## Python

## What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

* web development (server-side),
* software development,
* mathematics,
* system scripting.

### 

### **What can Python do?**

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

### **Why Python?**

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

### **Good to know**

* The most recent major version of Python is **Python 3,** which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
* In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

### **Python Syntax compared to other programming languages**

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

print("Hello, World!")

## Comments

Python has commenting capability for the purpose of in-code documentation.

Comments start with a **#**, and Python will render the rest of the line as a comment:

Comments can be used to explain Python code.

Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

#This is a comment.

print("Hello, World!")

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

### **Example**

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

## Multiline Comments

Python does not really have a syntax for multiline comments.

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

### **Example**

#This is a comment

#written in

#more than just one line

print("Hello, World!")

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

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

### **Example**

"""

This is a comment

written in

more than just one line

"""

print("Hello, World!")

# Python Variables

## Variables

Variables are containers for storing data values.

## Creating Variables

Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

### **Example**

x = 5

y = "John"

print(x)

print(y)

Variables do not need to be declared with any particular type, and can even change type after they have been set.

### **Example**

x = 4       # x is of type int  
x = "Sally" # x is now of type str  
print(x)

## Casting

If you want to specify the data type of a variable, this can be done with casting.

**Example**

x = str(3)

y = int(3)

z = float(3)

print(x)

print(y)

print(z)

## Get the Type

You can get the data type of a variable with the type() function.

### **Example**

x = 5

y = "John"

print(type(x))

print(type(y))

### **Output**

<class 'int'>  
<class 'str'>

## Single or Double Quotes?

String variables can be declared either by using single or double quotes:

### **Example**

x = "John"

print(x)

#double quotes are the same as single quotes:

x = 'John'

print(x)

## Case-Sensitive

Variable names are case-sensitive.

### **Example**

This will create two variables:

a = 4

A = "Sony"

print(a)

print(A)

# Python - Variable Names

## Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total\_volume). Rules for Python variables:

* A variable name must start with a letter or the underscore character
* A variable name cannot start with a number
* A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ )
* Variable names are case-sensitive (age, Age and AGE are three different variables)
* A variable name cannot be any of the [Python keywords](https://www.w3schools.com/python/python_ref_keywords.asp).

### **Example**

myvar = "John"

my\_var = "John"

\_my\_var = "John"

myVar = "John"

MYVAR = "John"

myvar2 = "John"

print(myvar)

print(my\_var)

print(\_my\_var)

print(myVar)

print(MYVAR)

print(myvar2)

### **Example**

Illegal variable names:

2myvar = "John"  
my-var = "John"  
my var = "John"

# Python Variables - Assign Multiple Values

## Many Values to Multiple Variables

Python allows you to assign values to multiple variables in one line:

### **Example**

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

print(x)

print(y)

print(z)

**Note:** Make sure the number of variables matches the number of values, or else you will get an error.

## One Value to Multiple Variables

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

### **Example**

x = y = z = "Orange"

print(x)

print(y)

print(z)

### **output**

Orange  
Orange  
Orange

# Python - Output Variables

## Output Variables

The Python print() function is often used to output variables.

x = "Python is awesome"

print(x)

In the print() function, you output multiple variables, separated by a comma:

### **Example**

x = "Python"  
y = "is"  
z = "awesome"  
print(x, y, z)

### **Output**

Python is awesome

# Python Data Types

## Built-in Data Types

In programming, data type is an important concept.

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

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

|  |  |
| --- | --- |
| Text Type: | str |
| Numeric Types: | int, float, complex |
| Sequence Types: | list, tuple |
| Mapping Type: | dict |
| Set Types: | set, |
| Boolean  Type: numeric –int,float,complex ,string,list,tuple,dictionaries,set,boolean | bool |

## Getting the Data Type

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

### **Example**

Print the data type of the variable x:

x = 5

print(type(x))

### **Output**

<class 'int'>

# Python Numbers

## Python Numbers

There are three numeric types in Python:

* int
* float
* complex

Variables of numeric types are created when you assign a value to them:

### **Example**

x = 1    # int  
y = 2.8  # float  
z =1+9j   # complex

To verify the type of any object in Python, use the type() function:

### **Example**

print(type(x))  
print(type(y))  
print(type(z))

## Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

### **Example**

Integers:

x = 1  
y = 35656222554887711  
z = -3255522  
  
print(type(x))  
print(type(y))  
print(type(z))

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

## Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

### **Example**

Floats:

x = 1.10  
y = 1.0  
z = -35.59  
  
print(type(x))  
print(type(y))  
print(type(z))

## Complex

Complex numbers are written with a "j" as the imaginary part:

### **Example**

Complex:

x = 3+5j  
y = 5j  
z = -5j  
  
print(type(x))  
print(type(y))  
print(type(z))

## Type Conversion

You can convert from one type to another with the int(), float(), and complex() methods:

### **Example**

#convert from int to float:

x = float(1)

#convert from float to int:

y = int(2.8)

#convert from int to complex:

z = complex(1)

print(x)

print(y)

print(z)

print(type(x))

print(type(y))

print(type(z))

# Python Strings

## Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

### **Example**

print("Hello")  
print('Hello')

## Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

### **Example**

a = "Hello"

print(a)

## Multiline Strings

You can assign a multiline string to a variable by using three quotes:

### **Example**

You can use three double quotes:

a = """Lorem ipsum dolor sit amet,

consectetur adipiscing elit,

sed do eiusmod tempor incididunt

ut labore et dolore magna aliqua."""

print(a)

### **output**

Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua.

Or three single quotes:

### **Example**

a = '''Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua.'''  
print(a)

## String Length

To get the length of a string, use the len() function.

### **Example**

The len() function returns the length of a string:

a = "Hello, World!"

print(len(a))

### **output**

13

## Check String

To check if a certain phrase or character is present in a string, we can use the keyword in.

### **Example**

Check if "free" is present in the following text:

txt = "The best things in life are free!"

print("free" in txt)

### **output**

True

Use it in an if statement:

### **Example**

Print only if "free" is present:

txt = "The best things in life are free!"  
if "free" in txt:  
  print("Yes, 'free' is present.")

### **output**

Yes, 'free' is present.

## Check if NOT

To check if a certain phrase or character is NOT present in a string, we can use the keyword not in.

### **Example**

Check if "expensive" is NOT present in the following text:

txt = "The best things in life are free!"  
print("expensive" not in txt)

True

# Python - Slicing Strings

## Slicing

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

### **Example**

Get the characters from position 2 to position 5 (not included):

b = "Hello, World!"

print(b[2:5])

### **output**

llo

**Note:**The first character has index 0.

## Slice From the Start

By leaving out the start index, the range will start at the first character:

### **Example**

Get the characters from the start to position 5 (not included):

b = "Hello, World!"  
print(b[:5])

### **output**

Hello

## Slice To the End

By leaving out the end index, the range will go to the end:

### **Example**

Get the characters from position 2, and all the way to the end:

b = "Hello, World!"  
print(b[2:])

### **output**

llo, World!

## Negative Indexing

Use negative indexes to start the slice from the end of the string:

### **Example**

Get the characters:

From: "o" in "World!" (position -5)

To, but not included: "d" in "World!" (position -2):

b = "Hello, World!"  
print(b[-5:-2])

### **output**

orl

# Python - Modify Strings

Python has a set of built-in methods that you can use on strings.

## Upper Case

### **Example**

The upper() method returns the string in upper case:

a = "Hello, World!"

print(a.upper())

### **output**

HELLO, WORLD!

## Lower Case

### **Example**

The lower() method returns the string in lower case:

a = "Hello, World!"  
print(a.lower())

hello, world!

## Remove Whitespace

Whitespace is the space before and/or after the actual text, and very often you want to remove this space.

### **Example**

The strip() method removes any whitespace from the beginning or the end:

a = " Hello, World! "  
print(a.strip()) # returns "Hello, World!"

Hello, World!

## Replace String

### **Example**

The replace() method replaces a string with another string:

a = "Hello, World!"  
print(a.replace("H", "J"))

Jello, World!

## Split String

The split() method returns a list where the text between the specified separator becomes the list items.

### **Example**

The split() method splits the string into substrings if it finds instances of the separator:

a = "Hello, World!"  
print(a.split(","))  # returns ['Hello', ' World!']

['Hello', ' World!']

# Python - String Concatenation

## String Concatenation

To concatenate, or combine, two strings you can use the + operator.

