>  TypeError: sequence item 0: expected str instance, int found |
| --- |

The join() method tries to join the keys (not values) of the dictionary with the string separator.

——————————————————————————————————————————————————————————————————————

### **\*str.ljust() / str.rjust()**

The string **ljust()** method returns a **left-justified** string while the string **rjust()** method returns a **right-justified** string of a given **minimum width**

SYNTAX

| string.ljust(width[, fillchar])  string.rjust(width[, fillchar]) |
| --- |

Here, fillchar is an optional parameter.

PARAMETERS

**ljust()** and **rjust()** method takes two parameters:

* **width** - width of the given string. If width is less than or equal to the length of the string, the original string is returned.
* **fillchar** (Optional) - character to fill the remaining space of the width

RETURN VALUE

The **ljust()** method returns the **left-justified** string within the given minimum width. Same case with **rjust()** but instead returns a **right-justified** string

If *fillchar* is defined, it also fills the remaining space with the defined character.

EXAMPLES

Example 1: *Left and right justify string of minimum width*

| *# example string* string = 'cat' width = 5  *# print left justified string* print(string.ljust(width))  *# print right justified string* print(string.rjust(width)) |
| --- |

Output:

| >>> cat  >>> cat |
| --- |

Here, the **minimum width** defined is **5**. So, the resultant string is of minimum length **5**.

When the string **‘cat’** is aligned to the **left**, it leaves two spaces on the **right** of the word.

When the string **‘cat’** is aligned to the **right**, it leaves two spaces on the **left** of the word.

Example 2: *Left and right justify string and fill the remaining spaces*

| *# example string* string = 'cat' width = 5 fillchar = '\*'  *# print left justified string* print(string.ljust(width, fillchar))  *# print right justified string* print(string.rjust(width, fillchar)) |
| --- |

Output:

| >>> cat\*\* >>> \*\*cat |
| --- |

Here, the string cat is aligned to the **left** and **right** respectively, and the remaining two spaces on the **right** and **left** are filled with the character \*.

——————————————————————————————————————————————————————————————————————

### **str.lower()**

*The* ***lower()*** *method converts all uppercase characters in a string into lowercase characters and returns it.*

Example:

| message = 'PYTHON IS FUN'  *# convert message to lowercase* print(message.lower()) |
| --- |

Output:

| >>> python is fun |
| --- |

RETURN VALUE

**lower()** method returns the lowercase string from the given string. It converts all uppercase characters to lowercase.

If no uppercase characters exist, it returns the original string.

EXAMPLES

Example 1: *Converting a string to lowercase*

| *# example string* string = "THIS SHOULD BE LOWERCASE!" print(string.lower())  *# string with numbers* *# all alphabets should be lowercase* string = "Th!s Sh0uLd B3 L0w3rCas3!" print(string.lower()) |
| --- |

Output:

| >>> this should be lowercase! >>> th!s sh0uld b3 l0w3rcas3! |
| --- |

Example 2: *How* ***lower()*** *is used in a program*

| *# first string* firstString = "PYTHON IS AWESOME!"  *# second string* secondString = "PyThOn Is AwEsOmE!"  if(firstString.lower() == secondString.lower()):  print("The strings are the same.") else:  print("The strings are not the same.") |
| --- |

Output:

| The strings are the same. |
| --- |

Note: If you want to convert to uppercase string, use **upper()**. You can also use **swapcase()** to swap between lowercase to uppercase.

——————————————————————————————————————————————————————————————————————

### **str.upper()**

*The* ***upper()*** *method converts all lowercase characters in a string into uppercase characters and returns it.*

Example:

| message = 'python is fun'  *# convert message to uppercase* print(message.upper()) |
| --- |

Output:

| >>> PYTHON IS FUN |
| --- |

RETURN VALUE

**upper()** method returns the uppercase string from the given string. It converts all lowercase characters to uppercase.

If no lowercase characters exist, it returns the original string.

EXAMPLES

Example 1: *Convert a string to uppercase*

| *# example string* string = "this should be uppercase!" print(string.upper())  *# string with numbers* *# all alphabets should be lowercase* string = "Th!s Sh0uLd B3 uPp3rCas3!" print(string.upper()) |
| --- |

Output:

| >>> THIS SHOULD BE UPPERCASE! >>> TH!S SH0ULD B3 UPP3RCAS3! |
| --- |

Example 2: *How* ***upper()*** *is used in a program*

| *# first string* firstString = "python is awesome!"  *# second string* secondString = "PyThOn Is AwEsOmE!"  if(firstString.upper() == secondString.upper()):  print("The strings are the same.") else:  print("The strings are not the same.") |
| --- |

Output:

| >>> The strings are the same. |
| --- |

——————————————————————————————————————————————————————————————————————

### **str.swapcase()**

*The* ***swapcase()*** *method returns the string by converting all the characters to their opposite letter case( uppercase to lowercase and vice versa).*

Example:

| name = "JoHn CeNa"  *# converts lowercase to uppercase and vice versa* print(name.swapcase()) |
| --- |

Output:

| >>> jOhN cEnA |
| --- |

RETURN VALUE

The **swapcase()** method returns:

* the string after converting its uppercase characters to lowercase, and lowercase characters to uppercase.

EXAMPLES

Example 1: *Python* ***swapcase()***

| sentence1 = "THIS SHOULD ALL BE LOWERCASE."  *# converts uppercase to lowercase* print(sentence1.swapcase())  sentence2 = "this should all be uppercase."  *# converts lowercase to uppercase* print(sentence2.swapcase())  sentence3 = "ThIs ShOuLd Be MiXeD cAsEd."  *# converts lowercase to uppercase and vice versa* print(sentence3.swapcase()) |
| --- |

Output:

| >>> this should all be lowercase. >>> THIS SHOULD ALL BE UPPERCASE. >>> tHiS sHoUlD bE mIxEd CaSeD. |
| --- |

Example 2: ***swapcase()*** *with Non-English character*

Not necessarily, string.swapcase().swapcase() == string. For example,

| text = "groß "  *# converts text to uppercase* print(text.swapcase())   *# performs swapcase() on text.swapcase()*  print(text.swapcase().swapcase())   print(text.swapcase().swapcase() == text) |
| --- |

Output:

| >>> GROSS  >>> gross  >>> False |
| --- |

In the above example, we have used the swapcase() method with the German word 'groß'. The letter 'ß' is 'ss' in English.

Here,

* text.**swapcase()** - converts 'groß' to *uppercase* i.e. 'GROSS'
* text.**swapcase()**.**swapcase()** - converts 'GROSS' to *lowercase* i.e. 'gross'

Hence the new string *'gross'* is not equal to *text (or “groß”)*.

——————————————————————————————————————————————————————————————————————

### **str.strip / lstrip / rstrip()**

*The* ***strip()*** *method returns a copy of the string by removing both the leading and the trailing characters (based on the string argument passed).*

Example:

| string = ' xoxo love xoxo '  *# Leading and trailing whitespaces are removed* print(string.strip())  *# All <whitespace>,x,o,e characters in the left* *# and right of string are removed* print(string.strip(' xoe'))  *# Argument doesn't contain space* *# No characters are removed.* print(string.strip('stx'))  string = 'android is awesome' print(string.strip('an')) |
| --- |

Output:

| >>> xoxo love xoxo >>> lov >>> xoxo love xoxo  >>> droid is awesome |
| --- |

Here, we can see that the first expression string.strip() without any arguments removed the whitespaces from the left and right of the string.

* string.**strip(' xoe')** - Removes all **whitespace**, **x**, **o**, and **e** that **lead** or **trailed** the string.
* string.**strip('stx')** - Since string has **whitespace** at the **beginning and end**, this expression **does not** change the string. x is **not removed** since it is at the **middle** of the string (whitespaces lead and trail the string)
* string.**strip('an')** - Removes ***‘an’*** leading the string.

The **lstrip()** method returns a copy of the string with leading characters removed (based on the string argument passed).

Example:

| random\_string = ' this is good '  *# Leading whitespace are removed* print(random\_string.lstrip())  *# Argument doesn't contain space* *# No characters are removed.* print(random\_string.lstrip('sti'))  print(random\_string.lstrip('s ti'))  website = 'https://www.programiz.com/' print(website.lstrip('htps:/.')) |
| --- |

Output:

| >>> this is good  >>> this is good  >>> his is good  >>> www.programiz.com/ |
| --- |

The **rstrip()** method returns a copy of the string with trailing characters removed (based on the string argument passed).

Example:

| random\_string = 'this is good ' *# Trailing whitespace are removed* print(random\_string.rstrip()) *# 'si oo' are not trailing characters so nothing is removed* print(random\_string.rstrip('si oo')) *# in 'sid oo', 'd oo' are the trailing characters, 'ood' is removed from the string* print(random\_string.rstrip('sid oo'))  website = 'www.programiz.com/'  print(website.rstrip('m/.')) |
| --- |

Output:

| >>> this is good >>> this is good >>> this is g >>> www.programiz.co |
| --- |

WORKINGS OF THE **strip()** METHOD

* When the character of the string in the left mismatches with all the characters in the ***chars*** argument, it stops removing the leading characters.
* Similarly, when the character of the string in the right mismatches with all the characters in the ***chars*** argument, it stops removing the trailing characters.

WORKINGS OF THE **lstrip()**/**rstrip()** METHOD

The **lstrip()** returns a copy of the string with **leading** characters stripped.

All combinations of characters in ***chars*** argument are **removed** from the **left** of the string **until the first mismatch**.

Same case with **rstrip()** but returns a copy with the **trailing** characters stripped and characters are **removed** from the **right** instead

——————————————————————————————————————————————————————————————————————

### **\*str.partition / rpartition()**

*The* ***partition()*** *method splits the string at the* ***first*** *occurrence of the* ***argument string*** *and returns a* ***tuple*** *containing the part* ***before the separator****,* ***argument string*** *and the part* ***after the separator****.*

Note: ***rpartition()*** method splits at the **last occurrence** instead

SYNTAX

| string.partition(separator) *#or* string.rpartition(separator) |
| --- |

PARAMETERS

Both **partition()** and **rpartition()** methods take a string separator parameter that separates the string at the **first** and **last** occurrence of it respectively.

RETURN VALUES

**partition()** and **rpartition()** methods return a 3-tuple containing:

* the part **before** the separator, **separator parameter**, and the part **after** the separator **if the separator parameter is found** in the string
* two **empty** strings, followed by the **string** itself **if the separator parameter is not found**

EXAMPLES

Example 1: ***partition()***

| string = "Python is fun"  *# 'is' separator is found* print(string.partition('is '))  *# 'not' separator is not found* print(string.partition('not '))  string = "Python is fun, isn't it"  *# splits at first occurrence of 'is'* print(string.partition('is')) |
| --- |

Output:

| >>> ('Python ', 'is ', 'fun') >>> ('Python is fun', '', '') >>> ('Python ', 'is', " fun, isn't it") |
| --- |

Example 2: ***rpartition()***

| string = "Python is fun"  *# 'is' separator is found* print(string.rpartition('is '))  *# 'not' separator is not found* print(string.rpartition('not '))  string = "Python is fun, isn't it"  *# splits at last occurrence of 'is'* print(string.rpartition('is')) |
| --- |

Output:

| >>> ('Python ', 'is ', 'fun') >>> ('', '', 'Python is fun') >>> ('Python is fun, ', 'is', "n't it") |
| --- |

### **\*str.translate()**

*The string* ***translate()*** *method returns a string where each character is* ***mapped*** *to its* ***corresponding character*** *in the* ***translation table****.*

***translate()*** *method takes the translation table to* ***replace/translate*** *characters in the given string* ***as per the mapping table****.*

*The translation table is created by the static method* ***maketrans()****.*

SYNTAX

| string.translate(table) |
| --- |

PARAMETERS

**translate()** method takes a single parameter:

* table - a translation table containing the mapping between two characters; usually created by maketrans()

RETURN VALUE

**translate()** method returns a **string** where each character is **mapped** to its **corresponding character** as per the translation table.

EXAMPLES

Example 1: *Translation/Mapping using a* ***translation table*** *with* ***translate()***

| *# first string* firstString = "abc" secondString = "ghi" thirdString = "ab"  string = "abcdef" print("Original string:", string)  translation = string.maketrans(firstString, secondString, thirdString)  *# translate string* print("Translated string:", string.translate(translation)) |
| --- |

Output:

| >>> Original string: abcdef >>> Translated string: idef |
| --- |

Here, the translation mapping translation contains the mapping from ***a***, ***b*** and ***c*** to ***g***, ***h*** and ***i*** respectively.

However, the removal string thirdString resets the mapping of ***a*** and ***b*** to ***None***.

Thus, when the string is translated using translate(), ***a*** and ***b*** are removed, and ***c*** replaced ***i*** outputting idef.

Note: If you cannot understand what's going inside **maketrans()**, please refer to **str.maketrans()**

Example 2: Translation/Mapping with translate() with manual translation table

| *# translation table - a dictionary* translation = {97: None, 98: None, 99: 105}  string = "abcdef" print("Original string:", string)  *# translate string* print("Translated string:", string.translate(translation)) |
| --- |

Output:

| >>> Original string: abcdef >>> Translated string: idef |
| --- |

Here, we don't create a translation table from **maketrans()** and we manually create the mapping dictionary for translation instead.

This translation is then used to translate the string to get the same output as the previous example.

——————————————————————————————————————————————————————————————————————

### **\*str.maketrans()**

The **maketrans()** method returns a mapping table for translation usable by the **translate()** method.

In simple terms, **maketrans()** method is a static method that creates a **one to one mapping** ofa **character to its translation/replacement**.

It creates a **Unicode** representation of each character for translation.

This translation mapping is then used for replacing a character to its mapped character when used in the **translate()** method.

SYNTAX

| string.maketrans(x[, y[, z]]) |
| --- |

Here, y and z are optional arguments.

PARAMETERS

**maketrans()** method takes **3** parameters:

* **x** - If only **one** argument is supplied, it **must be a dictionary**.

The dictionary should contain a 1-to-1 mapping from a single character string to its translation OR a Unicode number (97 for 'a') to its translation.

* **y** - If **two** arguments are passed, it **must be two strings** with **equal length**.

Each character in the first string is a replacement to its corresponding index in the second string.

* **z** - If **three** arguments are passed, **each** character in the third argument is mapped to ***None***.

RETURN VALUES

The **maketrans()** method returns a translation table with a 1-to-1 mapping of a Unicode ordinal to its translation/replacement.

EXAMPLES

Example 1: *Translation table using a dictionary with* ***maketrans()***

| *# example dictionary* dict = {"a": "123", "b": "456", "c": "789"} string = "abc" print(string.maketrans(dict))  *# example dictionary* dict = {97: "123", 98: "456", 99: "789"} string = "abc" print(string.maketrans(dict)) |
| --- |

Output:

| >>> {97: '123', 98: '456', 99: '789'} >>> {97: '123', 98: '456', 99: '789'} |
| --- |

Here, a dictionary dict is defined. It contains a mapping of characters **a**,**b** and **c** to **123**, **456**, and **789** respectively.

**maketrans()** creates a mapping of the character's Unicode ordinal to its corresponding translation.

So, 97 (**'a'**) is mapped to **'123'**, 98 **'b'** to **456** and 99 **'c'** to **789**. This can be demonstrated from the output of both dictionaries.

Note: **If two or more** characters are mapped in the dictionary, **it raises an exception**.

Example 2: *Translation table using two strings with* ***maketrans()***

| *# first string* firstString = "abc" secondString = "def" string = "abc" print(string.maketrans(firstString, secondString))  *# example dictionary* firstString = "abc" secondString = "defghi" string = "abc" print(string.maketrans(firstString, secondString)) |
| --- |

Output:

| >>> {97: 100, 98: 101, 99: 102} >>> ValueError: the first two maketrans arguments must have equal  length |
| --- |

Here first, two strings of **equal length** abc and def are defined. And the corresponding translation is created.

Printing only the first translation gives you a **1-to-1 mapping** to each character's Unicode ordinal in firstString to the same indexed character on secondString.

In this case, 97 (**'a'**) is mapped to 100 (**'d'**), 98 (**'b'**) to 101 (**'e'**) and 99 (**'c'**) to 102 (**'f'**).

Trying to create a translation table for **unequal length strings** raises a ***ValueError*** **exception** indicating that the strings **must have equal length**.

Example 3: *Translational table with removable string with* ***maketrans()***

| *# first string* firstString = "abc" secondString = "def" thirdString = "abd" string = "abc" print(string.maketrans(firstString, secondString, thirdString)) |
| --- |

Output:

| >>> {97: None, 98: None, 99: 102, 100: None} |
| --- |

Here, first, the mapping **between** the two strings **firstString** and **secondString** are created.

Then, the third argument thirdString **resets** the mapping of **each character** in it to **None** and also creates a **new mapping for non-existent characters**.

In this case, thirdString **resets** the mapping of 97 (**'a'**) and 98 (**'b'**) to ***None***, and also creates a new mapping for 100 (**'d'**) mapped to ***None***.

### **str.replace()**

*The* ***replace()*** *method replaces each matching occurrence of a substring with another string.*

Example:

| text = 'bat ball'  *# replace 'ba' with 'ro'* replaced\_text = text.replace('ba', 'ro') print(replaced\_text) |
| --- |

Output:

| >>> rot roll |
| --- |

SYNTAX

| str.replace(old, new [, count]) |
| --- |

PARAMETERS

The **replace()** method can take a maximum of three arguments:

* **old** - the old substring we want to replace
* **new** - new substring which will replace the old substring
* **count** (optional) - the number of times you want to replace the old substring with the new string

Note: If count is not specified, the **replace()** method **replaces all occurrences** of the ***old*** substring with the ***new*** string.

EXAMPLES

Example 1: *Using* ***replace()***

| song = 'cold, cold heart'  *# replacing 'cold' with 'hurt'* print(song.replace('cold', 'hurt'))  song = 'Let it be, let it be, let it be, let it be'  *# replacing only two occurrences of 'let'* print(song.replace('let', "don't let", 2)) |
| --- |

Output:

| >>> hurt, hurt heart >>> Let it be, don't let it be, don't let it be, let it be |
| --- |

Example 2: *More Examples on String* ***replace()***

| song = 'cold, cold heart' replaced\_song = song.replace('o', 'e')  *# The original string is unchanged* print('Original string:', song)  print('Replaced string:', replaced\_song)  song = 'let it be, let it be, let it be'  *# maximum of 0 substring is replaced* *# returns copy of the original string* print(song.replace('let', 'so', 0)) |
| --- |

Output:

| >>> Original string: cold, cold heart >>> Replaced string: celd, celd heart >>> let it be, let it be, let it be |
| --- |

### **str.split() / str.rsplit()**

The split() method splits a string from the **left** at the specified separator and returns a list of substrings.

The rsplit() method splits a string from the **right** at the specified separator and returns a list of substrings.

SYNTAX

| str.split([separator [, maxsplit]])  str.rsplit([separator [, maxsplit]]) |
| --- |

PARAMETERS

Both the **split()** and **rsplit()** method takes a maximum of 2 parameters:

* **separator** (optional)- Delimiter at which splits occur. If not provided, the string is splitted at whitespaces.
* **maxsplit** (optional) - Maximum number of splits. If not provided, there is no limit on the number of splits.

RETURN VALUE

Both the **split()** and **rsplit()** method returns a list of strings.

EXAMPLES

Example 1: *How* ***split()*** *works in Python?*

| text= 'Love thy neighbor'  *# splits at space* print(text.split())  grocery = 'Milk, Chicken, Bread'  *# splits at ','* print(grocery.split(', '))  *# Splits at ':'* print(grocery.split(':')) |
| --- |

Output:

| >>> ['Love', 'thy', 'neighbor'] >>> ['Milk', 'Chicken', 'Bread'] >>> ['Milk, Chicken, Bread'] |
| --- |

Here,

* text.**split()** - splits string into a list of substrings at each space character
* grocery.**split(', ')** - splits string into a list of substrings at each comma and space character
* grocery.**split(':')** - since there are no colons in the string, split() does not split the string.

Example 2: *How* ***split()*** *works when* ***maxsplit*** *is specified?*

The maxsplit parameter is an optional parameter that can be used with the split() method in Python.

It specifies the maximum number of splits to be performed on a string.

| grocery = 'Milk, Chicken, Bread, Butter'  *# maxsplit: 2* print(grocery.split(', ', 2))  *# maxsplit: 1* print(grocery.split(', ', 1))  *# maxsplit: 5* print(grocery.split(', ', 5))  *# maxsplit: 0* print(grocery.split(', ', 0)) |
| --- |

Output:

| >>> ['Milk', 'Chicken', 'Bread, Butter'] >>> ['Milk', 'Chicken, Bread, Butter'] >>> ['Milk', 'Chicken', 'Bread', 'Butter'] >>> ['Milk, Chicken, Bread, Butter'] |
| --- |

If ***maxsplit*** is specified, the list will have a **maximum** of ***maxsplit*+1** items.

Example 3: *How* ***rsplit()*** *works when maxsplit is specified?*

| grocery = 'Milk, Chicken, Bread, Butter'  *# maxsplit: 2* print(grocery.rsplit(', ', 2))  *# maxsplit: 1* print(grocery.rsplit(', ', 1))  *# maxsplit: 5* print(grocery.rsplit(', ', 5))  *# maxsplit: 0* print(grocery.rsplit(', ', 0)) |
| --- |

Output:

| >>> ['Milk, Chicken', 'Bread', 'Butter'] >>> ['Milk, Chicken, Bread', 'Butter'] >>> ['Milk', 'Chicken', 'Bread', 'Butter'] >>> ['Milk, Chicken, Bread, Butter'] |
| --- |

### **\*str.splitlines()**

*The* ***splitlines()*** *method splits the string at line breaks and returns a list...*

Example:

| *# \n is a line boundary*  sentence = 'I\nlove\nPython\nProgramming.'  *# returns a list after splitting string at line breaks*  resulting\_list = sentence.splitlines()  print(resulting\_list) |
| --- |

Output:

| >>> ['I', 'love', 'Python', 'Programming.'] |
| --- |

SYNTAX

| string.splitlines([keepends]) |
| --- |

Here, *keepends* can be **True** or **any number**.

PARAMETERS

The **splitlines()** method can take a single parameter:

* **keepends**(optional) - it determines whether line breaks are included in the resulting list or not. It's value can be True or any number.

RETURN VALUE

The splitlines() method returns:

* a list of lines in the string.

If there are no line break characters, it returns a list with a single item (a single line).

| Representation Description \n Line Feed \r Carriage Return \r\n Carriage Return + Line Feed \v or \x0b Line Tabulation \f or \x0c Form Feed \x1c File Separator \x1d Group Separator \x1e Record Separator \x85 Next Line (C1 Control Code) \u2028 Line Separator \u2029 Paragraph Separator |
| --- |

EXAMPLES

Example 1: *Python String* ***splitlines()***

| *# '\n' is a line break*  grocery = 'Milk\nChicken\nBread\rButter'  *# returns a list after splitting the grocery string*  print(grocery.splitlines()) |
| --- |

Output:

| ['Milk', 'Chicken', 'Bread', 'Butter'] |
| --- |

In the above example, we have used the **splitlines()** method to split the grocery string i.e. 'Milk\nChicken\r\nBread\rButter' at the line breaks.

Here, grocery.**splitlines()** splits grocery at line break '\n' and returns a list '['Milk', 'Chicken', 'Bread', 'Butter']' after removing the line break.

Example 2: ***splitlines()*** *with Multi Line String*

| *# multi line string*  grocery = '''Milk Chicken Bread Butter'''  *# returns a list after splitting the grocery string* print(grocery.splitlines()) |
| --- |

Output:

| ['Milk', 'Chicken', 'Bread', 'Butter'] |
| --- |

Here, the **splitlines()** method splits the multi line string grocery and returns the list ['Milk', 'Chicken', 'Bread', 'Butter'].

Example 3: *Passing Boolean Value in* ***splitlines()***

| grocery = 'Milk\nChicken\nBread\rButter'  *# returns a list including line breaks*  resulting\_list1 = grocery.splitlines(True) print(resulting\_list1)  *# returns a list without including line breaks*  resulting\_list2 = grocery.splitlines(False) print(resulting\_list2) |
| --- |

Output:

| ['Milk\n', 'Chicken\n', 'Bread\r', 'Butter'] ['Milk', 'Chicken', 'Bread', 'Butter'] |
| --- |

In the above example, we have passed Boolean values True and False in the **splitlines()** method to split 'Milk\nChicken\nBread\rButter'.

Here, in the method on passing:

* True - returns a list including linebreaks in all items i.e. '['Milk\n', 'Chicken\n', 'Bread\r', 'Butter']'
* False - returns a list without including linebreaks in the items i.e. ['Milk', 'Chicken', 'Bread', 'Butter']

Example 4: *Passing Number in* ***splitlines()***

The **splitlines()** method takes an integer value as a parameter. Here, 0 represents True and other positive or negative numbers indicate False.

| grocery = 'Milk\nChicken\nBread\rButter'  *# returns list including line breaks*  resulting\_list1 = grocery.splitlines(0) print(resulting\_list1)  *# returns list without including line breaks*  resulting\_list2 = grocery.splitlines(5) print(resulting\_list2) |
| --- |

Output:

| ['Milk\n', 'Chicken\n', 'Bread\r', 'Butter'] ['Milk', 'Chicken', 'Bread', 'Butter'] |
| --- |

——————————————————————————————————————————————————————————————————————

### **str.startswith()**

*The* ***startswith()*** *method returns* ***True*** *if a string* ***starts with the specified prefix****(string).* ***If not****, it returns* ***False****.*

Example:

| message = 'Python is fun'  *# check if the message starts with Python* print(message.startswith('Python')) |
| --- |

Output:

| >>> True |
| --- |

SYNTAX

| str.startswith(prefix[, start[, end]]) |
| --- |

PARAMETERS

**startswith()** method takes a maximum of three parameters:

* **prefix** - String or tuple of strings to be checked
* **start** (optional) - Beginning position where prefix is to be checked within the string.
* **end** (optional) - Ending position where prefix is to be checked within the string.

RETURN VALUE

**startswith()** method returns a **boolean**.

* It returns **True** if the string **starts** with the specified prefix.
* It returns **False** if the string **doesn't start** with the specified prefix.

EXAMPLES

Example 1:***startswith()*** *With start and end Parameters*

| text = "Python programming is easy."  *# start parameter: 7* *# 'programming is easy.' string is searched* result = text.startswith('programming is', 7) print(result)  *# start: 7, end: 18* *# 'programming' string is searched* result = text.startswith('programming is', 7, 18) print(result)  result = text.startswith('program', 7, 18) print(result) |
| --- |

Output:

| >>> True >>> False >>> True |
| --- |

**Passing Tuple to startswith()**

It's possible to pass a **tuple of prefixes** to the **startswith()** method in Python.

If the string starts with **any** item of the **tuple**, **startswith()** returns **True**. If not, it returns **False**

Example 3: ***startswith()*** *With Tuple Prefix*

| text = "programming is easy" result = text.startswith(('python', 'programming'))  *# prints True* print(result)  result = text.startswith(('is', 'easy', 'java'))  *# prints False* print(result)  *# With start and end parameter* *# 'is easy' string is checked* result = text.startswith(('programming', 'easy'), 12, 19)  *# prints False* print(result) |
| --- |

Output:

| >>> True >>> False >>> False |
| --- |

——————————————————————————————————————————————————————————————————————

### **str.title()**

*The* ***title()*** *method returns a string with the first letter of each word capitalized; a title cased string.*

Example:

| text = 'My favorite number is 25.' print(text.title())  text = '234 k3l2 \*43 fun' print(text.title()) |
| --- |

Output:

| >>> My Favorite Number Is 25. >>> 234 K3L2 \*43 Fun |
| --- |

RETURN VALUE

**title()** method returns a title cased version of the string. Meaning, the first character of each word is capitalized (if the first character is a letter).

EXAMPLES

Example 2: ***title()*** *with apostrophes*

| text = "He's an engineer, isn't he?" print(text.title()) |
| --- |

Output:

| >>> He'S An Engineer, Isn'T He? |
| --- |

However, **title()** capitalizes the **first letter after apostrophes** as well.

To solve this issue, you can use **regex** as follows:

| import re  def titlecase(s):  return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",  lambda mo: mo.group(0)[0].upper() +  mo.group(0)[1:].lower(),  s)  text = "He's an engineer, isn't he?" print(titlecase(text)) |
| --- |

Output:

| >>> He's An Engineer, Isn't He? |
| --- |

——————————————————————————————————————————————————————————————————————

### **\*str.zfill()**

*The* ***zfill()*** *method returns a copy of the string with the '0' character padded to the left.*

SYNTAX

| str.zfill(width) |
| --- |

PARAMETERS

**zfill()** takes a single character ***width***.

The width specifies the length of the returned string from **zfill()** with digit **0** filled to the left.

RETURN VALUE

**zfill()** returns a copy of the string with 0 filled to the left. The length of the returned string depends on the ***width*** provided.

* Suppose, the initial length of the string is 10. And, the ***width*** is specified 15. In this case, **zfill()** returns a copy of the string with five '0' digits filled to the left.
* Suppose, the initial length of the string is 10. And, the ***width*** is specified 8. In this case, **zfill()** doesn't fill '0' digits to the left and returns a copy of the original string. The length of the returned string in this case will be 10.

EXAMPLES

Example 1: *How* ***zfill()*** *works in Python?*

| text = "program is fun" print(text.zfill(15)) print(text.zfill(20)) print(text.zfill(10)) |
| --- |

Output:

| >>> 0program is fun >>> 000000program is fun >>> program is fun |
| --- |

However, if a string starts with the sign prefix ('+', '-'), the digit 0 is filled **after** the **first sign prefix** character.

| number = "-290" print(number.zfill(8))  number = "+290" print(number.zfill(8))  text = "--random+text" print(text.zfill(20)) |
| --- |

Output:

| >>> -0000290 >>> +0000290 >>> -0000000-random+text |
| --- |

### **\*str.format\_map()**

*The* ***format\_map()*** *method is* ***similar*** *to str.****format(\*\*mapping)*** *except that str.****format(\*\*mapping)*** *creates a* ***new dictionary*** *whereas str.****format\_map(mapping) doesn't****.*

Before talking about **format\_map()**, let's see how str.**format(\*\*mapping)** works for Python **Dictionaries**.

| point = {'x':4,'y':-5} print('{x} {y}'.format(\*\*point)) |
| --- |

Output:

| >>> 4 -5 |
| --- |

The **format\_map(mapping)** is similar to str.**format(\*\*mapping)** method.

The only difference is that str.**format(\*\*mapping)** copies the **dictionary** whereas str.**format\_map(mapping)** makes a **new** dictionary during method call. This can be useful **if you are working with a dict subclass**.

SYNTAX

| str.format\_map(mapping) |
| --- |

PARAMETER

**format\_map()** takes a single argument mapping(dictionary).

RETURN VALUE

**format\_map()** formats the given string and returns it.

EXAMPLES

Example 1: *How* ***format\_map()*** *works*

| point = {'x':4,'y':-5} print('{x} {y}'.format\_map(point))  point = {'x':4,'y':-5, 'z': 0} print('{x} {y} {z}'.format\_map(point)) |
| --- |

Output:

| >>> 4 -5 >>> 4 -5 0 |
| --- |

Example 2: *How* ***format\_map()*** *works with dict subclass*

| class Coordinate(dict):  def \_\_missing\_\_(self, key):  return key  print('({x}, {y})'.format\_map(Coordinate(x='6'))) print('({x}, {y})'.format\_map(Coordinate(y='5'))) print('({x}, {y})'.format\_map(Coordinate(x='6', y='5'))) |
| --- |

Output:

| >>> (6, y) >>> (x, 5) >>> (6, 5) |
| --- |

## 1.5 Lists

Lists are used to contain a list of values. They are denoted by square brackets. ([]). Commas are used to separate the values within a list. One list can contain a mix of different data types, as shown below:

| List = [5.0, 4, "f"] *#contains float, integer, string* |
| --- |

Although, it is generally good practice to only stick to one data type per list.

Examples:

| List\_1 = [1,2,3,4,5,6] List\_2 = [5.0, 9.8, 5.3] List\_3 = ["H","E","L","L","O",1,2,3] |
| --- |

Lists also can contain lists.

| BigList = [[1,2,3],["A","B","C"]] List1 = ["H","E","L","L","O"] List2 = ["W","O","R","L","D"] List3 = [List1,List2] |
| --- |

## 1.5.1 List Methods

### **list.index()**

*The* ***index()*** *method returns the index of the specified element in the list.*

Example

| animals = ['cat', 'dog', 'rabbit', 'horse']  *# get the index of 'dog'* index = animals.index('dog')   print(index) |
| --- |

Output

| >>> 1 |
| --- |

SYNTAX

| list.index(element, start, end) |
| --- |

PARAMETERS

The list **index()** method can take a maximum of three arguments:

* **element** - the element to be searched
* **start** (optional) - start searching from this index
* **end** (optional) - search the element up to this index

RETURN VALUES

* The **index()** method returns the index of the given element in the list.
* If the element is not found, a ValueError exception is raised.

Note: The **index()** method only returns the first occurrence of the matching element.

EXAMPLES

Example 1: *Find the* ***index*** *of the element*

| *# vowels list* vowels = ['a', 'e', 'i', 'o', 'i', 'u']  *# index of 'e' in vowels* index = vowels.index('e')  print('The index of e:', index)  *# element 'i' is searched* *# index of the first 'i' is returned* index = vowels.index('i')   print('The index of i:', index) |
| --- |

Output

| >>> The index of e: 1 >>> The index of i: 2 |
| --- |

Example 2: *Index of the element* ***not*** *present in the List*

| *# vowels list* vowels = ['a', 'e', 'i', 'o', 'u']  *# index of 'p' is vowels* index = vowels.index('p')  print('The index of p:', index) |
| --- |

Output

| >>> ValueError: 'p' is not in list |
| --- |

Example 3: ***index()*** *with Start and End Parameters*

| *# alphabets list* alphabets = ['a', 'e', 'i', 'o', 'g', 'l', 'i', 'u']  *# index of 'i' in alphabets* index = alphabets.index('e') *# 1*  print('The index of e:', index)  *# 'i' after the 4th index is searched* index = alphabets.index('i', 4) *# 6*  print('The index of i:', index)  *# 'i' between 3rd and 5th index is searched* index = alphabets.index('i', 3, 5) *# Error!*  print('The index of i:', index) |
| --- |

Output

| >>> The index of e: 1 >>> The index of i: 6 >>> Traceback (most recent call last):  File "\*lt;string>", line 13, in   ValueError: 'i' is not in list |
| --- |

### **list.append()**

*The* ***append()*** *method adds an item to the end of the list.*

Example

| currencies = ['Dollar', 'Euro', 'Pound']  *# append 'Yen' to the list* currencies.append('Yen')  print(currencies) |
| --- |

Output

| >>> ['Dollar', 'Euro', 'Pound', 'Yen'] |
| --- |

SYNTAX

| list.append(item) |
| --- |

PARAMETERS

**append()** takes a single argument

* **item** - an item (number, string, list etc.) to be added at the end of the list

EXAMPLES

Example 1: *Adding Elements to a List*

| *# animals list* animals = ['cat', 'dog', 'rabbit']  *# Add 'guinea pig' to the list* animals.append('guinea pig')  print('Updated animals list: ', animals) |
| --- |

Output

| >>> Updated animals list: ['cat', 'dog', 'rabbit', 'guinea pig'] |
| --- |

Example 2: *Adding a List to a List*

| *# animals list* animals = ['cat', 'dog', 'rabbit']  *# list of wild animals* wild\_animals = ['tiger', 'fox']  *# appending wild\_animals list to animals* animals.append(wild\_animals)  print('Updated animals list: ', animals) |
| --- |

Output

| >>> Updated animals list: ['cat', 'dog', 'rabbit', ['tiger', 'fox']] |
| --- |

Note: If you need to add **items of a list** (rather than the list itself) to **another list**, use the **extend()** method.

### **list.extend()**

*The* ***extend()*** *method adds all the* ***elements of an iterable*** *(list, tuple, string etc.) to the* ***end*** *of the* ***list****.*

Example

| *# create a list* prime\_numbers = [2, 3, 5]  *# create another list* numbers = [1, 4]  *# add all elements of prime\_numbers to numbers* numbers.extend(prime\_numbers)   print('List after extend():', numbers) |
| --- |

Output:

| >>> List after extend(): [1, 4, 2, 3, 5] |
| --- |

SYNTAX

| list1.extend(iterable) |
| --- |

PARAMETERS

As mentioned, the **extend()** method takes an **iterable** such as **list, tuple, string** etc.

EXAMPLE

Example 1: *Using* ***extend()*** *Method*

| *# languages list* languages = ['French', 'English']  *# another list of language* languages1 = ['Spanish', 'Portuguese']  *# appending language1 elements to language* languages.extend(languages1)   print('Languages List:', languages) |
| --- |

Output

| >>> Languages List: ['French', 'English', 'Spanish', 'Portuguese'] |
| --- |

Example 2: *Add Elements of Tuple and Set to List*

| *# languages list* languages = ['French']  *# languages tuple* languages\_tuple = ('Spanish', 'Portuguese')  *# languages set* languages\_set = {'Chinese', 'Japanese'}  *# appending language\_tuple elements to language* languages.extend(languages\_tuple) print('New Language List:', languages)  *# appending language\_set elements to language* languages.extend(languages\_set) print('Newer Languages List:', languages) |
| --- |

Output

| >>> New Languages List: ['French', 'Spanish', 'Portuguese'] >>> Newer Languages List: ['French', 'Spanish', 'Portuguese', 'Japanese', 'Chinese'] |
| --- |

Other Ways to Extend a List

You can also append all elements of an iterable to the list using:

1. the + operator

| a = [1, 2] b = [3, 4]  a += b *# a = a + b*   *# Output: [1, 2, 3, 4]* print('a =', a) |
| --- |

Output

| >>> a = [1, 2, 3, 4] |
| --- |

2. the list slicing syntax

| a = [1, 2] b = [3, 4]  a[len(a):] = b   *# Output: [1, 2, 3, 4]* print('a =', a) |
| --- |

Output

| >>> a = [1, 2, 3, 4] |
| --- |

Python **extend()** vs **append()**

If you need to add an **element to the end of a list**, you can use the **append()** method.

| a1 = [1, 2] a2 = [1, 2] b = (3, 4)  *# a1 = [1, 2, 3, 4]* a1.extend(b)  print(a1)  *# a2 = [1, 2, (3, 4)]* a2.append(b) print(a2) |
| --- |

Output

| >>> [1, 2, 3, 4] >>> [1, 2, (3, 4)] |
| --- |

To learn more, visit list **append()** method.