### **Example**

Merge variable a with variable b into variable c:

a = "Hello"  
b = "World"  
c = a + b  
print(c)

HelloWorld

### **Example**

To add a space between them, add a " ":

a = "Hello"  
b = "World"  
c = a + " " + b  
print(c)

Hello World

# Python - Format - Strings

## String Format

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

### **Example**

age = 36  
txt = "My name is John, I am " + age  
print(txt)

But we can combine strings and numbers by using the format() method!

The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are:

### **Example**

Use the format() method to insert numbers into strings:

age = 36   
txt = "My name is John, and I am {}"  
print(txt.format(age))

My name is John, and I am 36

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

### **Example**

quantity = 3  
itemno = 567  
price = 49.95  
myorder = "I want {} pieces of item {} for {} dollars."  
print(myorder.format(quantity, itemno, price))

I want 3 pieces of item 567 for 49.95 dollars.

ou can use index numbers {0} to be sure the arguments are placed in the correct placeholders:

### **Example**

quantity = 3  
itemno = 567  
price = 49.95  
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."  
print(myorder.format(quantity, itemno, price))

I want to pay 49.95 dollars for 3 pieces of item 567

# Python - Escape Characters

## Escape Character

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

To fix this problem, use the escape character \":

### **Example**

The escape character allows you to use double quotes when you normally would not be allowed:

txt = "We are the so-called \"Vikings\" from the north."

We are the so-called "Vikings" from the north.

Escape Characters

Other escape characters used in Python:

|  |  |  |
| --- | --- | --- |
| **Code** | **Result** | **Try it** |
| \' | Single Quote | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_string_escape2) |
| \\ | Backslash | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_string_backslash) |
| \n | New Line | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_string_newline) |
| \t | Tab | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_string_t) |
| \b | Backspace | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_b) |

# Python Booleans

Booleans represent one of two values: True or False.

## Boolean Values

In programming you often need to know if an expression is True or False.

You can evaluate any expression in Python, and get one of two answers, True or False.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

### **Example**

print(10 > 9)

print(10 == 9)

print(10 < 9)

When you run a condition in an if statement, Python returns True or False:

### **Example**

Print a message based on whether the condition is True or False:

a = 200

b = 33

if b > a:

print("b is greater than a")

else:

print("b is not greater than a")

# Python Operators

## Python Operators

Operators are used to perform operations on variables and values.

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

### **Example**

print(10 + 5)

Python divides the operators in the following groups:

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

## Python Arithmetic Operators

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Try it** |
| + | Addition | x + y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_add) |
| - | Subtraction | x - y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_sub) |
| \* | Multiplication | x \* y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_mult) |
| / | Division | x / y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_div) |
| % | Modulus | x % y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_mod) |
| \*\* | Exponentiation(power) | x \*\* y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_exp) |
| // | Floor division | x // y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_floordiv) |

x = 5

y = 3

print(x + y)

x = 12

y = 3

print(x / y)

x = 5

y = 2

print(x % y)

x = 2

y = 4

print(x \*\* y) #same as 2\*2\*2\*2

x = 15

y = 2

print(x // y)

#the floor division // rounds the result down to the nearest whole number

## Python Assignment Operators

Assignment operators are used to assign values to variables:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Example** | | **Same As** |  |
| = | x = 5 | | x = 5 |  |
| += | x += 3 | | x = x + 3 |  |
| -= | x -= 3 | | x = x - 3 |
| \*= | x \*= 3 | | x = x \* 3 |
| /= | x /= 3 | | x = x / 3 |
| %= | x %= 3 | | x = x % 3 |
| //= | x //= 3 | | x = x // 3 |
| \*\*= | x \*\*= 3 | | x = x \*\* 3 |
|  | |
|  | |

x = 5

x -= 3

print(x)

## Python Comparison Operators

Comparison operators are used to compare two values:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Try it** |
| == | Equal | x == y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare1) |
| != | Not equal | x != y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare2) |
| > | Greater than | x > y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare4) |
| < | Less than | x < y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare5) |
| >= | Greater than or equal to | x >= y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare6) |
| <= | Less than or equal to | x <= y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_compare7) |

x = 5

y = 3

print(x != y)