### **list.insert()**

*The* ***insert()*** *method inserts an* ***element*** *to the list* ***at the specified index****.*

Example

| *# create a list of vowels* vowel = ['a', 'e', 'i', 'u']  *# 'o' is inserted at index 3 (4th position)* vowel.insert(3, 'o')  print('List:', vowel) |
| --- |

Output

| >>> List: ['a', 'e', 'i', 'o', 'u'] |
| --- |

SYNTAX

| list.insert(i, elem) |
| --- |

Here, elem is inserted to the list at the ith index. All the elements after elem are shifted to the right.

PARAMETERS

The **insert()** method takes two parameters:

* **index** - the index where the element needs to be inserted
* **element** - this is the element to be inserted in the list

Notes:

* If index is 0, the element is inserted at the beginning of the list.
* If index is 3, the index of the inserted element will be 3 (4th element in the list).

EXAMPLES

Example 1: *Inserting an Element to the List*

| *# create a list of prime numbers* prime\_numbers = [2, 3, 5, 7]  *# insert 11 at index 4* prime\_numbers.insert(4, 11)   print('List:', prime\_numbers) |
| --- |

Output

| >>> List: [2, 3, 5, 7, 11] |
| --- |

Example 2: *Inserting a Tuple (as an Element) to the List*

| mixed\_list = [{1, 2}, [5, 6, 7]]  *# number tuple* number\_tuple = (3, 4)  *# inserting a tuple to the list* mixed\_list.insert(1, number\_tuple)   print('Updated List:', mixed\_list) |
| --- |

Output

| >>> Updated List: [{1, 2}, (3, 4), [5, 6, 7]] |
| --- |

### **list.remove()**

*The* ***remove()*** *method removes the first matching element (which is passed as an argument) from the list.*

Example

| *# create a list* prime\_numbers = [2, 3, 5, 7, 9, 11]  *# remove 9 from the list* prime\_numbers.remove(9)   *# Updated prime\_numbers List* print('Updated List: ', prime\_numbers) |
| --- |

Output

| >>> Updated List: [2, 3, 5, 7, 11] |
| --- |

SYNTAX

| list.remove(element) |
| --- |

PARAMETERS

* The **remove()** method takes a single element as an argument and removes it from the list.
* If the **element** doesn't exist, it throws **ValueError: list.remove(x): x not in list** exception.

EXAMPLES

Example 1: *Remove element from the list*

| *# animals list* animals = ['cat', 'dog', 'rabbit', 'guinea pig']  *# 'rabbit' is removed* animals.remove('rabbit')   *# Updated animals List* print('Updated animals list: ', animals) |
| --- |

Output

| >>> Updated animals list: ['cat', 'dog', 'guinea pig'] |
| --- |

Example 2: ***remove()*** *method on a list having duplicate elements*

If a list contains duplicate elements, the **remove()** method only removes the first matching element.

| *# animals list* animals = ['cat', 'dog', 'dog', 'guinea pig', 'dog']  *# 'dog' is removed* animals.remove('dog')  *# Updated animals list* print('Updated animals list: ', animals) |
| --- |

Output

| Updated animals list: ['cat', 'dog', 'guinea pig', 'dog'] |
| --- |

Here, only the first occurrence of element 'dog' is removed from the list.

Example 3: *Deleting an element that doesn't exist*

| *# animals list* animals = ['cat', 'dog', 'rabbit', 'guinea pig']  *# Deleting 'fish' element* animals.remove('fish')   *# Updated animals List* print('Updated animals list: ', animals) |
| --- |

Output

| >>> Traceback (most recent call last):  File ".. .. ..", line 5, in <module>  animal.remove('fish')  ValueError: list.remove(x): x not in list |
| --- |

Here, we are getting an error because the animals list doesn't contain 'fish'.

If you need to delete elements based on the index (like the fourth element), you can use the **pop()** method (counterpart of **insert()**).

Also, you can use the **Python del statement** to remove items from the list.

### **list.count()**

*The* ***count()*** *method returns the number of times the specified element appears in the list.*

Example

| *# create a list* numbers = [2, 3, 5, 2, 11, 2, 7]  *# check the count of 2* count = numbers.count(2)   print('Count of 2:', count) |
| --- |

Output

| >>> Count of 2: 3 |
| --- |

SYNTAX

| list.count(element) |
| --- |

PARAMETERS

The **count()** method takes a single argument:

* **element** - the element to be counted

RETURN VALUE

The **count()** method returns the number of times the element appears within the list.

EXAMPLES

Example 1: *Use of count()*

| *# vowels list* vowels = ['a', 'e', 'i', 'o', 'i', 'u']  *# count element 'i'* count = vowels.count('i')   *# print count* print('The count of i is:', count)  *# count element 'p'* count = vowels.count('p')   *# print count* print('The count of p is:', count) |
| --- |

Output

| >>> The count of i is: 2 >>> The count of p is: 0 |
| --- |

Example 2: *Count Tuple and List Elements Inside List*

| *# random list* random = ['a', ('a', 'b'), ('a', 'b'), [3, 4]]  *# count element ('a', 'b')* count = random.count(('a', 'b'))  *# print count* print("The count of ('a', 'b') is:", count)  *# count element [3, 4]* count = random.count([3, 4])  *# print count* print("The count of [3, 4] is:", count) |
| --- |

Output

| >>> The count of ('a', 'b') is: 2 >>> The count of [3, 4] is: 1 |
| --- |

### **list.pop()**

*The* ***pop()*** *method removes the item at the given index from the list and returns the removed item.*

Example

| *# create a list of prime numbers* prime\_numbers = [2, 3, 5, 7]  *# remove the element at index 2* removed\_element = prime\_numbers.pop(2)  print('Removed Element:', removed\_element) print('Updated List:', prime\_numbers) |
| --- |

Output

| >>> Removed Element: 5 >>> Updated List: [2, 3, 7] |
| --- |

SYNTAX

| list.pop(index) |
| --- |

PARAMETERS

The **pop()** method takes a single argument (index).

* The argument passed to the method is **optional**. If not passed, the **default index -1** is passed as an argument (index of the **last** item).
* If the index passed to the method is not in range, it throws **IndexError: pop index out of range** exception.

RETURN VALUES

The **pop()** method returns the item present at the given index. This item is also removed from the list.

EXAMPLES

Example 1: *Pop item at the given index from the list*

| *# programming languages list* languages = ['Python', 'Java', 'C++', 'French', 'C']  *# remove and return the 4th item* return\_value = languages.pop(3)  print('Return Value:', return\_value)  *# Updated List* print('Updated List:', languages) |
| --- |

Output

| >>> Return Value: French >>> Updated List: ['Python', 'Java', 'C++', 'C'] |
| --- |

Note: Index in Python starts from 0, not 1.

If you need to pop the 4th element, you need to pass 3 to the **pop()** method.

Example 2: ***pop()*** *without an index, and for negative indices*

| *# programming languages list* languages = ['Python', 'Java', 'C++', 'Ruby', 'C']  *# remove and return the last item* print('When index is not passed:')  print('Return Value:', languages.pop())  print('Updated List:', languages)  *# remove and return the last item* print('\nWhen -1 is passed:')  print('Return Value:', languages.pop(-1))  print('Updated List:', languages)  *# remove and return the third last item* print('\nWhen -3 is passed:')  print('Return Value:', languages.pop(-3))  print('Updated List:', languages) |
| --- |

Output

| >>> When index is not passed: >>> Return Value: C >>> Updated List: ['Python', 'Java', 'C++', 'Ruby'] >>>  When -1 is passed: >>> Return Value: Ruby >>> Updated List: ['Python', 'Java', 'C++'] >>>  When -3 is passed: >>> Return Value: Python >>> Updated List: ['Java', 'C++'] |
| --- |

If you need to remove the **given item** from the list, you can use the **remove()** method.

And, you can use the **del statement** to remove an **item or slices** from the list.

### **list.reverse**

*The* ***reverse()*** *method reverses the elements of the list.*

Example

| *# create a list of prime numbers* prime\_numbers = [2, 3, 5, 7]  *# reverse the order of list elements* prime\_numbers.reverse()   print('Reversed List:', prime\_numbers) |
| --- |

Output

| >>> Reversed List: [7, 5, 3, 2] |
| --- |

SYNTAX

| list.reverse() |
| --- |

EXAMPLES

Example 1: *Reverse a List*

| *# Operating System List* systems = ['Windows', 'macOS', 'Linux'] print('Original List:', systems)  *# List Reverse* systems.reverse()   *# updated list* print('Updated List:', systems) |
| --- |

Output

| >>> Original List: ['Windows', 'macOS', 'Linux'] >>> Updated List: ['Linux', 'macOS', 'Windows'] |
| --- |

Example 2: *Reverse a List Using Slicing Operator*

| *# Operating System List* systems = ['Windows', 'macOS', 'Linux'] print('Original List:', systems)  *# Reversing a list*  *# Syntax: reversed\_list = systems[start:stop:step]*  reversed\_list = systems[::-1]   *# updated list* print('Updated List:', reversed\_list) |
| --- |

Output

| >>> Original List: ['Windows', 'macOS', 'Linux'] >>> Updated List: ['Linux', 'macOS', 'Windows'] |
| --- |

Example 3: *Accessing Elements in Reversed Order*

If you need to access individual elements of a list in the reverse order, it's better to use the **reversed()** function.

| *# Operating System List* systems = ['Windows', 'macOS', 'Linux']  *# Printing Elements in Reversed Order* for o in reversed(systems):  print(o) |
| --- |

Output

| >>> Linux >>> macOS >>> Windows |
| --- |

### **\*list.sort()**

*The* ***sort()*** *method sorts the items of a list in ascending or descending order.*

Examples

| prime\_numbers = [11, 3, 7, 5, 2]  *# sorting the list in ascending order* prime\_numbers.sort()  print(prime\_numbers) |
| --- |

Output

| >>> [2, 3, 5, 7, 11] |
| --- |

SYNTAX

| list.sort(key=..., reverse=...) |
| --- |

Alternatively, you can also use Python's built-in sorted() function for the same purpose.

| sorted(list, key=..., reverse=...) |
| --- |

Note: The simplest difference between **sort()** and **sorted()** is: **sort()** **changes the list directly** and **doesn't return any value**, while **sorted() doesn't change the list** and **returns the sorted list**.

PARAMETERS

By default, **sort()** doesn't require any extra parameters. However, it has two optional parameters:

* **reverse** - If **True**, the sorted list is **reversed** (or sorted in **Descending order**)
* **key** - function that serves as a key for the **sort comparison**

RETURN VALUE

The **sort()** method **doesn't return any value**. Rather, it **changes the original list**.

If you want a function to **return the sorted list** **rather than** **change the original list**, use **sorted()**.

EXAMPLES

Example 1: *Sort a given list*

| *# vowels list* vowels = ['e', 'a', 'u', 'o', 'i']  *# sort the vowels* vowels.sort()  *# print vowels* print('Sorted list:', vowels) |
| --- |

Output

| >>> Sorted list: ['a', 'e', 'i', 'o', 'u'] |
| --- |

Sort in Descending order

The **sort()** method accepts a **reverse parameter** as an **optional argument**.

Setting **reverse = True** sorts the list in the **descending order**.

| list.sort(reverse=True) |
| --- |

Alternatively for **sorted()**, you can use the following code.

| sorted(list, reverse=True) |
| --- |

Example 2: *Sort the list in Descending order*

| *# vowels list* vowels = ['e', 'a', 'u', 'o', 'i']  *# sort the vowels* vowels.sort(reverse=True)  *# print vowels* print('Sorted list (in Descending):', vowels) |
| --- |

Output

| >>> Sorted list (in Descending): ['u', 'o', 'i', 'e', 'a'] |
| --- |

Sort with custom function using key

If you want your own implementation for sorting, the **sort()** method also accepts a **key function** as an **optional parameter**.

Based on the results of the **key function**, you can sort the given list.

| list.sort(key=len) |
| --- |

Alternatively for sorted:

| sorted(list, key=len) |
| --- |

Here, ***len*** is Python's in-built function to count the **length of an element**.

The list is sorted based on the **length of each element**, from **lowest count to highest**.

We know that a ***tuple*** is sorted using its **first parameter by default**. Let's look at how to customize the **sort()** method to sort using the **second element**.

Example 3: *Sort the list using key*

| *# take second element for sort* def takeSecond(elem):  return elem[1]  *# random list* random = [(2, 2), (3, 4), (4, 1), (1, 3)]  *# sort list with key* random.sort(key=takeSecond)  *# print list* print('Sorted list:', random) |
| --- |

Output

| >>> Sorted list: [(4, 1), (2, 2), (1, 3), (3, 4)] |
| --- |

Let's take another example. Suppose we have a **list** of information about the employees of an office where each **element is a dictionary**.

We can sort the list in the following way:

| *# sorting using custom key* employees = [  {'Name': 'Alan Turing', 'age': 25, 'salary': 10000},  {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000},  {'Name': 'John Hopkins', 'age': 18, 'salary': 1000},  {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, ]  *# custom functions to get employee info* def get\_name(employee):  return employee.get('Name')  def get\_age(employee):  return employee.get('age')  def get\_salary(employee):  return employee.get('salary')  *# sort by name (Ascending order)* employees.sort(key=get\_name) print(employees, end='\n\n')  *# sort by Age (Ascending order)* employees.sort(key=get\_age) print(employees, end='\n\n')  *# sort by salary (Descending order)* employees.sort(key=get\_salary, reverse=True) print(employees, end='\n\n') |
| --- |

Output

| >>> [{'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'John Hopkins', 'age': 18, 'salary': 1000}, {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}]  >>> [{'Name': 'John Hopkins', 'age': 18, 'salary': 1000}, {'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}, {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}]  >>> [{'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, {'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}, {'Name': 'John Hopkins', 'age': 18, 'salary': 1000}] |
| --- |

Here, for the **first** case, our custom function **returns the name of each employee**. Since the name is a ***string***, Python by default sorts it using the **alphabetical order**.

For the **second** case, age (***int***) is returned and is sorted in **ascending order**.

For the **third** case, the function returns the salary (***int***), and is sorted in the **descending order using reverse = True**.

It is a good practice to use the ***lambda* function** when the function can be **summarized in one line**. So, we can also write the above program as:

| *# sorting using custom key* employees = [  {'Name': 'Alan Turing', 'age': 25, 'salary': 10000},  {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000},  {'Name': 'John Hopkins', 'age': 18, 'salary': 1000},  {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, ]  *# sort by name (Ascending order)* employees.sort(key=lambda x: x.get('Name')) print(employees, end='\n\n')  *# sort by Age (Ascending order)* employees.sort(key=lambda x: x.get('age')) print(employees, end='\n\n')  *# sort by salary (Descending order)* employees.sort(key=lambda x: x.get('salary'), reverse=True) print(employees, end='\n\n') |
| --- |

Output

| >>> [{'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'John Hopkins', 'age': 18, 'salary': 1000}, {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}]  >>> [{'Name': 'John Hopkins', 'age': 18, 'salary': 1000}, {'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}, {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}]  >>> [{'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, {'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}, {'Name': 'John Hopkins', 'age': 18, 'salary': 1000}] |
| --- |

### **\*list.copy()**

*The* ***copy()*** *method returns a shallow copy of the list.*

Example

| *# mixed list* prime\_numbers = [2, 3, 5]  *# copying a list* numbers = prime\_numbers.copy()   print('Copied List:', numbers) |
| --- |

Output

| >>> Copied List: [2, 3, 5] |
| --- |

SYNTAX

| new\_list = list.copy() |
| --- |

EXAMPLE

Example 1: *Copying a List*

| *# mixed list* my\_list = ['cat', 0, 6.7]  *# copying a list* new\_list = my\_list.copy()   print('Copied List:', new\_list) |
| --- |

Output

| >>> Copied List: ['cat', 0, 6.7] |
| --- |

If you modify the **new\_list** in the above example, **my\_list** will **not be modified**.

List copy using =

We can also use the **=** operator to copy a list. For example,

| old\_list = [1, 2, 3] new\_list = old\_list |
| --- |

However, there is one problem with copying lists in this way. If you modify **new\_list**, **old\_list** is also modified. It is because the new list is referencing or pointing to the same **old\_list** object.

| old\_list = [1, 2, 3]  *# copy list using =* new\_list = old\_list   *# add an element to list* new\_list.append('a')  print('New List:', new\_list) print('Old List:', old\_list) |
| --- |

Output

| >>> Old List: [1, 2, 3, 'a'] >>> New List: [1, 2, 3, 'a'] |
| --- |

However, if you need the original list **unchanged** when the new list is **modified**, you can use the **copy()** method.

Example 2: Copy List Using Slicing Syntax

| *# shallow copy using the slicing syntax* *# mixed list* list = ['cat', 0, 6.7] *# copying a list using slicing* new\_list = list[:] *# Adding an element to the new list* new\_list.append('dog') *# Printing new and old list* print('Old List:', list) print('New List:', new\_list) |
| --- |

Output

| >>> Old List: ['cat', 0, 6.7] >>> New List: ['cat', 0, 6.7, 'dog'] |
| --- |

### **\*list.clear()**

*The* ***clear()*** *method removes all items from the list.*

Example

| prime\_numbers = [2, 3, 5, 7, 9, 11]  *# remove all elements* prime\_numbers.clear()  *# Updated prime\_numbers List* print('List after clear():', prime\_numbers) |
| --- |

Output

| >>> List after clear(): [] |
| --- |

SYNTAX

| list.clear() |
| --- |

EXAMPLES

Example 1: Working of clear() method

| *# Defining a list* list = [{1, 2}, ('a'), ['1.1', '2.2']] *# clearing the list* list.clear() print('List:', list) |
| --- |

Output

| >>> List: [] |
| --- |

Example 2: Emptying the List Using del

| *# Defining a list* list = [{1, 2}, ('a'), ['1.1', '2.2']]  *# clearing the list* del list[:]  print('List:', list) |
| --- |

Output

| >>> List: [] |
| --- |

## 1.6 Tuples

#TODO

## 1.7 Dictionary

#TODO

## 1.8 Classes

Python Classes

A class is considered as a blueprint of objects. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

### SYNTAX

#TODO

# 2.0 Operators

Operators are used to perform operations on variables and values.

## 2.1 Math operators

Used to perform math operations

| Operator | Name/Description | Example |
| --- | --- | --- |
| + | addition | a+b |
| - | subtraction | a-b |
| \* | multiplication | a\*b |
| / | division | a/b |
| % | modulo. Returns the remainder of the division | a%b  5%2 = 1 |
| \*\* | exponent/power | a\*\*b  5\*\*2 = 25 |
| // | Floor division. Same as division, but rounds down to nearest whole number | a//b  5//2 = 2 |

Tips:  
  
 - To square a number, \*\*0.5 is preferred over math.sqrt()

It more convenient as there is no need to import math, and saves on runtime

* To check if a number is even, the modulo operator can be used to check if the number is divisible by 2.

For example:

| num = 5 if num%2 == 0: // check if number is divisible by 2  print("Number is a even number") else:  print("Number is a odd number") |
| --- |

## 2.2 Assignment operators

Used to perform math operations to variables and assign values

| Operator | Example | Same as |
| --- | --- | --- |
| = | x = 5 | x = 5 |
| += | x += 3 | x = x+3 |
| -= | x-= 4 | x = x-4 |
| \*= | x\*=2 | x = x\*2 |
| /= | x/= 3 | x = x/3 |
| %= | x %= 3 | x = x%3 |
| \*\*= | x\*\*= 5 | x = x\*\*5 |
| //= | x//= 8 | x//= 8 |

## 2.3 Comparison operators

Used to compare two values

| Operator | Name | Example |
| --- | --- | --- |
| == | is equal to | x == y |
| != | not equal to | x != y |
| > | greater than | x > y |
| < | less than | x < y |
| >= | greater than or equal to | x >= y |
| <= | less than or equal to | x <= y |

## 2.4 Logical operators

Logical operators are used to combine conditional statements

| Name | Description | Example |
| --- | --- | --- |
| and | returns True if both statements are true | x == 5 and y == 10 |
| or | returns True if either statement is true | x == 1 or y == 3 |
| not | returns False if result is True, returns True if result is False; Reverses the result | not x == 5 |

## 2.5 "in" operator

Check if a sequence is in an object

For example,

| "a" in "aeiou" //True 5 in [1,2,3,4] //False [1,3] in [[1,3], [2,4], [5,9]] //True |
| --- |

## 2.6 operator order

Parentheses (brackets) can be used to change the order in which operators are performed. Operators in brackets are evaluated first

Examples:

| print((5+3)/4) // 2  print(True or False and False) // True print((True or False) and False) //False  print(not True and False) //False  print(not(True and False)) // True |
| --- |

For math operators, multiplication and divisions are evaluated first, followed by addition and subtraction. For example, 5+3/2 is the same as 5+(3/2)

# 

# 3.0 If statements

An "if statement" is written by using the if keyword.

Example:

| a = 5 b = 3  if a > b:  print("a is greater than b") |
| --- |

In this example, the condition for the if statement is a>b. Since this condition is met, the program will print out the line "a is greater than b"

## 3.1 Indentation

Python uses indentation (white space at the start of the line) to set the scope of the code.

Example 1:

| a = 2 b = 1  if a == 1 and b == 1:  print("both a and b are 1")  print("yay!") |
| --- |

Output:

|  |
| --- |

Example 2:

| a = 2 b = 1  if a == 1 and b == 1:  print("both a and b are 1") print("yay!") |
| --- |

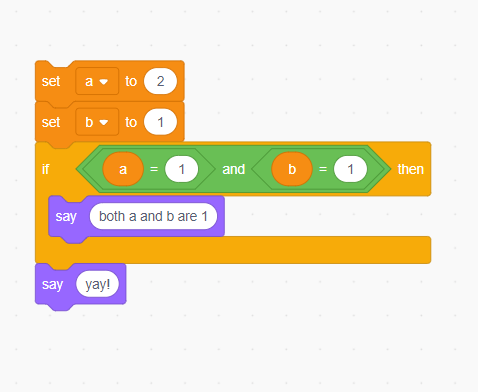
Output:

| yay! |
| --- |

In example 1, both print statements are indented. This means that the if statement "contains" both prints, hence the two messages are only outputted when the condition is met

In example 2, the first print is indented but the second isn't. This means that the if statement only "contains" the first print, and the second print is "outside" the if statement, hence the second print would still print even when the condition of the if statement is not met

Another way to understand indentation is using code blocks to visualise it. Here's example 2 in scratch:



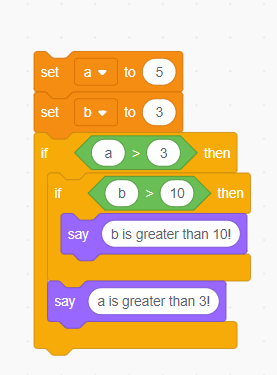
With nested if statements (if statement inside an if statement), indentation becomes more complex. Heres an example in python and scratch:  
  
Python:

| a = 5 b = 3  if a > 3:  if b > 10:  print("b is greater than 10!")  print("a is greater than 3!") |
| --- |

output:

| a is greater than 3! |
| --- |

Scratch:



## 3.2 Else

Else is used to catch anything that does not meet the other conditions.

For example

| a = 5  if a > 5:  print("a is greater than 5") else:  print("a is less than or equal to 5") |
| --- |

output

| a is less than or equal to 5 |
| --- |

## 3.3 Elif

elif is the shorthand of "else if". If the previous conditions were not met, then the elif statement would be evaluated.

Example:

| x = 3 y = 3 if x < y:  print("x is less than y") elif x == y:  print("x is equal to y") |
| --- |

output

| x is equal to y |
| --- |

You can also chain multiple elif statements together and include an else statement, as such:

| grade = 65  if grade => 80:  print("Excellent!") elif grade >= 70:  print("Well done!") elif grade >= 60:  print("Nice try!") else:  print("bad") |
| --- |

output

| Nice try! |
| --- |

# 4.0 (User-defined) Functions

**It is recommended to learn indentation before learning this section. It is found at "3.1** [**Indentation**](#_g3bcssdjzl82)**"**

A function is a block of code that only runs when it is called. You can also pass data into a function

**Defining functions**

A function can be defined with the "def" keyword

| def function\_name():  print("function!") |
| --- |

**Calling functions**

Calling a function runs the code in the function. You can call a function via its name followed by parenthesis

Example:

| def function\_name():  print("function!")  function\_name() |
| --- |

Output:

| function! |
| --- |

## 4.1 Arguments and parameters

Data can be passed into a function as arguments.

Arguments can be added to a function inside the parentheses when defining it. They are called parameters. A parameter acts as a local variable and it cannot be used outside the function.

A value can be passed as an argument by including the value inside the parenthesis in the function call.

In the example below, the function welcome has a parameter (name), and it is called once, with the argument being "Toby"

Example:

| def welcome(name):  print("Welcome, {}!".format(name))  welcome("Toby") |
| --- |

Output:

| Welcome, Toby! |
| --- |

You can have multiple arguments for one function, by using a comma to separate each one

Example:

| def shout(name, message):  print("Hey, {}! {}".format(name,message))  shout("Lee", "You smell like a dirty sock") |
| --- |

Output:

| Hey, Lee! You smell like a dirty sock |
| --- |

## 4.2 Return statements

Functions can also have a return statement, which lets the function return a value.

| def power(a,b):  return a\*\*b  print(power(5,2)) // printing the function return result = power(2,2) // assigning the function return to a variable |
| --- |

When the return statement is evaluated, it will also end the function. This can be used to stop the rest of the code in the function from running, as such:

| def isEven(n):  if n%2 == 0:  return  print("Is even!")   isEven(2) //outputs "Is even!"  isEven(1) //no output |
| --- |

## \*4.3 Default parameter value

A function parameter can have a default value, by using the "=" operator to assign it a value in the function definition.

When no value is passed to the parameter during a function call, the parameter will have the default value instead

Example 1:

| def getScore(score=100):  print("My score is {}".format(score))  getScore(50) getScore() |
| --- |

output:

| My score is 50 My score is 100 |
| --- |

Example 2:

| def intro(name, country = "Sweden"):  print("Hi, I'm {} and I am from {}".format(name, country))  intro("Jack") intro("Jayden","Singapore") |
| --- |

output:

| Hi, I'm Jack and I am from Sweden Hi, I'm Jayden and I am from Singapore |
| --- |

# 5.0 Loops

In computer programming, **loops are used to repeat a block of code**.

For example, if we want to show a message **100 times**, then we can use a **loop**. It's just a simple example; you can achieve much more with loops.

There are 2 types of loops in Python:

* ***for*** loop
* ***while*** loop

## 5.1 For Loops

*In Python, a* ***for*** *loop is used to iterate over sequences such as* ***lists****,* ***tuples****,* ***string****, etc.*

For example,

| languages = ['Swift', 'Python', 'Go', 'JavaScript']  *# run a loop for each item of the list* for language in languages:  print(language) |
| --- |

Output

| >>> Swift >>> Python >>> Go >>> JavaScript |
| --- |

In the above example, we have created a ***list*** called **languages**.

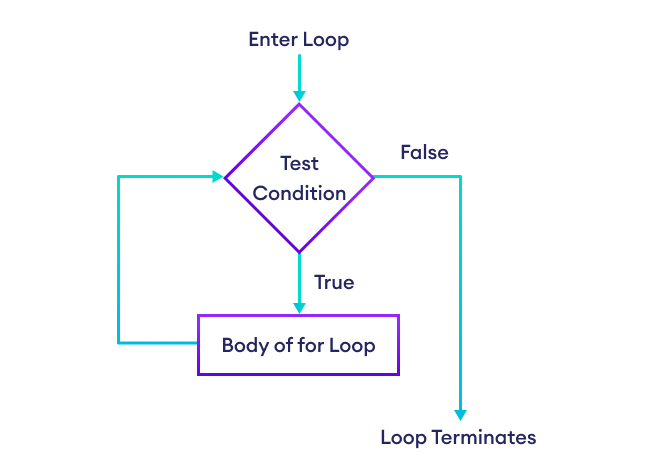
Initially, the value of **language** is set to the **first element** of the array,i.e. **Swift**, so the ***print*** statement **inside the loop** is executed.

**language** is updated with the next element of the list, and the **print statement is executed again**. This way, the **loop runs until the last element of the list is accessed**.

SYNTAX

| for val in sequence:  *# statement(s)* |
| --- |

Here, ***val*** accesses each item of sequence on each iteration. The loop continues **until we reach the last item in the sequence**.

Flowchart of Python for Loop

*Working of Python for loop*

EXAMPLES

Example: *Loop Through a String*

| for x in 'Python':  print(x) |
| --- |

Output

| >>> P >>> y >>> t >>> h >>> o >>> n |
| --- |

Python for Loop with Python **range()**

A range is a series of values **between two numeric intervals**.

We use Python's built-in function **range()** to define a range of values. For example,

| values = range(4) |
| --- |

Here,**4** inside **range()** defines a range containing values **0**, **1**, **2**, **3**.

In Python, we can use a ***for*** loop to iterate **over a range**. For example,

| *# use of range() to define a range of values* values = range(4)  *# iterate from i = 0 to i = 3* for i in values:  print(i) |
| --- |

Output

| >>> 0 >>> 1 >>> 2 >>> 3 |
| --- |

In the above example, we have used the ***for*** loop to iterate over a range from **0** to **3**.

The value of ***i*** is set to **0** and it is updated to the next number of the range on each iteration. This process continues until **3** is reached.

| Iteration | Condition | Action |
| --- | --- | --- |
| 1st | True | 0 is printed. i is increased to 1. |
| 2nd | True | 1 is printed. i is increased to 2. |
| 3rd | True | 2 is printed. i is increased to 3. |
| 4th | True | 3 is printed. i is increased to 4. |
| 5th | False | The loop is terminated |

Note: To learn more about the use of ***for* loop with range**, visit Python ***range()***.

Using a for Loop Without Accessing Items

It is **not mandatory** to use items of a sequence within a ***for*** loop.

For example,

| languages = ['Swift', 'Python', 'Go']  for language in languages:  print('Hello')  print('Hi') |
| --- |

Output

| >>> Hello >>> Hi >>> Hello >>> Hi >>> Hello >>> Hi |
| --- |

Here, the loop runs **three** times because our list has **three** items. In each iteration, the loop body prints **'Hello'** and **'Hi'**. The items of the list are **not** used within the loop.

If we do not intend to use items of a sequence **within the loop**, we can write the loop like this:

| languages = ['Swift', 'Python', 'Go']  for \_ in languages:  print('Hello')  print('Hi') |
| --- |

The **\_** symbol is used to denote that the elements of a sequence will ***not* be used within the loop body**.

Python for loop with else

A for loop can have an optional else block. The else part is executed when the loop is exhausted (after the loop iterates through every item of a sequence). For example,

| digits = [0, 1, 5]  for i in digits:  print(i) else:  print("No items left.") |
| --- |

Output

| >>> 0 >>> 1 >>> 5 >>> No items left. |
| --- |

Here, the for loop prints **all the items** of the ***digits* list**. When the loop finishes, it executes the **else block** and prints ***No items left***.

Note: The else block will **not execute** if the for loop is stopped by a ***break*** statement.

## \*5.2 While Loops

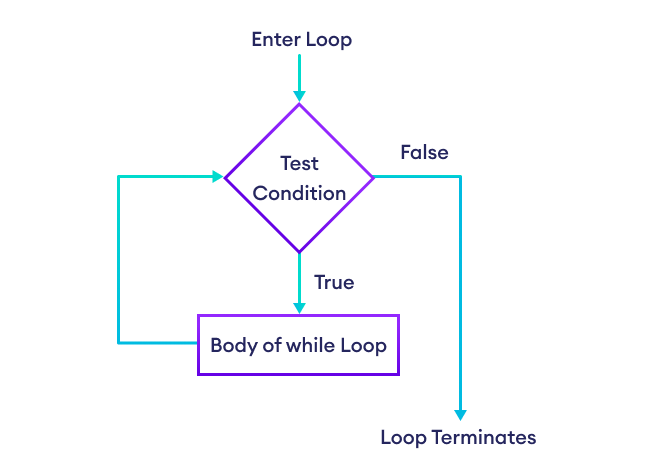
*Python while loop is used to run a block code until a certain condition is met.*

SYNTAX

| while condition:  *# body of while loop* |
| --- |

Here,

1. A ***while*** loop evaluates the ***condition***
2. If the ***condition*** evaluates to **True**, the code inside the ***while*** loop is executed.
3. ***condition*** is evaluated again.
4. This process continues until the condition is **False**.
5. When ***condition*** evaluates to **False**, the loop stops.



Flowchart of Python while Loop

*Flowchart of while Loop*

EXAMPLES

Example 1: *Python* ***while*** *Loop*

| *# program to display numbers from 1 to 5*  *# initialize the variable* i = 1 n = 5  *# while loop from i = 1 to 5* while i <= n:  print(i)  i = i + 1 |
| --- |

Output

| >>> 1 >>> 2 >>> 3 >>> 4 >>> 5 |
| --- |

Here's how the program works:

| Variable |  | Condition: ***i*** <= ***n*** | Action |
| --- | --- | --- | --- |
| i = 1 | n = 5 | True | 1 is printed. i is increased to 2. |
| i = 2 | n = 5 | True | 2 is printed. i is increased to 3. |
| i = 3 | n = 5 | True | 3 is printed. i is increased to 4. |
| i = 4 | n = 5 | True | 4 is printed. i is increased to 5. |
| i = 5 | n = 5 | True | 5 is printed. i is increased to 6. |
| i = 6 | n = 5 | False | The loop is terminated. |

Example 2: *Python* ***while*** *Loop*

| *# program to calculate the sum of numbers* *# until the user enters zero*  total = 0  number = int(input('Enter a number: '))  *# add numbers until number is zero* while number != 0:  total += number *# total = total + number*    *# take integer input again*  number = int(input('Enter a number: '))    print('total =', total) |
| --- |

Output

| >>> Enter a number: *12* >>> Enter a number: *4* >>> Enter a number: *-5* >>> Enter a number: *0* >>> total = *11* |
| --- |

In the above example, the while iterates until the user enters ***zero***. When the user enters ***zero***, the test condition evaluates to **False** and the loop **ends**.

Infinite while Loop in Python

If the condition of a loop is always **True**, the loop runs for **infinite times** (until the memory is full).

For example,

| age = 32  *# the test condition is always True* while age > 18:  print('You can vote') |
| --- |

In the above example, the condition **always** evaluates to **True**. Hence, the loop body will run for **infinite** times.

Python While loop with else

In Python, a ***while*** loop may have an **optional else block**.

Here, the else part is executed ***after*** the condition of the loop evaluates to **False**.

| counter = 0  while counter < 3:  print('Inside loop')  counter = counter + 1 else:  print('Inside else') |
| --- |

Output

| >>> Inside loop >>> Inside loop >>> Inside loop >>> Inside else |
| --- |

Note: The else block will **not** execute if the while loop is terminated by a ***break*** statement.

E.g.

| counter = 0  while counter < 3:  *# loop ends because of break*  *# the else part is not executed*   if counter == 1:  break   print('Inside loop')  counter = counter + 1 else:  print('Inside else') |
| --- |

Output

| >>> Inside loop  >>> Inside else |
| --- |

Python for Vs while loops

The for loop is usually used when the number of iterations is known. For example,

| *# this loop is iterated 4 times (0 to 3)* for i in range(4):  print(i) |
| --- |

The while loop is usually used when the number of iterations is unknown. For example,

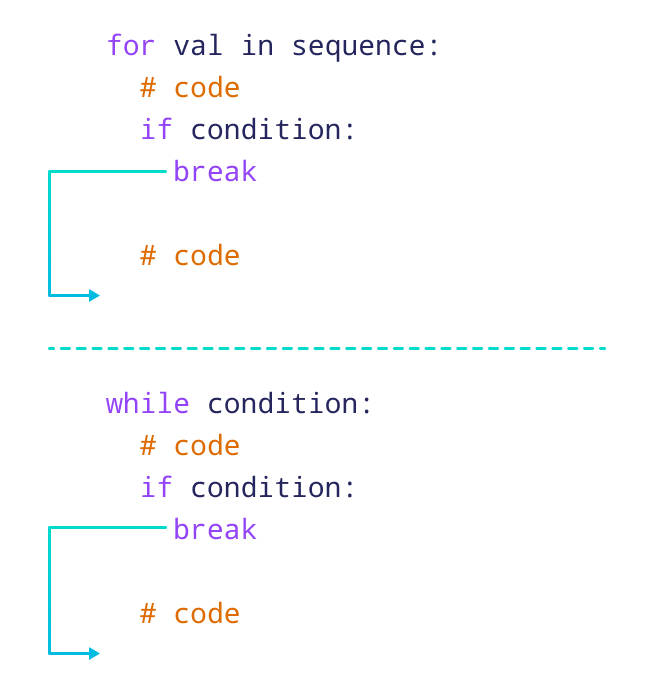
| while condition:  *# run code until the condition evaluates to False* |
| --- |

## \*5.3 Break and Continue

*The* ***break*** *statement is used to* ***terminate the loop immediately*** *when it is encountered.*

SYNTAX

| break |
| --- |

Working of Python break Statement

Python break Statement with for Loop

We can use the break statement with the for loop to terminate the loop when a certain condition is met.