# returns False because 5 is not equal to 3

x = 5

y = 3

print(x != y)

# returns True because 5 is not equal to 3

x = 5

y = 3

print(x >= y)

# returns True because five is greater, or equal, to 3

## Python Logical Operators

Logical operators are used to combine conditional statements:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Try it** |
| and | Returns True if both statements are true | x < 5 and  x < 10 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_logical1) |
| or | Returns True if one of the statements is true | x < 5 or x < 4 | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_logical2) |
| not | Reverse the result, returns False if the result is true | not(x < 5 and x < 10) |  |

x = 5

print(x > 3 and x < 10)

# returns True because 5 is greater than 3 AND 5 is less than 10

x = 5

print(x > 3 or x < 4)

# returns True because one of the conditions are true (5 is greater than 3, but 5 is not less than 4)

x = 5

print(not(x > 3 and x < 10))

# returns False because not is used to reverse the result

## Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Try it** |
| is | Returns True if both variables are the same object | x is y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_identity1) |
| is not | Returns True if both variables are not the same object | x is not y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_identity2) |

IS OPERATOR

x = ["apple", "banana"]

y = ["apple", "banana"]

z = x

print(x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y, even if they have the same content

print(x == y)

# to demonstrate the difference betweeen "is" and "==": this comparison returns True because x is equal to y

IS NOT OPERATOR

x = ["apple", "banana"]

y = ["apple", "banana"]

z = x

print(x is not z)

# returns False because z is the same object as x

print(x is not y)

# returns True because x is not the same object as y, even if they have the same content

print(x != y)

# to demonstrate the difference betweeen "is not" and "!=": this comparison returns False because x is equal to y

## Python Membership Operators

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Try it** |
| in | Returns True if a sequence with the specified value is present in the object | x in y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_membership1) |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y |  |

IN

x = ["apple", "banana"]

print("banana" in x)

# returns True because a sequence with the value "banana" is in the list

NOT IN

x = ["apple", "banana"]

print("pineapple" not in x)

# returns True because a sequence with the value "pineapple" is not in the list

## Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** | **Try it** |
| & | AND | Sets each bit to 1 if both bits are 1 | x & y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_and) |
| | | OR | Sets each bit to 1 if one of two bits is 1 | x | y | [Try it »](https://www.w3schools.com/python/trypython.asp?filename=demo_oper_or) |

## Python Collections (Arrays)

There are four collection data types in the Python programming language:

* **List** is a collection which is ordered and changeable. Allows duplicate members.
* [**Tuple**](https://www.w3schools.com/python/python_tuples.asp) is a collection which is ordered and unchangeable. Allows duplicate members.
* [**Set**](https://www.w3schools.com/python/python_sets.asp) is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members.
* [**Dictionary**](https://www.w3schools.com/python/python_dictionaries.asp) is a collection which is ordered\*\* and changeable. No duplicate members.

\*Set items are unchangeable, but you can remove and/or add items whenever you like.

# Python Lists

mylist = ["apple", "banana", "cherry"]

## List

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](https://www.w3schools.com/python/python_tuples.asp), [Set](https://www.w3schools.com/python/python_sets.asp), and [Dictionary](https://www.w3schools.com/python/python_dictionaries.asp), all with different qualities and usage.

Lists are created using square brackets:

### **Example**

Create a List:

thislist = ["apple", "banana", "cherry"]

print(thislist)

## List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

## Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

**Note:** There are some [list methods](https://www.w3schools.com/python/python_lists_methods.asp) that will change the order, but in general: the order of the items will not change.

## Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

## Allow Duplicates

Since lists are indexed, lists can have items with the same value:

### **Example**

Lists allow duplicate values:

thislist = ["apple", "banana", "cherry", "apple", "cherry"]

print(thislist)

## List Length

To determine how many items a list has, use the len() function:

### **Example**

Print the number of items in the list:

thislist = ["apple", "banana", "cherry"]

print(len(thislist))

## List Items - Data Types

List items can be of any data type:

### **Example**

String, int and boolean data types:

list1 = ["apple", "banana", "cherry"]

list2 = [1, 5, 7, 9, 3]

list3 = [True, False, False]

print(list1)

print(list2)

print(list3)

### **Example**

A list with strings, integers and boolean values:

list1 = ["abc", 34, True, 40, "male"]

print(list1)

## type()

From Python's perspective, lists are defined as objects with the data type 'list':

<class 'list'>

### **Example**

What is the data type of a list?

mylist = ["apple", "banana", "cherry"]

print(type(mylist))

It is also possible to use the list() constructor when creating a new list.