For example,

| for i in range(5):  if i == 3:  break  print(i) |
| --- |

Output

| >>> 0 >>> 1 >>> 2 |
| --- |

In the above example, we have used the ***for*** loop to print the value of ***i***. Notice the use of the ***break*** statement,

| if i == 3:  break |
| --- |

Here, when ***i*** is equal to ***3***, the break statement terminates the loop. Hence, the output doesn't include values after ***2***.

Note: The ***break*** statement is almost always used with decision-making statements.

Python break Statement with while Loop

We can also terminate the while loop using the break statement.

For example,

| *# program to find first 5 multiples of 6*  i = 1  while i <= 10:  print('6 \* ',(i), '=',6 \* i)   if i >= 5:  break    i = i + 1 |
| --- |

Output

| >>> 6 \* 1 = 6 >>> 6 \* 2 = 12 >>> 6 \* 3 = 18 >>> 6 \* 4 = 24 >>> 6 \* 5 = 30 |
| --- |

In the above example, we have used the ***while*** loop to find the first **5** multiples of **6**. Here notice the line,

| if i >= 5:  break |
| --- |

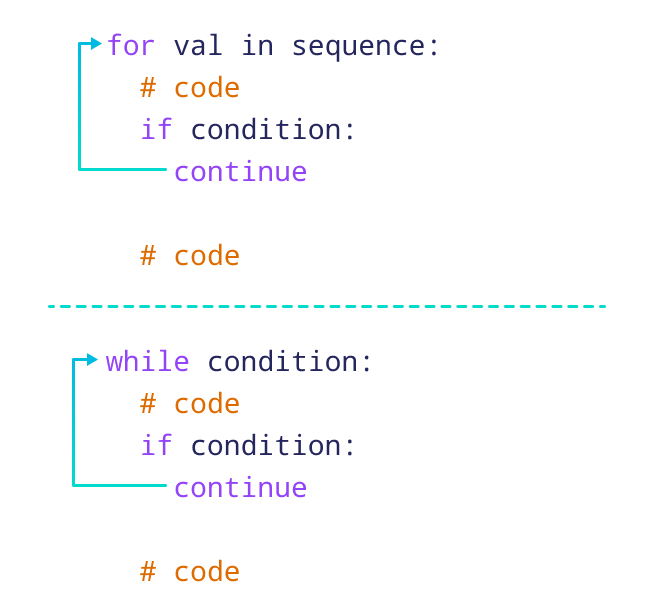
This means when ***i*** is **greater than or equal** to **5**, the ***while*** loop is terminated.

Python continue Statement

The ***continue*** statement is used to **skip** the current iteration of the loop and the control flow of the program goes to the **next iteration**.

SYNTAX

| continue |
| --- |

Working of Python continue Statement

Python continue Statement with for Loop

We can use the ***continue*** statement with the **for** loop to skip the current iteration of the loop. Then the control of the program **jumps to the next iteration**. For example,

| for i in range(5):  if i == 3:  continue  print(i) |
| --- |

Output

| >>> 0 >>> 1 >>> 2 >>> 4 |
| --- |

In the above example, we have used the ***for*** loop to print the value of ***i***. Notice the use of the ***continue*** statement,

| if i == 3:  continue |
| --- |

Here, when ***i*** is equal to **3**, the ***continue*** statement is executed. Hence, the value **3** is not printed to the output.

Python continue Statement with while Loop

In Python, we can also skip the current iteration of the ***while*** loop using the ***continue*** statement. For example,

| *# program to print odd numbers from 1 to 10*  num = 0  while num < 10:  num += 1    if (num % 2) == 0:  continue   print(num) |
| --- |

Output

| >>> 1 >>> 3 >>> 5 >>> 7 >>> 9 |
| --- |

In the above example, we have used the ***while*** loop to print the **odd** numbers between **1** to **10**. Notice the line,

| if (num % 2) == 0:  continue |
| --- |

Here, when the number is **even**, the ***continue*** statement skips the current iteration and starts the next iteration.

## \*5.4 Pass

In Python programming, the ***pass*** statement is a **null** statement which can be used as a **placeholder for future code**.

Suppose we have a loop or a function that is **not implemented yet**, but we want to **implement it in the future**. In such cases, we can use the ***pass*** statement.

SYNTAX

| pass |
| --- |

Using pass With Conditional Statement

| n = 10  *# use pass inside if statement* if n > 10:  pass  print('Hello') |
| --- |

Here, notice that we have used the ***pass*** statement inside the ***if*** statement.

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

Suppose we didn't use pass or just put a comment as:

| n = 10  if n > 10:  *# write code later*  print('Hello') |
| --- |

Here, we will get an error message: ***IndentationError: expected an indented block***

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

Use of pass Statement inside Function or Class

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

For example,

| def function(args):  pass  class Example:  pass |
| --- |

# 6.0 Built-in Functions

*Python has several functions that are readily available for use. These functions are called* ***built-in*** *functions.*

## 6.1 File I/O

A file is a container in computer storage devices used for storing 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 the **resources that are tied with the file are freed**.

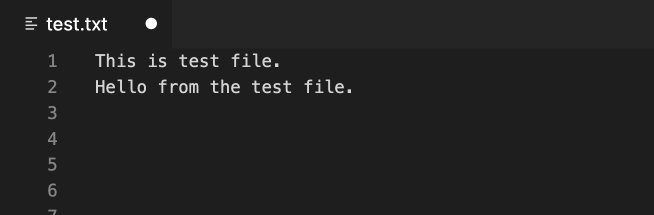
Hence, in Python, a file operation takes place in the following order:

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

Opening Files in Python

In Python, we use the ***open()*** method to open files.

To demonstrate how we open files in Python, let's suppose we have a file named test.txt with the following content.



*Opening Files in Python*

Now, let's try to open data from this file using the ***open()*** function.

| *# open file in current directory* file1 = open("test.txt") |
| --- |

Here, we have created a file object named **file1**. This object can be used to work with files and directories.

By default, the files are open in **read mode** (cannot be modified). The code above is equivalent to:

| file1 = open("test.txt", "r") |
| --- |

Here, we have explicitly specified the mode by passing the **"r"** argument which means file is opened for **reading**.

Different Modes to Open a File in Python

| **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 a file 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**) |

Here's few simple examples of how to open a file in different modes,

| file1 = open("test.txt") *# equivalent to 'r' or 'rt'* file1 = open("test.txt",'w') *# write in text mode* file1 = open("img.bmp",'r+b') *# read and write in binary mode* |
| --- |

Reading Files in Python

After we open a file, we use the ***read()*** method to **read** its contents. For example,

| *# open a file* file1 = open("test.txt", "r")  *# read the file* read\_content = file1.read() print(read\_content) |
| --- |

Output

| >>> This is a test file. >>> Hello from the test file. |
| --- |

In the above example, we have read the **test.txt** file that is available in our current directory. Notice the code,

| read\_content = file1.read() |
| --- |

Here, **file1.*read()*** reads the **test.txt** file and is stored in the ***read\_content*** variable.

Closing Files in Python

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

Closing a file will free up the resources that were tied with the file. It is done using the ***close()*** method in Python. For example,

| *# open a file* file1 = open("test.txt", "r")  *# read the file* read\_content = file1.read() print(read\_content)  *# close the file* file1.close() |
| --- |

Output

| >>> This is a test file. >>> Hello from the test file. |
| --- |

Here, we have used the ***close()*** method to close the file.

After we perform file operation, we should always close the file; it's a good programming practice.

Exception Handling in Files

If an exception occurs when we are performing some operation with the file, the code exits **without closing the file**. A safer way is to use a ***try...finally*** block.

Let's see an example,

| try:  file1 = open("test.txt", "r")  read\_content = file1.read()  print(read\_content)  finally:  *# close the file*  file1.close() |
| --- |

Here, we have closed the file in the ***finally*** block as ***finally*** always executes, and the file will be closed **even if an exception occurs**.

Use of with...open Syntax

In Python, we can use the ***with...open*** syntax to **automatically close the file**. For example,

| with open("test.txt", "r") as file1:  read\_content = file1.read()  print(read\_content) |
| --- |

Note: Since we don't have to worry about closing the file, make a habit of using the **with...open** syntax.

Writing to Files in Python

There are **two** things we need to remember while **writing to a file**.

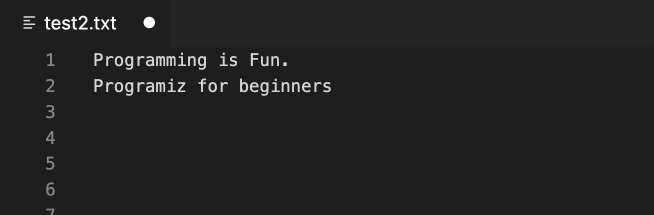
* If we try to **open a file that doesn't exist**, a **new file is created**.
* If a file **already exists**, its **content is erased**, and **new content is added to the file**.

In order to write into a file in Python, we need to **open it in write mode** by passing **"w"** inside ***open()*** as a **second argument**.

Suppose, we **don't have a file named *test2.txt***. Let's see what happens if we write contents to the ***test2.txt*** file.

| with open(test2.txt', 'w') as file2:   # write contents to the test2.txt file  file2.write('Programming is Fun.')  fil2.write('Programiz for beginners') |
| --- |

Here, a new ***test2.txt*** file is created and this file will have contents specified inside the ***write()*** method.



*Writing to Python Files*

Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

Here is the complete list of methods in text mode with a brief description:

| Method | Description |
| --- | --- |
| close() | Closes an opened file. It has no effect if the file is already closed. |
| detach() | Separates the underlying binary buffer from the TextIOBase and returns it. |
| fileno() | Returns an integer number (file descriptor) of the file. |
| flush() | Flushes the write buffer of the file stream. |
| isatty() | Returns True if the file stream is interactive. |
| read(***n***) | Reads at most ***n*** **characters** from 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**) | Reads and returns **one** line from the file. Reads in at most ***n*** **bytes** if specified. |
| readlines(***n*** = **-1**) | Reads and returns a **list of lines** from the file. Reads in at most ***n*** **bytes/characters** if specified. |
| seek(***offset***,*from*=**SEEK\_SET**) | Changes 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 an integer that represents the current position of the file's object. |
| truncate(***size*** = **None)** | Resizes the file stream to ***size*** bytes. If ***size*** is not specified, resizes to **current location**. |
| writable() | Returns **True** if the file stream can be written to. |
| write(s) | Writes the **string *s*** to the file and **returns the number of characters written**. |
| writelines(***lines***) | Writes a **list** of ***lines*** to the file. |

## 6.2 Character Recognition with *ord()* and *chr()*

*The* ***ord()*** *function returns an* ***integer*** *representing the* ***Unicode character****.*

*The* ***chr()*** *method converts an* ***integer*** *to its* ***unicode character*** *and returns it.*

In other words, ***ord()*** takes a **character** (E.g. **“A”**) and converts it into **numbers** (For **“A”**, it’s **“65”**)

And ***chr()*** takes the **number** (**“65”**) and turns it back into the **character** (**“A”**).

| character = 'A' *# find unicode of A* unicode\_char = **ord(**character**)** print(unicode\_char) |
| --- |

Output

| >>> 65 |
| --- |

| *#find character of unicode 97* print(chr(97)) *#find character of unicode 98* print(chr(98)) |
| --- |

Output

| >>> a >>> b |
| --- |

chr() SYNTAX, PARAMETERS and RETURN VALUE

| chr(number) *#number - an integer number in the range 0 to 1,114,111* |
| --- |

Returns:

* a **unicode character** of the **corresponding integer argument** (in the range **0** to **1,114,111**)
* **ValueError** - for an **out of range** **integer** number
* **TypeError** - for a **non-integer** argument

ord() SYNTAX, PARAMETERS and RETURN VALUE

| ord(ch) *#ch - a Unicode character* |
| --- |

The ***ord()*** function returns an **integer** representing the **Unicode character**.

Note: A list of common unicode characters is provided in the attached Excel file ***unicode.xlsx***

## 6.3 Math Functions

### **abs()**

*The* ***abs()*** *function returns the absolute value of the given number. If the number is a complex number, abs() returns its magnitude.*

Example

| number = -20  absolute\_number = abs(number) print(absolute\_number)  *# Output: 20* |
| --- |

SYNTAX and PARAMETERS

| abs(num) |
| --- |

* **num** - a number whose absolute value is to be returned. The number can be:
  + integer
  + floating number
  + complex number

RETURN VALUE

* For integers - integer absolute value is returned
* For floating numbers - floating absolute value is returned
* For complex numbers - magnitude of the number is returned

### **complex()**

*The* ***complex()*** *method returns a complex number when real and imaginary parts are provided, or it converts a string to a complex number.*

SYNTAX and PARAMETERS

| complex([real[, imag]]) |
| --- |

* **real** - real part. If real is omitted, it defaults to 0.
* **imag** - imaginary part. If imag is omitted, it defaults to 0.

If the first parameter passed to this method is a **string**, it will be interpreted as a **complex number**. In this case, the **second parameter** **shouldn't be passed**.

RETURN VALUE

As suggested by the name, ***complex()*** method returns a complex number.

If the string passed to this method is not a valid complex number, ***ValueError* exception** is raised.

Note: The string passed to ***complex()*** should be in the form real+imagJ or real+imagJ

### **divmod()**

*The* ***divmod()*** *method takes two numbers as arguments and returns their quotient and remainder in a tuple.*

Example

| *# returns the quotient and remainder of 8/3* result = divmod(8, 3)  print('Quotient and Remainder = ',result)  *# Output: Quotient and Remainder = (2, 2)* |
| --- |

SYNTAX and PARAMETERS

| divmod(number1, number2) |
| --- |

* **number1** - ***numerator***, can be an integer or a floating point number
* **number2** - ***denominator***, can be an integer or a floating point number

RETURN VALUE

* ***(quotient, remainder)*** - a **tuple** that contains ***quotient*** and ***remainder*** of the division
* **TypeError** - for any non-numeric argument

### **min()**

*The* ***min()*** *function returns the* ***smallest*** *item in an iterable. It can also be used to find the smallest item between* ***two or more parameters****.*

Example

| numbers = [9, 34, 11, -4, 27]  *# find the smallest number* min\_number = min(numbers) print(min\_number)  *# Output: -4* |
| --- |

The min() function has two forms:

| *# to find the smallest item in an iterable* min(iterable, \*iterables, key, default)  *# to find the smallest item between two or more objects* min(arg1, arg2, \*args, key) |
| --- |

1. ***min()*** with iterable arguments

SYNTAX and PARAMETERS

| min(iterable, \*iterables, key, default) |
| --- |

* **iterable** - an iterable such as list, tuple, set, dictionary, etc.
* **\*iterables** (optional) - any number of iterables; can be more than one
* **key** (optional) - key function where the iterables are passed and comparison is performed based on its return value
* **default** (optional) - default value if the given iterable is empty

RETURN VALUE

***min()*** returns the smallest element from an iterable.

Example: *The smallest string in a list*

| languages = ["Python", "C Programming", "Java", "JavaScript"] smallest\_string = min(languages);  print("The smallest string is:", smallest\_string) |
| --- |

Output

| >>> The smallest string is: C Programming |
| --- |

### **max()**

*Inverse of* ***min()****. Refer to* [***min()***](#_6naeuq2vxf1)

RETURN VALUE

***max()*** returns the **largest** element from an iterable instead.

### **round()**

*The* ***round()*** *function returns a floating-point number rounded to the specified number of decimals.*

Example

| number = 13.46  *# round 13.46 to the nearest integer* rounded\_number = round(number) print(rounded\_number)  *# Output: 13* |
| --- |

SYNTAX and PARAMETERS

| round(number, ndigits) |
| --- |

* **number** - the number to be rounded
* **ndigits** (optional) - number up to which the given number is rounded; defaults to **0**

RETURN VALUE

The ***round()*** function returns the

* **nearest integer** to the given number if ndigits is **not provided**
* number rounded off to the **ndigits digits** if ndigits **is provided**

Example

| *# for integers* print(round(10))  *# for floating point* print(round(10.7))  *# even choice* print(round(5.5)) |
| --- |

Output

| >>> 10 >>> 11 >>> 6 |
| --- |

Note: The behaviour of ***round()*** for **floats** can be surprising. Notice ***round(*2.675, 2)** gives **2.67** instead of the expected **2.68**. **This is not a bug**: it's a result of the fact that most decimal fractions **can't be represented exactly as a float**.

When the **decimal 2.675** is converted to a **binary floating-point number**, it's again replaced with a **binary approximation**, whose exact value is:

| 2.67499999999999982236431605997495353221893310546875 |
| --- |

Due to this, it is **rounded down** to 2.67.

**If you're in a situation where this precision is needed,** consider using the ***decimal*** module, which is **designed for floating-point arithmetic**:

| from decimal import Decimal  *# normal float* num = 2.675 print(round(num, 2))  *# using decimal.Decimal (passed float as string for precision)* num = Decimal('2.675') print(round(num, 2)) |
| --- |

Output

| >>> 2.67 >>> 2.68 |
| --- |

## 6.4 Type Functions

### **bool()**

*The* ***bool()*** *method takes a specified argument and returns its boolean value.*

Example

| test = 1  *# returns boolean value of 1* print(test, 'is', bool(test))  *# Output: 1 is True* |
| --- |

SYNTAX and PARAMETERS

| bool(argument) |
| --- |

* **argument** - whose boolean value is returned

RETURN VALUES

* **False** - if argument is **empty**, **False**, **0** or **None**
* **True** - if argument is **any number** (besides 0), **True** or a **string**

Example:***bool()*** *with* ***True*** *arguments*

| test = 254 *# bool() with an integer number* print(test, 'is', bool(test))  test1 = 25.14 *# bool() with a floating point number* print(test1, 'is', bool(test1))  test2 = 'Python is the best' *# bool() with a string* print(test2, 'is', bool(test2))  test3 = True *# bool() with True* print(test3, 'is', bool(test3)) |
| --- |

Output

| >>> 254 is True >>> 25.14 is True >>> Python is the best is True >>> True is True |
| --- |

In the above example, we have used the ***bool()*** method with various arguments like integer, floating point numbers, and string.

Here, the method returns **True** values for arguments like ***25***, ***25.14***, ***'Python is a String'***, and **True**.

Example: ***bool()*** *with* ***False*** *arguments*

| test = [] *# bool() with an empty argument* print(test, 'is' ,bool(test))  test1 = 0 *# bool() with zero* print(test1, 'is' ,bool(test1))  test2 = None *# bool() with none* print(test2, 'is' ,bool(test2))  test3 = False *# bool() with False* print(test3, 'is' ,bool(test3)) |
| --- |

Output

| >>> [] is False >>> 0 is False >>> None is False >>> False is False |
| --- |

In the above example, the ***bool()*** method returns **False** values for arguments like **0**, **None**, **False** and **[]**.

### **list()**

*The list() constructor returns a list in Python.*

Example

| text = 'Python'  *# convert string to list* text\_list = list(text) print(text\_list)  *# check type of text\_list* print(type(text\_list))  *# Output: ['P', 'y', 't', 'h', 'o', 'n']* *# <class 'list'>* |
| --- |

SYNTAX and PARAMETERS

list([iterable])

* **iterable** (optional) - an object that could be a **sequence** (***string***, ***tuples***) or **collection** (***set***, ***dictionary***) or any ***iterator*** object

RETURN VALUES

* If no parameters are passed, it returns an **empty list**
* If an ***iterable*** is passed as a parameter, it creates a **list consisting of iterable's items**.

Example 1: *Create lists from* ***string****,* ***tuple****, and* ***list***

| *# empty list* print(list())  *# vowel string* vowel\_string = 'aeiou' print(list(vowel\_string))  *# vowel tuple* vowel\_tuple = ('a', 'e', 'i', 'o', 'u') print(list(vowel\_tuple))  *# vowel list* vowel\_list = ['a', 'e', 'i', 'o', 'u'] print(list(vowel\_list)) |
| --- |

Output

| >>> [] >>> ['a', 'e', 'i', 'o', 'u'] >>> ['a', 'e', 'i', 'o', 'u'] >>> ['a', 'e', 'i', 'o', 'u'] |
| --- |

Example 2: *Create lists from* ***set*** *and* ***dictionary***

| *# vowel set* vowel\_set = {'a', 'e', 'i', 'o', 'u'} print(list(vowel\_set))  *# vowel dictionary* vowel\_dictionary = {'a': 1, 'e': 2, 'i': 3, 'o':4, 'u':5} print(list(vowel\_dictionary)) |
| --- |

Output

| >>> ['a', 'o', 'u', 'e', 'i'] >>> ['o', 'e', 'a', 'u', 'i'] |
| --- |

Note: In the case of dictionaries, the keys of the dictionary will be the items of the list. Also, the order of the elements will be random.

### **object()**

*The* ***object()*** *function returns a* ***featureless object*** *which is a base for all classes. In other words, it’s useless... most of the time*

SYNTAX

| o = object() |
| --- |

### **any()**

*The* ***any()*** *function* ***returns True*** *if* ***any element*** *of an iterable is* ***True****. If not, it returns* ***False****.*

Example

| boolean\_list = ['True', 'False', 'True']  *# check if any element is true* result = any(boolean\_list) print(result)  *# Output: True* |
| --- |

SYNTAX and PARAMETERS

| any(iterable) |
| --- |

The ***any()*** function takes an iterable (list, string, dictionary etc.) in Python.

RETURN VALUES

* **True** if **at least one** element of an iterable is **true**
* **False** if **all elements** are **false** or if an iterable is **empty**

| Condition | Return Value |
| --- | --- |
| All values are true | True |
| All values are false | False |
| One value is true (others are false) | True |
| One value is false (others are true) | True |
| Empty Iterable | False |

### **all()**

*The* ***all()*** *function returns* ***True*** *if* ***ALL*** *elements in the given iterable are* ***true****. If not, it returns* ***False****. It is the* ***opposite*** *of* ***any()***

For syntax and parameters, refer to [*any()*](#_wgsjrehy829v)

RETURN VALUES

* **True** - If **all** elements in an iterable are **True**
* **False** - If **any** element in an iterable is **False**

| Condition | Return Value |
| --- | --- |
| All values are true | True |
| All values are false | False |
| One value is true (others are false) | False |
| One value is false (others are true) | False |
| Empty Iterable | True |

### **callable()**

*The* ***callable()*** *method returns* ***True*** *if the object passed* ***appears callable****. If not, it returns* ***False****. Used for debugging*

SYNTAX and PARAMETERS

| callable(object) |
| --- |

***callable()*** method takes a single argument ***object***.

RETURN VALUES

* **True** - if the object appears callable
* **False** - if the object is not callable.

It is important to remember that, **even if** ***callable()*** is **True**, *calling* the object may still fail.

However, if ***callable()*** returns **False**, calling the object will **certainly** fail.

### **filter()**

*The* ***filter()*** *function selects* ***elements from an iterable*** *(****list****,* ***tuple*** *etc.) based on the output of a function.*

The function is applied to each element of the iterable and if it returns **True**, the element is selected by the **filter()** function.

Example

| *# returns True if the argument passed is even*  def check\_even(number):  if number % 2 == 0:  return True    return False   numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  *# if an element passed to check\_even() returns True, select it* even\_numbers\_iterator = filter(check\_even, numbers)  *# converting to list* even\_numbers = list(even\_numbers\_iterator)  print(even\_numbers)  *# Output: [2, 4, 6, 8, 10]* |
| --- |

SYNTAX and PARAMETERS

| filter(function, iterable) |
| --- |

* **function** - a function
* **iterable** - an iterable like ***sets***, ***lists***, ***tuples*** etc.

RETURN VALUE

The ***filter()*** function returns an iterator.

Note: You can easily convert iterators to **sequences** like ***lists***, ***tuples***, ***strings*** using ***list()***, ***tuple()*** and ***str()*** etc.

Example 1: ***filter()***

| letters = ['a', 'b', 'd', 'e', 'i', 'j', 'o']  *# a function that returns True if letter is vowel* def filter\_vowels(letter):  vowels = ['a', 'e', 'i', 'o', 'u']  return True if letter in vowels else False  filtered\_vowels = filter(filter\_vowels, letters)  *# converting to tuple* vowels = tuple(filtered\_vowels) print(vowels) |
| --- |

Output

| >>> ('a', 'e', 'i', 'o') |
| --- |

Here, the ***filter()*** function extracts only the vowel letters from the letters list. Here's how this code works:

* Each element of the letters list is passed to the filter\_vowels() function.
* If ***filter\_vowels()*** returns **True**, that element is extracted otherwise it's filtered out.

Note: It's also possible to filter lists using a **loop**, however, using the ***filter()*** function is **much cleaner**.

Example 2: *Using Lambda Function Inside* ***filter()***

| numbers = [1, 2, 3, 4, 5, 6, 7]  *# the lambda function returns True for even numbers*  even\_numbers\_iterator = filter(lambda x: (x%2 == 0), numbers)  *# converting to list* even\_numbers = list(even\_numbers\_iterator)  print(even\_numbers) |
| --- |

Output

| >>> [2, 4, 6] |
| --- |

Here, we have directly passed a lambda function inside ***filter()***.

Our lambda function returns **True** for **even** numbers. Hence, the ***filter()*** function returns an iterator containing **even numbers only**.

Example 3: *Using None as a Function Inside* ***filter()***

| *# random list* random\_list = [1, 'a', 0, False, True, '0']  filtered\_iterator = filter(None, random\_list)  *#converting to list* filtered\_list = list(filtered\_iterator)  print(filtered\_list) |
| --- |

Output

| >>> [1, 'a', True, '0'] |
| --- |

When **None** is used as the first argument to the filter() function, all elements that are truthy values (gives True if converted to boolean) are extracted.

### **map()**

*The* ***map()*** *function applies a given function to each element of an* ***iterable*** *(****list****,* ***tuple*** *etc.) and returns an iterator containing the results.*

Example

| numbers = [2, 4, 6, 8, 10]  *# returns the square of a number* def square(number):  return number \* number  *# apply square() to each item of the numbers list* squared\_numbers\_iterator = map(square, numbers)  *# converting to list* squared\_numbers = list(squared\_numbers\_iterator) print(squared\_numbers)  *# Output: [4, 16, 36, 64, 100]* |
| --- |

SYNTAX and PARAMETERS

| map(function, iterable, ...) |
| --- |

* **function** - a function
* **iterable** - an iterable like sets, lists, tuples, etc

You can pass **more than one** iterable to the ***map()*** function.

RETURN VALUE

The ***map()*** function returns an object of map class. The returned value can be passed to functions like

* ***list()*** - to convert to **list**
* ***set()*** - to convert to a **set**, and so on

Example 1: *Working of* ***map()***

| def calculateSquare(n):  return n\*n   numbers = (1, 2, 3, 4) result = map(calculateSquare, numbers) print(result)  *# converting map object to set* numbersSquare = set(result) print(numbersSquare) |
| --- |

Output

| >>> <map object at 0x7f722da129e8> >>> {16, 1, 4, 9} |
| --- |

In the above example, each item of the tuple is squared.

Example 2: *How to use* ***lambda*** *function with* ***map()****?*

Since ***map()*** expects a function to be passed in, lambda functions are commonly used while working with ***map()*** functions.

A ***lambda*** function is a **short function without a name**. Learn more about lambda functions under [4.0 Functions](#_dcunidt1usbg)

| numbers = (1, 2, 3, 4) result = map(lambda x: x\*x, numbers) print(result)  *# converting map object to set* numbersSquare = set(result) print(numbersSquare) |
| --- |

Output

| >>> <map 0x7fafc21ccb00> >>> {16, 1, 4, 9} |
| --- |

There is functionally no difference between this and Example 1...

Example 3: *Passing* ***Multiple Iterators*** *to* ***map()*** *Using* ***Lambda***

In this example, corresponding items of **two lists** are added.

| num1 = [4, 5, 6] num2 = [5, 6, 7] result = map(lambda n1, n2: n1+n2, num1, num2) print(list(result)) |
| --- |

Output

| >>> [9, 11, 13] |
| --- |

## 6.5 Miscellaneous Functions

### **dir()**

*The* ***dir()*** *method returns the* ***list*** *of* ***valid attributes*** *of the passed object. Useful when using an* ***unknown function***

Example

| number = [12]  *# returns valid attributes of the number list*  print(dir(number))  *# Output: ['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']* |
| --- |

SYNTAX and PARAMETERS

| dir(object) |
| --- |

* **object** - can be an empty/filled **tuple**, **list**, **set**, **dictionary** etc or any **user-defined object (**or **function**)

RETURN VALUES

Returns the list of attributes of the object passed to the method

### **iter()**

*The* ***iter()*** *method returns an iterator for the given argument.*

Example

| *# list of vowels* phones = ['apple', 'samsung', 'oneplus'] phones\_iter = iter(phones)  print(next(phones\_iter))  print(next(phones\_iter))  print(next(phones\_iter))   *# Output:* *# apple* *# samsung* *# oneplus* |
| --- |

SYNTAX and PARAMETERS

| iter(object, sentinel [optional]) |
| --- |

* **object** - can be a **list**, **set**, **tuple**, etc.
* **sentinel** [optional] - a **special value** that is used to represent the **end** of a sequence

RETURN VALUES

* **iterator object** for the given argument **until the sentinel character is found**
* **TypeError** for a user-defined object that doesn't implement ***\_\_iter\_\_()***, and ***\_\_next\_\_()*** or ***\_\_getitem()\_\_***

Example 2: ***iter()*** *for custom objects*

| class PrintNumber:  def \_\_init\_\_(self, max):  self.max = max  *# iter() method in a class*  def \_\_iter\_\_(self):  self.num = 0  return self *# next() method in a class*   def \_\_next\_\_(self):  if(self.num >= self.max):  raise StopIteration  self.num += 1  return self.num  print\_num = PrintNumber(3)  print\_num\_iter = iter(print\_num) print(next(print\_num\_iter)) *# 1* print(next(print\_num\_iter)) *# 2* print(next(print\_num\_iter)) *# 3*  *# raises StopIteration* print(next(print\_num\_iter)) |
| --- |

Output

| >>> 1 >>> 2 >>> 3 >>> Traceback (most recent call last):  File "", line 23, in   File "", line 11, in \_\_next\_\_  StopIteration |
| --- |

In the example above, we have printed the iterator numbers **1**, **2**, **3** using the ***\_\_iter\_\_()*** and ***\_\_next\_\_()*** methods.

Here, the ***\_\_next()\_\_*** method here has a loop that runs till ***self.num*** is **greater than or equal**to ***self.max***

Since we have passed **3** as the parameter to the **PrintNumber()** object, ***self.max*** is **initialised** to **3**. Therefore, the **loop stops** at **3**.

When ***self.num*** reaches the value of ***self.max*** which is **3**, the ***next()*** method raises a **StopIteration *exception***.

Example 3: ***iter()*** *with Sentinel Parameter*

| class DoubleIt:   def \_\_init\_\_(self):  self.start = 1   def \_\_iter\_\_(self):  return self   def \_\_next\_\_(self):  self.start \*= 2  return self.start   \_\_call\_\_ = \_\_next\_\_   my\_iter = iter(DoubleIt(), 16)  for x in my\_iter:  print(x) |
| --- |

Output

| >>> 2 >>> 4 >>> 8 |
| --- |

In the above example, we haven't implemented a **StopIteration** condition.

Instead, we have used the ***iter()*** method with a ***sentinel* parameter** to stop the iteration:

| my\_iter = iter(DoubleIt(), 16) |
| --- |

The value of the ***sentinel* parameter** here is **16** so the program will stop when the value from the ***\_\_next\_\_()*** method is **equal to this number**.

At this point in the code, the program will raise a **StopIteration** automatically.

### **next()**

*The* ***next()*** *function returns the next item from the iterator.*

Example

| marks = [65, 71, 68, 74, 61]  *# convert list to iterator* iterator\_marks = iter(marks)  *# the next element is the first element* marks\_1 = next(iterator\_marks) print(marks\_1)  *# find the next element which is the second element* marks\_2 = next(iterator\_marks) print(marks\_2)  *# Output: 65* *# 71* |
| --- |

SYNTAX and PARAMETERS

| next(iterator, default) |
| --- |

* **iterator** - **next()** retrieves next item from the iterator
* **default** (optional) - this value is returned if the iterator is exhausted (there is **no** next item)

RETURN VALUE

* The ***next()*** function returns the **next item** from the iterator.
* If the iterator is **exhausted**, it returns the **default value** passed as an **argument**.
* If the default parameter is **omitted** and the iterator is **exhausted**, it raises the **StopIteration *exception***.

Example 1: Get the next item

| random = [5, 9, 'cat']  *# converting the list to an iterator* random\_iterator = iter(random) print(random\_iterator)  *# Output: 5* print(next(random\_iterator))  *# Output: 9* print(next(random\_iterator))  *# Output: 'cat'* print(next(random\_iterator))  *# This will raise Error* *# iterator is exhausted* print(next(random\_iterator)) |
| --- |

Output

| >>> <list\_iterator object at 0x7feb49032b00> >>> 5 >>> 9 >>> cat >>> Traceback (most recent call last):  File "python", line 18, in <module>  StopIteration |
| --- |

A ***list*** is an **iterable** and you can get its ***iterator*** from it by using the ***iter()*** function in Python.

Learn more about

* Python iterators
* Python [iter()](#_ndseznf1l1e0) function

We got an error from the last statement in the above program because we tried to get the next item when **no next item was available** (iterator is **exhausted**).

In such cases, you can give a **default** value as the ***second* parameter**.

Example 2: *Passing default value to* ***next()***

| random = [5, 9]  *# converting the list to an iterator* random\_iterator = iter(random)  *# Output: 5* print(next(random\_iterator, '-1'))  *# Output: 9* print(next(random\_iterator, '-1'))  *# random\_iterator is exhausted* *# Output: '-1'* print(next(random\_iterator, '-1')) print(next(random\_iterator, '-1')) print(next(random\_iterator, '-1')) |
| --- |

Output

| >>> 5 >>> 9 >>> -1 >>> -1 >>> -1 |
| --- |

### **len()**

*The* ***len()*** *function returns the number of items (length) in an object.*

Example

| languages = ['Python', 'Java', 'JavaScript']  *# compute the length of languages* length = len(languages) print(length)  *# Output: 3* |
| --- |

SYNTAX and PARAMETERS

| len(s) |
| --- |

* **sequence** - ***string***, ***bytes***, ***tuple***, ***list***, ***range*** OR,
* **collection** - ***dictionary***, ***set***, **frozen *set***

RETURN VALUES

* ***len()*** function returns the number of items of an object.
* Failing to pass an argument or passing an **invalid argument** will raise a **TypeError *exception***.

Example 1: *How* ***len()*** *works with* ***strings*** *and* ***bytes****?*

| testString = '' print('Length of', testString, 'is', len(testString))  testString = 'Python' print('Length of', testString, 'is', len(testString))  *# byte object* testByte = b'Python' print('Length of', testByte, 'is', len(testByte))  testList = [1, 2, 3]  *# converting to bytes object* testByte = bytes(testList) print('Length of', testByte, 'is', len(testByte)) |
| --- |

Output

| >>> Length of is 0 >>> Length of Python is 6 >>> Length of b'Python' is 6 >>> Length of b'\x01\x02\x03' is 3 |
| --- |

Example 2: *How* ***len()*** *works with* ***tuples****,* ***lists*** *and* ***range****?*

| testList = [] print(testList, 'length is', len(testList))  testList = [1, 2, 3] print(testList, 'length is', len(testList))  testTuple = (1, 2, 3) print(testTuple, 'length is', len(testTuple))  testRange = range(1, 10) print('Length of', testRange, 'is', len(testRange)) |
| --- |

Output

| >>> [] length is 0 >>> [1, 2, 3] length is 3 >>> (1, 2, 3) length is 3 >>> Length of range(1, 10) is 9 |
| --- |

Example 3: *How* ***len()*** *works with* ***dictionaries*** *and* ***sets****?*

| testSet = {1, 2, 3} print(testSet, 'length is', len(testSet))  *# Empty Set* testSet = set() print(testSet, 'length is', len(testSet))  testDict = {1: 'one', 2: 'two'} print(testDict, 'length is', len(testDict))  testDict = {} print(testDict, 'length is', len(testDict))  testSet = {1, 2} *# frozenSet* frozenTestSet = frozenset(testSet) print(frozenTestSet, 'length is', len(frozenTestSet)) |
| --- |

Output

| >>> {1, 2, 3} length is 3 >>> set() length is 0 >>> {1: 'one', 2: 'two'} length is 2 >>> {} length is 0 >>> frozenset({1, 2}) length is 2 |
| --- |

Internally, len() calls the object's \_\_len\_\_ method. You can think of len() as:

| def len(s):  return s.\_\_len\_\_() |
| --- |

So, you can assign custom length to the object (if necessary)

Example 4: *How* ***len()*** *works for custom objects?*

| class Session:  def \_\_init\_\_(self, number = 0):  self.number = number    def \_\_len\_\_(self):  return self.number   *# default length is 0* s1 = Session() print(len(s1))  *# giving custom length* s2 = Session(6) print(len(s2)) |
| --- |

Output

| >>> 0 >>> 6 |
| --- |

### **range()**

*The* ***range()*** *function returns a sequence of numbers between the given range.*

Example

| *# create a sequence of numbers from 0 to 3* numbers = range(4)  *# iterating through the sequence of numbers* for i in numbers:  print(i)  *# Output:*  *# 0* *# 1* *# 2* *# 3* |
| --- |

Note: ***range()*** returns an immutable sequence of numbers that can be easily converted to ***lists***, ***tuples***, ***sets*** etc.