### **Example**

Using the list() constructor to make a List:

thislist = list(("apple", "banana", "cherry"))

print(thislist)

# Python - Access List Items

## Access Items

List items are indexed and you can access them by referring to the index number:

### **Example**

Print the second item of the list:

thislist = ["apple", "banana", "cherry"]  
print(thislist[1])

banana

**Note:**The first item has index 0.

### **Negative Indexing**

Negative indexing means start from the end

-1 refers to the last item, -2 refers to the second last item etc.

### **Example**

Print the last item of the list:

thislist = ["apple", "banana", "cherry"]

print(thislist[-1])

cherry

### **Range of Indexes**

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

### **Example**

Return the third, fourth, and fifth item:

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(thislist[2:5])

#This will return the items from position 2 to 5.

#Remember that the first item is position 0,

#and note that the item in position 5 is NOT included

['cherry', 'orange', 'kiwi']

**Note:** The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

### **Example**

This example returns the items from the beginning to, but NOT including, "kiwi":

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(thislist[:4])

#This will return the items from index 0 to index 4.

#Remember that index 0 is the first item, and index 4 is the fifth item

#Remember that the item in index 4 is NOT included

By leaving out the end value, the range will go on to the end of the list:

### **Example**

This example returns the items from "cherry" to the end:

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(thislist[2:])

#This will return the items from index 2 to the end.

#Remember that index 0 is the first item, and index 2 is the third

['cherry', 'orange', 'kiwi', 'melon', 'mango']

### **Range of Negative Indexes**

Specify negative indexes if you want to start the search from the end of the list:

### **Example**

This example returns the items from "orange" (-4) to, but NOT including "mango" (-1):

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(thislist[-4:-1])

#Negative indexing means starting from the end of the list.

#This example returns the items from index -4 (included) to index -1 (excluded)

#Remember that the last item has the index -1,

['orange', 'kiwi', 'melon']

## Check if Item Exists

To determine if a specified item is present in a list use the in keyword:

### **Example**

Check if "apple" is present in the list:

thislist = ["apple", "banana", "cherry"]

if "apple" in thislist:

print("Yes, 'apple' is in the fruits list")

Yes, 'apple' is in the fruits list

# Python - Change List Items

## Change Item Value

To change the value of a specific item, refer to the index number:

### **Example**

Change the second item:

thislist = ["apple", "banana", "cherry"]

thislist[1] = "blackcurrent"

print(thislist)

['apple', 'blackcurrant', 'cherry']

## Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

### **Example**

Change the values "banana" and "cherry" with the values "blackcurrent" and "watermelon":

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]

thislist[1:3] = ["blackcurrent", "watermelon"]

print(thislist)

['apple', 'blackcurrent', 'watermelon', 'orange', 'kiwi', 'mango']

If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

### **Example**

Change the second value by replacing it with two new values:

thislist = ["apple", "banana", "cherry"]

thislist[1:2] = ["blackcurrant", "watermelon"]

print(thislist)

['apple', 'blackcurrant', 'watermelon', 'cherry']

**Note:** The length of the list will change when the number of items inserted does not match the number of items replaced.

If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

### **Example**

Change the second and third value by replacing it with one value:

thislist = ["apple", "banana", "cherry"]

thislist[1:3] = ["watermelon"]

print(thislist)

['apple', 'watermelon']

## Insert Items

To insert a new list item, without replacing any of the existing values, we can use the insert() method.

The insert() method inserts an item at the specified index:

### **Example**

Insert "watermelon" as the third item:

thislist = ["apple", "banana", "cherry"]

thislist.insert(2, "watermelon")

print(thislist)

# Python - Add List Items

## Append Items

To add an item to the end of the list, use the append() method:

### **Example**

Using the append() method to append an item:

thislist = ["apple", "banana", "cherry"]

thislist.append("orange")

print(thislist)

['apple', 'banana', 'cherry', 'orange']

## Insert Items

To insert a list item at a specified index, use the insert() method.