SYNTAX and PARAMETERS

| range(start, stop, step) |
| --- |

The ***start*** and ***step*** parameters in ***range()*** are optional.

Example 1: ***range()*** *with* ***Stop*** *Argument*

If we pass a single argument to ***range()***, it means we are passing the ***stop*** argument.

In this case, ***range()*** returns a sequence of numbers starting from **0** up to the **number** (but **not including** the number).

| *# numbers from 0 to 3 (4 is not included)* numbers = range(4) print(list(numbers)) *#Output: [0, 1, 2, 3]*  *# if 0 or negative number is passed, we get an empty sequence* numbers = range(-4) print(list(numbers)) *#Output: []* |
| --- |

Example 2: ***range()*** *with* ***Start*** *and* ***Stop*** *Arguments*

If we pass two arguments to ***range()***, it means we are passing ***start*** and ***stop*** arguments.

In this case, **range()** returns a sequence of numbers starting from **start** (**inclusive**) up to **stop** (**exclusive**).

| *# numbers from 2 to 4 (5 is not included)* numbers = range(2, 5) print(list(numbers)) *#Output: [2, 3, 4]*  *# numbers from -2 to 3 (4 is not included)* numbers = range(-2, 4)  print(list(numbers)) *#Output: [-2, -1, 0, 1, 2, 3]*  *# returns an empty sequence of numbers* numbers = range(4, 2)  print(list(numbers)) *#Output: []* |
| --- |

Example 3: ***range()*** *with* ***Start****,* ***Stop*** *and* ***Step*** *Arguments*

If we pass all three arguments,

* the *first* argument is ***start***
* the *second* argument is ***stop***
* the *third* argument is ***step***

The ***step*** argument specifies the **incrementation** between two numbers in the sequence.

| *# numbers from 2 to 10 with increment 3 between numbers* numbers = range(2, 10, 3) print(list(numbers)) *#Output: [2, 5, 8]*  *# numbers from 4 to -1 with increment of -1* numbers = range(4, -1, -1)  print(list(numbers)) *#Output: [4, 3, 2, 1, 0]*  *# numbers from 1 to 4 with increment of 1* *# range(0, 5, 1) is equivalent to range(5)* numbers = range(0, 5, 1)  print(list(numbers)) *#Output: [0, 1, 2, 3, 4]* |
| --- |

Note: The default value of start is **0**, and the default value of step is **1**. That's why ***range(*0, 5, 1*)*** is equivalent to ***range(*5*)***.

*range()* in for Loop

The ***range()*** function is commonly used in a ***for*** loop to iterate the loop a ***certain number of times***. For example,

| *# iterate the loop 5 times* for i in range(5):  print(i, 'Hello') |
| --- |

Output

| >>> 0 Hello >>> 1 Hello >>> 2 Hello >>> 3 Hello >>> 4 Hello |
| --- |

### **reversed()**

*The* ***reversed()*** *method computes the* ***reverse*** *of a given sequence object and returns it in the form of a* ***list****.*

Example

| seq\_string = 'Python'  *# reverse of a string* print(list(reversed(seq\_string)))  *# Output: ['n', 'o', 'h', 't', 'y', 'P']* |
| --- |

SYNTAX and PARAMETERS

| reversed(sequence\_object) |
| --- |

* ***sequence\_object*** - an **indexable** object to be reversed (can be a ***tuple***, ***string***, ***list***, ***range***, etc.)

Note: Since we can't index objects such as a ***set*** and a ***dictionary***, they are **not considered sequence objects**.

RETURN VALUE

* a **reversed list** of items present in a **sequence object**

Example 1: *Python* ***reversed()*** *with Built-In Sequence Objects*

| seq\_tuple = ('P', 'y', 't', 'h', 'o', 'n')  *# reverse of a tuple object* print(list(reversed(seq\_tuple)))  seq\_range = range(5, 9)  *# reverse of a range* print(list(reversed(seq\_range)))  seq\_list = [1, 2, 4, 3, 5]  *# reverse of a list* print(list(reversed(seq\_list))) |
| --- |

Output

| >>> ['n', 'o', 'h', 't', 'y', 'P'] >>> [8, 7, 6, 5] >>> [5, 3, 4, 2, 1] |
| --- |

In the above example, we have used the ***reversed()*** method with objects like ***tuple***, ***range*** and a ***list***.

When using the ***reversed()*** method with these objects, we need to use the ***list()*** method to convert the output from the ***reversed()*** method to a ***list***.

Example 2: ***reversed()*** *with Custom Objects*

| class Vowels:  vowels = ['a', 'e', 'i', 'o', 'u']   def \_\_reversed\_\_(self):  return reversed(self.vowels)  v = Vowels()  *# reverse a custom object v* print(list(reversed(v))) |
| --- |

Output

| ['u', 'o', 'i', 'e', 'a'] |
| --- |

In the above example, we have used the ***reversed()*** method with a **custom object *v*** of the Vowels ***class***.

Here, the method returns the **reverse order** of the sequence in the ***vowels* list**.

### **sorted()**

*The* ***sorted()*** *function* ***sorts*** *the elements of a given iterable in a specific order (****ascending*** *or* ***descending****) and returns it as a* ***list****.*

Example

| numbers = [4, 2, 12, 8]  sorted\_numbers = sorted(numbers) print(sorted\_numbers)  *# Output: [2, 4, 8, 12]* |
| --- |

SYNTAX and PARAMETERS

| sorted(iterable, key=None, reverse=False) |
| --- |

* **iterable** - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.
* **reverse** (Optional) - If **True**, the sorted list is **reversed** (or sorted in **descending order**). Defaults to **False** if not provided.
* **key** (Optional) - A function that serves as a ***key*** for the sort comparison. Defaults to **None**.

Example 1: *Sort* ***string****,* ***list****, and* ***tuple***

| *# vowels list* py\_list = ['e', 'a', 'u', 'o', 'i'] print(sorted(py\_list))  *# string* py\_string = 'Python' print(sorted(py\_string))  *# vowels tuple* py\_tuple = ('e', 'a', 'u', 'o', 'i') print(sorted(py\_tuple)) |
| --- |

Output

| >>> ['a', 'e', 'i', 'o', 'u'] >>> ['P', 'h', 'n', 'o', 't', 'y'] >>> ['a', 'e', 'i', 'o', 'u'] |
| --- |

Notice that in all cases that a sorted list is returned.

Note: A ***list*** also has the ***sort()*** method which performs the same way as ***sorted()***. The only difference is that the ***sort()*** method doesn't **return any value** and **changes the original list**.

Example 2: *Sort in descending order*

The ***sorted()*** function accepts a ***reverse*** parameter as an optional argument.

Setting ***reverse* = True** sorts the iterable in the ***descending*** order.

| *# set* py\_set = {'e', 'a', 'u', 'o', 'i'} print(sorted(py\_set, reverse=True))  *# dictionary* py\_dict = {'e': 1, 'a': 2, 'u': 3, 'o': 4, 'i': 5} print(sorted(py\_dict, reverse=True))  *# frozen set* frozen\_set = frozenset(('e', 'a', 'u', 'o', 'i')) print(sorted(frozen\_set, reverse=True)) |
| --- |

Output

| >>> ['u', 'o', 'i', 'e', 'a'] >>> ['u', 'o', 'i', 'e', 'a'] >>> ['u', 'o', 'i', 'e', 'a'] |
| --- |

key Parameter in Python sorted() function

**If you want your own implementation** for ***sorting***, ***sorted()*** also accepts a ***key* function** as an optional parameter.

Based on the returned value of the key function, you can sort the given iterable.

| sorted(iterable, key=len) |
| --- |

Here, ***len()*** is Python's in-built function to count the **length** of an object.

The list is sorted based on the **length** of the element, from the **lowest** count to **highest**.

Example 3: *Sort the list using* ***sorted()*** *having a* ***key*** *function*

| *# take the second element for sort* def take\_second(elem):  return elem[1]   *# random list* random = [(2, 2), (3, 4), (4, 1), (1, 3)]  *# sort list with key* sorted\_list = sorted(random, key=take\_second)  *# print list* print('Sorted list:', sorted\_list) |
| --- |

Output

| Sorted list: [(4, 1), (2, 2), (1, 3), (3, 4)] |
| --- |

Example 4: *Sorting with multiple keys*

Let us suppose that we have the following list:

| *# Nested list of student's info in a Science Olympiad* *# List elements: (Student's Name, Marks out of 100, Age)*  participant\_list = [  ('Alison', 50, 18),  ('Terence', 75, 12),  ('David', 75, 20),  ('Jimmy', 90, 22),  ('John', 45, 12) ] |
| --- |

We want to sort the list in such a way that the **student with the highest marks is in the beginning**. In case the students have ***equal marks***, they must be sorted so that the **younger participant comes *first***.

We can achieve this type of sorting with **multiple *keys***by returning ***tuples*** instead of a ***number***.

**Two tuples** can be compared by comparing their **elements** starting from first. If there is a ***tie*** (elements are **equal**), the ***second*** element is compared, and so on.

| >>> (1,3) > (1, 4) False >>> (1, 4) < (2,2) True >>> (1, 4, 1) < (2, 1) True |
| --- |

Let's use this logic to build our sorting logic.

| *# Nested list of student's info in a Science Olympiad* *# List elements: (Student's Name, Marks out of 100 , Age)* participant\_list = [  ('Alison', 50, 18),  ('Terence', 75, 12),  ('David', 75, 20),  ('Jimmy', 90, 22),  ('John', 45, 12) ]   def sorter(item):  *# Since highest marks first, least error = most marks*  error = 100 - item[1]  age = item[2]  return (error, age)   sorted\_list = sorted(participant\_list, key=sorter) print(sorted\_list) |
| --- |

Output

| [('Jimmy', 90, 22), ('Terence', 75, 12), ('David', 75, 20), ('Alison', 50, 18), ('John', 45, 12)] |
| --- |

Since the sorting logic function is small and fits in one line, ***lambda*** function is used inside the ***key*** rather than passing a separate function name.

The above program can be rewritten using the ***lambda*** function in the following way:

| *# Nested list of student's info in a Science Olympiad* *# List elements: (Student's Name, Marks out of 100 , Age)* participant\_list = [  ('Alison', 50, 18),  ('Terence', 75, 12),  ('David', 75, 20),  ('Jimmy', 90, 22),  ('John', 45, 12) ]  sorted\_list = sorted(participant\_list, key=lambda item: (100-item[1], item[2])) print(sorted\_list) |
| --- |

Output

| [('Jimmy', 90, 22), ('Terence', 75, 12), ('David', 75, 20), ('Alison', 50, 18), ('John', 45, 12)] |
| --- |

To learn more about lambda functions, visit Python Lambda Functions.

### slice()

*The* ***slice()*** *function returns a slice object that is used to slice any sequence (****string****,* ***tuple****,* ***list****,* ***range****, or* ***bytes****).*

Example

| text = 'Python Programing'  *# get slice object to slice Python* sliced\_text = slice(6) print(text[sliced\_text])  *# Output: Python* |
| --- |

SYNTAX and PARAMETERS

| slice(start, stop, step) |
| --- |

* **start** (optional) - Starting integer where the slicing of the object **starts**. Default to **None** if not provided.
* **stop** - Integer **until** which the slicing takes place. The slicing stops at index stop **-1** (last element).
* **step** (optional) - Integer value which determines the **increment** between **each index** for slicing. Defaults to **None** if not provided.

RETURN VALUE

***slice()*** returns a ***slice* objec**t.

Note: We can use slice with **any *object***which supports ***sequence protocol*** (implements \_\_getitem\_\_() and \_\_len()\_\_ method).

Example 1: *Create a slice object for slicing*

| *# contains indices (0, 1, 2)* result1 = slice(3) print(result1)  *# contains indices (1, 3)* result2 = slice(1, 5, 2) print(slice(1, 5, 2)) |
| --- |

Output

| >>> slice(None, 3, None) >>> slice(1, 5, 2) |
| --- |

Here, ***result1*** and ***result2*** are ***slice* objects**.

Now we know about slice objects, let's see how we can get ***substring***, ***sub-list***, ***sub-tuple***, etc. from ***slice* objects**.

Example 2: *Get substring using slice object*

| *# Program to get a substring from the given string*   py\_string = 'Python'  *# stop = 3* *# contains 0, 1 and 2 indices* slice\_object = slice(3)  print(py\_string[slice\_object]) *# Pyt*  *# start = 1, stop = 6, step = 2* *# contains 1, 3 and 5 indices* slice\_object = slice(1, 6, 2) print(py\_string[slice\_object]) *# yhn* |
| --- |

Output

| >>> Pyt >>> yhn |
| --- |

Example 3: *Get substring using negative index*

| py\_string = 'Python'  *# start = -1, stop = -4, step = -1* *# contains indices -1, -2 and -3* slice\_object = slice(-1, -4, -1)  print(py\_string[slice\_object]) *# noh* |
| --- |

Output

| >>> noh |
| --- |

Example 4: *Get sublist and sub-tuple*

| py\_list = ['P', 'y', 't', 'h', 'o', 'n'] py\_tuple = ('P', 'y', 't', 'h', 'o', 'n')  *# contains indices 0, 1 and 2* slice\_object = slice(3) print(py\_list[slice\_object]) *# ['P', 'y', 't']*  *# contains indices 1 and 3* slice\_object = slice(1, 5, 2) print(py\_tuple[slice\_object]) *# ('y', 'h')* |
| --- |

Output

| >>> ['P', 'y', 't'] >>> ('y', 'h') |
| --- |

Example 5: *Get sublist and sub-tuple using negative index*

| py\_list = ['P', 'y', 't', 'h', 'o', 'n'] py\_tuple = ('P', 'y', 't', 'h', 'o', 'n')  *# contains indices -1, -2 and -3* slice\_object = slice(-1, -4, -1)  print(py\_list[slice\_object]) *# ['n', 'o', 'h']*  *# contains indices -1 and -3* slice\_object = slice(-1, -5, -2) print(py\_tuple[slice\_object]) *# ('n', 'h')* |
| --- |

Output

| >>> ['n', 'o', 'h'] >>> ('n', 'h') |
| --- |

Example 6: *Using Indexing Syntax for Slicing*

The ***slice* object** can be substituted with the **indexing syntax** in Python.

You can alternately use the following syntax for slicing:

| obj[start:stop:step] |
| --- |

For example,

| py\_string = 'Python'  *# contains indices 0, 1 and 2* print(py\_string[0:3]) *# Pyt*  *# contains indices 1 and 3* print(py\_string[1:5:2]) *# yh* |
| --- |

Output

| >>> Pyt >>> yh |
| --- |

### **locals()**

*The* ***locals()*** *method returns a dictionary with all the local variables and symbols for the current program.*

Example

| print(locals()) |
| --- |

Output

| {'In': ['', 'locals()'],  'Out': {},  '\_': '',  '\_\_': '',  '\_\_\_': '',  '\_\_builtin\_\_': ,  '\_\_builtins\_\_': ,  '\_\_name\_\_': '\_\_main\_\_',  '\_dh': ['/home/repl'],  '\_i': '',  '\_i1': 'locals()',  '\_ih': ['', 'locals()'],  '\_ii': '',  '\_iii': '',  '\_oh': {},  '\_sh': ,  'exit': ,  'get\_ipython': >,  'quit': } |
| --- |

Example 1: *Python* ***locals()***

| class local:  l = 50  *# locals inside a class*  print('\nlocals() value inside class\n', locals()) |
| --- |

Output

| >>> locals() value inside class  {'\_\_module\_\_': '\_\_main\_\_', '\_\_qualname\_\_': 'PLocal', 'l': 50} |
| --- |

Python compiler maintains a **symbol table** which contains the necessary information about the program being written. There are two types of symbol tables in Python - **Local** and **Global**.

A Local Symbol table stores all the information related to the program's **local scope** (within the **class** or a **method**). We can access this **symbol table** with the ***locals()*** method.

Typically, python programmers use the ***locals()*** method to **restrict any *variable* and *method* inside the scope of a *method* or a *class***.

In the above example, we have a class named ***local***. Here, we have used the ***locals()*** method to return the variables and methods of this class.

Example 2: ***locals()*** *to change values*

| def localsPresent():  present = True  print(present)  locals()['present'] = False;  print(present)  localsPresent() |
| --- |

Output

| >>> True >>> True |
| --- |

In the above example, we have changed the **value** of the ***present*** variable inside a function ***localsPresent*** using the ***locals()*** method.

Since ***locals()*** returns a ***dictionary***, we have used a ***method* with a *dictionary* item** i.e. the variable ***present*** and changed its **value** to **False**.

### **input()**

*The* ***input()*** *function takes input from the user and returns it.*

Example

| name = input("Enter your name: ") print(name)  *# Output:*  *# Enter your name: James* *# James* |
| --- |

SYNTAX and PARAMETERS

| input([prompt]) |
| --- |

* prompt (Optional) - a string that is written to standard output (usually screen) without trailing newline

RETURN VALUES

The ***input()*** function reads a line from the input (usually from the user), converts the line into a ***string*** by removing the trailing *newline*, and returns it.

If ***EOF*** is read, it raises an **EOFError** exception.

Example 1: *How* ***input()*** *works in Python?*

| *# get input from user*  inputString = input()  print('The inputted string is:', inputString) |
| --- |

Output

| >>> Python is interesting. >>> The inputted string is: Python is interesting |
| --- |

Example 2: *Get input from user with a prompt*

| *# get input from user*  inputString = input('Enter a string: ')  print('The inputted string is: ', inputString) |
| --- |

Output

| >>> Enter a string: Python is interesting. >>> The inputted string is: Python is interesting |
| --- |

### **repr()**

*The* ***repr()*** *function returns a printable representation of the given object.*

Example

| numbers = [1, 2, 3, 4, 5]  *# create a printable representation of the list* printable\_numbers = repr(numbers) print(printable\_numbers)  *# Output: [1, 2, 3, 4, 5]* |
| --- |

SYNTAX and PARAMETERS

| repr(obj) |
| --- |

* **obj** - the object whose printable representation has to be returned

RETURN VALUE

The ***repr()*** function returns a **printable** representational ***string*** of the given object.

Example 1: *How* ***repr()*** *works in Python?*

| var = 'foo'  print(repr(var)) |
| --- |

Output

| >>> 'foo' |
| --- |

Here, we assign a value 'foo' to var. Then, the repr() function returns "'foo'", 'foo' inside double-quotes.

When the result from repr() is passed to eval(), we will get the original object (for many types).

| >>> eval(repr(var)) 'foo' |
| --- |

Example 2: *Implement* ***\_\_repr\_\_()*** *for custom objects*

Internally, ***repr()*** function calls ***\_\_repr\_\_()*** of the given object.