The insert() method inserts an item at the specified index:

### **Example**

Insert an item as the second position:

thislist = ["apple", "banana", "cherry"]

thislist.insert(1, "orange")

print(thislist)

['apple', 'orange', 'banana', 'cherry']

## Extend List

To append elements from another list to the current list, use the extend() method.

### **Example**

Add the elements of tropical to thislist:

thislist = ["apple", "banana", "cherry"]

tropical = ["mango", "pineapple", "papaya"]

thislist.extend(tropical)

print(thislist)

['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']

The elements will be added to the end of the list.

## Add Any Iterable

The extend() method does not have to append lists, you can add any iterable object (tuples, sets, dictionaries etc.).

### **Example**

Add elements of a tuple to a list:

thislist = ["apple", "banana", "cherry"]

thistuple = ("kiwi", "orange")

thislist.extend(thistuple)

print(thislist)

['apple', 'banana', 'cherry', 'kiwi', 'orange']

# Python - Remove List Items

## Remove Specified Item

The remove() method removes the specified item.

### **Example**

Remove "banana":

thislist = ["apple", "banana", "cherry"]

thislist.remove("banana")

print(thislist)

['apple', 'cherry']

If there are more than one item with the specified value, the remove() method removes the first occurance:

### **Example**

Remove the first occurance of "banana":

thislist = ["apple", "banana", "cherry", "banana", "kiwi"]

thislist.remove("banana")

print(thislist)

['apple', 'cherry', 'banana', 'kiwi']

## Remove Specified Index

The pop() method removes the specified index.

### **Example**

Remove the second item:

thislist = ["apple", "banana", "cherry"]

thislist.pop(1)

print(thislist)

['apple', 'cherry']

If you do not specify the index, the pop() method removes the last item.

### **Example**

Remove the last item:

thislist = ["apple", "banana", "cherry"]

thislist.pop()

print(thislist)

['apple', 'banana']

The del keyword also removes the specified index:

### **Example**

Remove the first item:

thislist = ["apple", "banana", "cherry"]

del thislist[0]

print(thislist)

['banana', 'cherry']

The del keyword can also delete the list completely.

### **Example**

Delete the entire list:

thislist = ["apple", "banana", "cherry"]

del thislist

print(thislist) #this will cause an error because you have succsesfully deleted "thislist".

Traceback (most recent call last):  
  File "demo\_list\_del2.py", line 3, in <module>  
    print(thislist) #this will cause an error because you have succsesfully deleted "thislist".  
NameError: name 'thislist' is not defined

## Clear the List

The clear() method empties the list.

The list still remains, but it has no content.

### **Example**

Clear the list content:

thislist = ["apple", "banana", "cherry"]

thislist.clear()

print(thislist)

[]

# Python - Sort Lists

## Sort List Alphanumerically

List objects have a sort() method that will sort the list alphanumerically, ascending, by default:

### **Example**

Sort the list alphabetically:

thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]

thislist.sort()

print(thislist)

['banana', 'kiwi', 'mango', 'orange', 'pineapple']

### **Example**

Sort the list numerically:

thislist = [100, 50, 65, 82, 23]

thislist.sort()

print(thislist)

[23, 50, 65, 82, 100]

## Sort Descending

To sort descending, use the keyword argument reverse = True:

### **Example**

Sort the list descending:

thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]

thislist.sort(reverse = True)

print(thislist)

['pineapple', 'orange', 'mango', 'kiwi', 'banana']

### **Example**

Sort the list descending:

thislist = [100, 50, 65, 82, 23]

thislist.sort(reverse = True)

print(thislist)

[100, 82, 65, 50, 23]

# Python - Copy Lists

## Copy a List

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy().

### **Example**

Make a copy of a list with the copy() method:

thislist = ["apple", "banana", "cherry"]

mylist = thislist.copy()

print(mylist)

['apple', 'banana', 'cherry']

Another way to make a copy is to use the built-in method list().