You can easily implement/override ***\_\_repr\_\_()*** so that ***repr()*** works differently.

| class Person:  name = 'Adam'   def \_\_repr\_\_(self):  return repr('Hello ' + self.name )  print(repr(Person())) |
| --- |

Output

| >>> 'Hello Adam' |
| --- |

### **format()**

*The* ***format()*** *method returns a formatted representation of the given value controlled by the format specifier.*

Example

| value = 45 *# format the integer to binary* binary\_value = format(value, 'b') print(binary\_value)  *# Output: 101101* |
| --- |

SYNTAX and PARAMETERS

| format(value[, format\_spec]) |
| --- |

* **value** - value that needs to be formatted
* **format\_spec** - The specification on how the value should be formatted.

The format specifier could be in the format:

| [[fill]align][sign][#][0][width][,][.precision][type] where, the options are fill ::= any character align ::= "<" | ">" | "=" | "^" sign ::= "+" | "-" | " " width ::= integer precision ::= integer type ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%" |
| --- |

Visit these entries to learn more about [format types and alignment](#_zg6iimnh4xcp).

RETURN VALUE

The ***format()*** function returns a formatted representation of a given value specified by the ***format specifier***.

Example 1: *Number formatting with* ***format()***

| *# d, f and b are type*  *# integer* print(format(123, "d"))  *# float arguments* print(format(123.4567898, "f"))  *# binary format* print(format(12, "b")) |
| --- |

Output

| >>> 123 >>> 123.456790 >>> 1100 |
| --- |

Example 2: *Formatting w/* ***fill****,* ***align****,* ***sign****,* ***width****,* ***precision*** *and* ***type***

| *# integer*  print(format(1234, "\*>+7,d"))  *# float number* print(format(123.4567, "^-09.3f")) |
| --- |

Output

| >>> \*+1,234 >>> 0123.4570 |
| --- |

Here, when formatting the integer **1234**, we've specified the formatting specifier ***\*>+7,d***. Let's understand each option:

* **\*** - It is the fill character that fills up the empty spaces after formatting
* **>** - It is the ***right alignment*** option that aligns the output string to the right
* **+** - It is the sign option that ***forces the number to be signed*** (having a sign on its left)
* **7** - It is the ***width*** option that forces the number to take a minimum width of **7**, other spaces will be filled by ***fill* character**
* **,** - It is the ***thousands* operator** that places a comma between all ***thousands*** (i.e. 1,000,000 instead of 1000000)
* **d** - It is the type option that specifies the number is an integer.

When formatting the **floating point** number **123.4567**, we've specified the format specifier ***^-09.3f***. These are:

* **^** - It is the ***centre alignment*** option that aligns the output string to the centre of the remaining space
* **-** - It is the ***sign*** option that forces ***only negative numbers to show the sign***
* **0** - It is the character that is placed **in place of** the ***empty spaces***.
* **9** - It is the width option that sets the **minimum *width***of the number to **9** (including decimal point, thousands comma and sign)
* **.3** - It is the ***precision*** operator that sets the precision of the given floating number to ***3 decimal places***
* f - It is the type option that specifies the number is a float.

# 7.0 External Libraries

*External libraries can be imported via the* ***import*** *method*

SYNTAX

| *#In this example, we will be trying to print pi* *#Notice how we have to use the math. prefix to get the value of pi*  try:  import math  print(math.pi)  *#Now notice how we shortened the prefix to m. instead of math. We can even get the value of e using the same m. prefix*  import math as m  print(m.pi)  print(m.e)  *#Now, we don't even need a prefix to get the value of pi, but we only get access to the value of pi (as shown by 'e' returning an error)*  from math import pi  print(pi)  print(e)  *#For those masochist out there who prefer PI instead of pi, this is for y'all*  finally:  from math import pi as PI  print(PI)  *#Keep in mind that a newer import statement will overwrite the older statement ASSUMING they are importing the same thing (E.g. from math import pi and from math import pi as PI* |
| --- |

Output

| >>> 3.141592653589793 >>> 3.141592653589793 >>> 2.718281828459045 >>> 3.141592653589793 >>> Traceback (most recent call last):  File "C:\Users\gjh02\OneDrive\Desktop\Programming 1\testing.py", line 14, in <module>  print(e)  ^  NameError: name 'e' is not defined >>> 3.141592653589793 |
| --- |

## 7.1 Regular Expressions

A **Reg**ular **Ex**pression (RegEx) is a sequence of characters that defines a search pattern. For example,

| ^a...s$ |
| --- |

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

A pattern defined using ***RegEx*** can be used to match against a string.

For example,

| Expression | String | Matched? |
| --- | --- | --- |
|  | abs | No match |
|  | alias | Match |
| ^a...s$ | abyss | Match |
|  | Alias | No Match |
|  | An abacus | No Match |

Python has a module named ***re*** to work with **RegEx**. Here's an example:

| import re  pattern = '^a...s$' test\_string = 'abyss' result = re.match(pattern, test\_string)  if result:  print("Search successful.") else:  print("Search unsuccessful.") |
| --- |

Here, we used ***re.match()*** function to search ***pattern*** within the ***test\_string***. The method returns a match object if the search is successful. If not, it returns **None**.

There are other several functions defined in the ***re*** module to work with RegEx. Before we explore that, let's learn about regular expressions themselves.

If you already know the basics of **RegEx**, jump to [Python RegEx Methods](#_u7gadc5pllly).

Specify Pattern Using RegEx

To specify regular expressions, metacharacters are used. In the above example, **^** and **$** are ***metacharacters***.

MetaCharacters

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

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

**[] - Square brackets**

*Square brackets specify a set of characters you wish to match.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | a | 1 Match |
| [abc] | ac | 2 Match |
|  | Hey Jude | No Match |
|  | abc de ca | 5 Match |

Here, **[abc]** will match if the string you are trying to match contains any of the ***a***, ***b*** or ***c***.

You can also specify a **range of characters** using ‘***-***’ inside square brackets.

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

You can complement (invert) the character set by using caret ‘***^***’ symbol at the start of a square-bracket.

* **[^abc]** means any character except ***a*** or ***b*** or ***c***.
* **[^0-9]** means any non-digit character.

**. - Period**

*A period matches any single character (except newline* ***'\n'****).*

| Expression | String | Matched? |
| --- | --- | --- |
|  | a | No Match |
| .. | ac | 1 Match |
|  | acd | 1 Match |
|  | acde | 2 Match (4 characters) |

In other words, it counts how many pairs of characters there are

**^ - Caret**

*The caret symbol* ***^*** *checks if a string* ***starts with*** *a certain character.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | a | 1 Match |
| ^a | abc | 1 Match |
|  | bac | No Match |

| ^ab | abc | 1 Match |
| --- | --- | --- |
|  | acb | No Match (starts with a but not  followed by b) |

**$ - Dollar**

*The dollar symbol* ***$*** *checks if a string* ***ends with*** *a certain character.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | a | 1 Match |
| a$ | formula | 1 Match |
|  | cab | No Match |

**\* - Star**

*The star symbol* ***\**** *matches* ***zero or more occurrences*** *of the pattern* ***left*** *to it.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | mn | 1 Match |
|  | man | 1 Match |
| ma\*n | maaan | 1 Match |
|  | main | No Match (a is not followed by n) |
|  | woman | 1 Match |

**+ - Plus**

*The plus symbol* ***+*** *matches* ***one or more occurrences*** *of the pattern left to it.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | mn | No Match (No ‘a’ character) |
|  | man | 1 Match |
| ma+n | maaan | 1 Match |
|  | main | No Match (a is not followed by n) |
|  | woman | 1 Match |

**? - Question Mark**

*The question mark symbol* ***?*** *matches* ***zero or one occurrence*** *of the pattern left to it.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | mn | 1 Match |
|  | man | 1 Match |
| ma?n | maaan | No Match (more than one ‘a’ character) |
|  | main | No Match (a is not followed by n) |
|  | woman | 1 Match |

**{} - Braces**

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

| Expression | String | Matched? |
| --- | --- | --- |
|  | abc dat | No match |
| a{2,3} | abc daat | 1 match (at daat) |
|  | aabc daaat | 2 matches (at aabc and daaat) |
|  | aabc daaaat | 2 matches (at aabc and daaaat) |

Let's try one more example. This RegEx [0-9]{2, 4} matches at least 2 digits but not more than 4 digits

| Expression | String | Matched? |
| --- | --- | --- |
|  | ab123csde | 1 match (match at ab123csde) |
| [0-9]{2,4} | 12 and 345673 | 3 matches (12, 3456, 73) |
|  | 1 and 2 | No Match |

**| - Alternation**

*Vertical bar* **|** *is used for alternation (****or*** *operator).*

| Expression | String | Matched? |
| --- | --- | --- |
|  | cde | No Match |
| a|b | ade | 1 match (match at ade) |
|  | acdbea | 3 matches (at acdbea) |

Here, a|b match any string that contains either a or b

**() - Group**

*Parentheses () 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***

| Expression | String | Matched? |
| --- | --- | --- |
|  | ab xz | No match |
| (a|b|c)xz | abxz | 1 match (match at abxz) |
|  | axz cabxz | 2 matches (at axzbc cabxz) |

**\ - Backslash**

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

For example,

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

If you are unsure if a character has a special meaning or not, you can put **\** in front of it. This makes sure the character is **not treated in a special way**.

**Special Sequences**

Special sequences make commonly used patterns easier to write. Here's a list of special sequences:

**\A** - *Matches if the specified characters are at the* ***start*** *of a string.*

| Expression | String | Matched? |
| --- | --- | --- |
| \Athe | the sun | Match |
|  | In the sun | No Match |

**\b** - *Matches if the specified characters are at the* ***beginning or end*** *of a word.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | football | Match |
| \bfoo | a football | Match |
|  | afootball | No Match |

|  | the foo | Match |
| --- | --- | --- |
| foo\b | the afoo test | Match |
|  | the afootest | No Match |

**\B** - ***Opposite*** *of \b. Matches if the specified characters are* ***not at the beginning or end*** *of a word.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | football | No Match |
| \bfoo | a football | No Match |
|  | afootball | Match |

|  | the foo | No Match |
| --- | --- | --- |
| foo\b | the afoo test | No Match |
|  | the afootest | Match |

**\d** - *Matches any* ***decimal digit****. Equivalent to* ***[0-9]***

| Expression | String | Matched? |
| --- | --- | --- |
| \d | 12abc3 | 3 matches (at 12abc3) |
|  | Python | No match |

**\D** - *Matches any* ***non-decimal digit****. Equivalent to* ***[^0-9]***

| Expression | String | Matched? |
| --- | --- | --- |
| \d | 1ab34"50 | 3 matches (at 1ab34"50) |
|  | 1345 | No match |

**\s** - *Matches where a string contains any* ***whitespace*** *character. Equivalent to* ***[ \t\n\r\f\v]****.*

| Expression | String | Matched? |
| --- | --- | --- |
| \s | Python RegEx | 1 Match |
|  | PythonRegEx | No Match |

**\S** - *Matches where a string contains any* ***non-whitespace*** *character. Equivalent to* ***[^ \t\n\r\f\v]****.*

| Expression | String | Matched? |
| --- | --- | --- |
| \S | a b | Match |
|  | *(assume it’s space ONLY)* | No Match |

**\w** - *Matches any* ***alphanumeric character*** *(digits and alphabets). Equivalent to* ***[a-zA-Z0-9\_]****. By the way,* ***underscore \_ is also considered an alphanumeric character****.*

| Expression | String | Matched? |
| --- | --- | --- |
| \w | 12&": ;c | 3 matches (at 12&": ;c) |
|  | %"> ! | No match |

**\W** - *Matches any* ***non-alphanumeric*** *character. Equivalent to* ***[^a-zA-Z0-9\_]***

| Expression | String | Matched? |
| --- | --- | --- |
| \W | 1a2%c | 1 match (at 1a2%c) |
|  | Python | No Match |

**\Z** - *Matches if the specified characters are at the* ***end*** *of a string.*

| Expression | String | Matched? |
| --- | --- | --- |
|  | I like Python | 1 Match |
| Python\Z | I like Python Programming | No Match |
|  | Python is not fun. | No Match |

Tip: To build and test regular expressions, you can use RegEx tester tools such as ***regex101***. This tool not only helps you in creating regular expressions, but it also helps you learn it.

Now you understand the basics of RegEx, let's discuss how to use **RegEx in your Python code**.

### **Python RegEx**

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

| import re |
| --- |

The module defines several functions and constants to work with RegEx.

#### **re.findall()**

*The* ***re.findall()*** *method returns a list of strings containing all matches.*

Example: ***re.findall()***

| *# Program to extract numbers from a string*  import re  string = 'hello 12 hi 89. Howdy 34' pattern = '\d+'  result = re.findall(pattern, string)  print(result) |
| --- |

Output

| >>> ['12', '89', '34'] |
| --- |

If the pattern is not found, ***re.findall()*** returns an **empty list**.

#### **re.split()**

*The* ***re.split*** *method splits the string where there is a match and returns a list of strings where the splits have occurred.*

Example: ***re.split()***

| import re  string = 'Twelve:12 Eighty nine:89.' pattern = '\d+'  result = re.split(pattern, string)  print(result) |
| --- |

Output

| >>> ['Twelve:', ' Eighty nine:', '.'] |
| --- |

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

You can pass ***maxsplit*** argument to the ***re.split()*** method. It's the **maximum number of splits that will occur**.

| import re  string = 'Twelve:12 Eighty nine:89 Nine:9.' pattern = '\d+'  *# maxsplit = 1* *# split only at the first occurrence* result = re.split(pattern, string, 1)  print(result) |
| --- |

Output

| >>> ['Twelve:', ' Eighty nine:89 Nine:9.'] |
| --- |

By the way, the default value of ***maxsplit*** is **0**; meaning **all possible splits**.

#### re.sub()

SYNTAX

| re.sub(pattern, replace, string) |
| --- |

The method returns a string where matched occurrences are replaced with the content of ***replace*** variable.

Example: ***re.sub()***

| *# Program to remove all whitespaces* import re  *# multiline string* string = 'abc 12\ de 23 \n f45 6'  *# matches all whitespace characters* pattern = '\s+'  *# empty string* replace = ''  new\_string = re.sub(pattern, replace, string)  print(new\_string) |
| --- |

Output

| >>> abc12de23f456 |
| --- |

If the pattern is not found, ***re.sub()*** returns the **original string**.

You can pass ***count*** as a **fourth** parameter to the ***re.sub()*** method. If omitted, it results to **0**. This will **replace all occurrences**.

| import re  *# multiline string* string = 'abc 12\ de 23 \n f45 6'  *# matches all whitespace characters* pattern = '\s+' replace = ''  new\_string = re.sub(r'\s+', replace, string, 1)  print(new\_string) |
| --- |

Output

| >>> abc12de 23 >>> f45 6 |
| --- |

#### **\*re.subn()**

*The* ***re.subn()*** *is similar to* ***re.sub()*** *except it returns a tuple of 2 items containing the new string and the number of substitutions made.*

Example: ***re.subn()***

| *# Program to remove all whitespaces* import re  *# multiline string* string = 'abc 12\ de 23 \n f45 6'  *# matches all whitespace characters* pattern = '\s+'  *# empty string* replace = ''  new\_string = re.subn(pattern, replace, string)  print(new\_string) |
| --- |

Output

| >>> ('abc12de23f456', 4) |
| --- |

#### re.search()

*The* ***re.search()*** *method takes* ***two*** *arguments: a* ***pattern*** *and a* ***string****. The method looks for the* ***first location*** *where the* ***RegEx pattern produces a match with the string****.*

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

| match = re.search(pattern, str) |
| --- |

Example: ***re.search()***

| import re  string = "Python is fun"  *# 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 |
| --- |

#### **Match object**

*You can get* ***methods*** *and* ***attributes*** *of a* ***match object*** *using* ***dir()*** *function.*

Some of the commonly used methods and attributes of match objects are:

##### **match.group()**

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

Example: Match object

| import re  string = '39801 356, 2102 1111'  *# Three digit number followed by space followed by two digit number* pattern = '(\d{3}) (\d{2})' *# match variable contains a Match object.* match = re.search(pattern, string)  if match:  print(match.group()) else:  print("pattern not found") |
| --- |

Output

| > 801 35 |
| --- |

Here, ***match*** variable contains a match object.

Our pattern **(\d{3}) (\d{2})** has two subgroups ***(\d{3})*** and ***(\d{2})***. You can get the part of the string of these parenthesized subgroups. Here's how:

| >>> match.group(1) '801'  >>> match.group(2) '35' >>> match.group(1, 2) ('801', '35')  >>> match.groups() ('801', '35') |
| --- |

##### **match.start(), match.end() and match.span()**

The ***start()*** function returns the **index** of the **start** of the matched substring. Similarly, ***end()*** returns the **end index** of the matched substring.

| >>> match.start() 2 >>> match.end() 8 |
| --- |

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

| >>> match.span() (2, 8) |
| --- |

##### **match.re and match.string**

The ***re*** attribute of a matched object returns a regular expression object. Similarly, ***string*** attribute returns the passed string.

| >>> match.re re.compile('(\\d{3}) (\\d{2})')  >>> match.string '39801 356, 2102 1111' |
| --- |

With this, we have covered all commonly used methods defined in the ***re*** module. If you want to learn more, you should visit the [Python 3 ***re*** module documentary](https://docs.python.org/3/library/re.html) on their website during your free time.

Using ***r*** prefix before RegEx

When ***r*** or ***R*** prefix is used before a regular expression, it means **raw string**. For example, **'\n'** is a **new line** whereas ***r*'\n'** means two characters: **a backslash \ followed by n**.

***Backlash*** **\** is used to escape various characters including all metacharacters. However, using ***r*** prefix makes **\** treated as a **normal character** as well.

Example: Raw string using ***r*** prefix

| import re  string = '\n and \r are escape sequences.'  result = re.findall(r'[\n\r]', string)  print(result) |
| --- |

Output

| >>> ['\n', '\r'] |
| --- |

## 7.2 Random

The built-in random module is used to generate random numbers, such as integers and floats

### **choice()**

choice() returns a random element from a sequence (list, tuples, etc)

Syntax:

| choice(sequence) |
| --- |

Example:

| import random  listy = [1,3,6,8] print(random.choice(listy)) |
| --- |

### **random()**

returns a random float between 0 and 1

Syntax:

| random() |
| --- |

Example:

| import random print(random.random()) |
| --- |

### **randint() and randrange()**

Both randint() and randrange() returns an integer between the start and end points (lower and upper limits)

Syntax:

| randint(start,stop) randrange(start,stop) |
| --- |

Example:

| import random print(random.randint(3,6)) print(random.randrange(3,6)) |
| --- |

The difference between randint() and randrange()

For randint(), the integer generated includes the endpoint (upper limit), while randrange() excludes the endpoint

### **uniform()**

uniform() returns a random float between the start and end points

Syntax:

| uniform(start,stop) |
| --- |

Example:

| import random print(random.uniform(4,8)) |
| --- |

### **shuffle()**

shuffle() reorganises the order of elements in a sequence (list, tuples, etc)

Syntax:

| shuffle(sequence) |
| --- |

Example:

| import random listy = [4,5,6,7,8,9] print(random.shuffle(listy)) |
| --- |