### **Example**

Make a copy of a list with the list() method:

thislist = ["apple", "banana", "cherry"]

mylist = list(thislist)

print(mylist)

['apple', 'banana', 'cherry']

# Python - Join Lists

## Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

### **Example**

Join two list:

list1 = ["a", "b", "c"]

list2 = [1, 2, 3]

list3 = list1 + list2

print(list3)

['a', 'b', 'c', 1, 2, 3]

Another way to join two lists is by appending all the items from list2 into list1, one by one:

### **Example**

Append list2 into list1:

list1 = ["a", "b" , "c"]

list2 = [1, 2, 3]

for x in list2:

list1.append(x)

print(list1)

['a', 'b', 'c', 1, 2, 3]

Or you can use the extend() method, where the purpose is to add elements from one list to another list:

### **Example**

Use the extend() method to add list2 at the end of list1:

list1 = ["a", "b" , "c"]

list2 = [1, 2, 3]

list1.extend(list2)

print(list1)

['a', 'b', 'c', 1, 2, 3]

# Python List count() Method

### **Example**

Return the number of times the value "cherry" appears in the fruits list:

fruits = ['apple', 'banana', 'cherry']  
  
x = fruits.count("cherry")

print(x)

1

# Python Tuples

mytuple = ("apple", "banana", "cherry")

Tuple

Tuples are used to store multiple items in a single variable.

A tuple is a collection which is ordered and **unchangeable**.

Tuples are written with round brackets.

### **Example**

Create a Tuple:

thistuple = ("apple", "banana", "cherry")

print(thistuple)

('apple', 'banana', 'cherry')

## Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

## Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

## Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

## Allow Duplicates

Since tuples are indexed, they can have items with the same value:

### **Example**

Tuples allow duplicate values:

thistuple = ("apple", "banana", "cherry", "apple", "cherry")

print(thistuple)

('apple', 'banana', 'cherry', 'apple', 'cherry')

## Tuple Length

To determine how many items a tuple has, use the len() function:

### **Example**

Print the number of items in the tuple:

thistuple = tuple(("apple", "banana", "cherry"))

print(len(thistuple))

3

## Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

### **Example**

One item tuple, remember the comma:

thistuple = ("apple",)

print(type(thistuple))

#NOT a tuple

thistuple = ("apple")

print(type(thistuple))

## Tuple Items - Data Types

Tuple items can be of any data type:

### **Example**

String, int and boolean data types:

tuple1 = ("apple", "banana", "cherry")

tuple2 = (1, 5, 7, 9, 3)

tuple3 = (True, False, False)

print(tuple1)

print(tuple2)

print(tuple3)

('apple', 'banana', 'cherry')

(1, 5, 7, 9, 3)

(True, False, False)

A tuple can contain different data types:

### **Example**

A tuple with strings, integers and boolean values:

tuple1 = ("abc", 34, True, 40, "male")

print(tuple1)

('abc', 34, True, 40, 'male')

## The tuple() Constructor

It is also possible to use the tuple() constructor to make a tuple.

### **Example**

Using the tuple() method to make a tuple:

thistuple = tuple(("apple", "banana", "cherry"))

print(thistuple)

('apple', 'banana', 'cherry')

# Python - Access Tuple Items

## Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

### **Example**

Print the second item in the tuple:

thistuple = ("apple", "banana", "cherry")

print(thistuple[1])

banana

## Negative Indexing

Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

### **Example**

Print the last item of the tuple:

thistuple = ("apple", "banana", "cherry")

print(thistuple[-1])

## Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

### **Example**

Return the third, fourth, and fifth item:

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

print(thistuple[2:5])

#This will return the items from position 2 to 5.

#Remember that the first item is position 0,

#and note that the item in position 5 is NOT included

('cherry', 'orange', 'kiwi')

### **Example**

This example returns the items from the beginning to, but NOT included, "kiwi":

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

print(thistuple[:4])

('apple', 'banana', 'cherry', 'orange')

By leaving out the end value, the range will go on to the end of the tuple:

### **Example**

This example returns the items from "cherry" and to the end:

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

print(thistuple[2:])

('cherry', 'orange', 'kiwi', 'melon', 'mango')

## Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

### **Example**

This example returns the items from index -4 (included) to index -1 (excluded)

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

print(thistuple[-4:-1])

('orange', 'kiwi', 'melon')

# Python - Update Tuples

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

But there are some workarounds.

## Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

### **Example**

Convert the tuple into a list to be able to change it:

x = ("apple", "banana", "cherry")

y = list(x)

y[1] = "kiwi"

x = tuple(y)

print(x)

("apple", "kiwi", "cherry")

## Add Items

Since tuples are immutable, they do not have a built-in append() method, but there are other ways to add items to a tuple.

1. **Convert into a list**: Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

### **Example**

Convert the tuple into a list, add "orange", and convert it back into a tuple:

thistuple = ("apple", "banana", "cherry")

y = list(thistuple)

y.append("orange")

thistuple = tuple(y)

print(thistuple)

('apple', 'banana', 'cherry', 'orange')

## Remove Items

**Note:** You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

### **Example**

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

thistuple = ("apple", "banana", "cherry")

y = list(thistuple)

y.remove("apple")

thistuple = tuple(y)

print(thistuple)

('banana', 'cherry')

# Python Sets

myset = {"apple", "banana", "cherry"}

## Set

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](https://www.w3schools.com/python/python_lists.asp), [Tuple](https://www.w3schools.com/python/python_tuples.asp), and [Dictionary](https://www.w3schools.com/python/python_dictionaries.asp), all with different qualities and usage.

A set is a collection which is unordered, unchangeable\*, and unindexed.

**\* Note:** Set items are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

### **Example**

Create a Set:

thisset = {"apple", "banana", "cherry"}

print(thisset)

# Note: the set list is unordered, meaning: the items will appear in a random order.

# Refresh this page to see the change in the result.

{'banana', 'apple', 'cherry'}

**Note:** Sets are unordered, so you cannot be sure in which order the items will appear.

## Set Items

Set items are unordered, unchangeable, and do not allow duplicate values.

## Unordered

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

## Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

Once a set is created, you cannot change its items, but you can remove items and add new items.

## Duplicates Not Allowed

Sets cannot have two items with the same value.

### **Example**

Duplicate values will be ignored:

thisset = {"apple", "banana", "cherry", "apple"}

print(thisset)

{'banana', 'cherry', 'apple'}

**Note:** The values True and 1 are considered the same value in sets, and are treated as duplicates:

### **Example**

True and 1 is considered the same value:

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

print(thisset)

{True, 2, 'banana', 'cherry', 'apple'}

**Note:** The values False and 0 are considered the same value in sets, and are treated as duplicates:

### **Example**

False and 0 is considered the same value:

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

print(thisset)

{False, True, 'cherry', 'apple', 'banana'}

## Get the Length of a Set

To determine how many items a set has, use the len() function.

### **Example**

Get the number of items in a set:

thisset = {"apple", "banana", "cherry"}

print(len(thisset))

3

## Set Items - Data Types

Set items can be of any data type:

### **Example**

String, int and boolean data types:

set1 = {"apple", "banana", "cherry"}

set2 = {1, 5, 7, 9, 3}

set3 = {True, False, False}

print(set1)

print(set2)

print(set3)

{'cherry', 'apple', 'banana'}

{1, 3, 5, 7, 9}

{False, True}

A set can contain different data types:

### **Example**

A set with strings, integers and boolean values:

set1 = {"abc", 34, True, 40, "male"}

print(set1)

{True, 34, 40, 'male', 'abc'}

## type()

From Python's perspective, sets are defined as objects with the data type 'set':

<class 'set'>

### **Example**

What is the data type of a set?

myset = {"apple", "banana", "cherry"}  
print(type(myset))

<class 'set'>

## The set() Constructor

It is also possible to use the set() constructor to make a set.

### **Example**

Using the set() constructor to make a set:

thisset = set(("apple", "banana", "cherry"))

print(thisset)

# Note: the set list is unordered, so the result will display the items in a random order.

{'banana', 'cherry', 'apple'}

## Access Items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

### **Example**

Loop through the set, and print the values:

thisset = {"apple", "banana", "cherry"}

for x in thisset:

print(x)

banana  
cherry  
apple

### **Example**

Check if "banana" is present in the set:

thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)

True

Change Items

Once a set is created, you cannot change its items, but you can add new items.

# Python - Add Set Items

## Add Items

Once a set is created, you cannot change its items, but you can add new items.

To add one item to a set use the add() method.

### **Example**

Add an item to a set, using the add() method:

thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)

{'apple', 'orange', 'banana', 'cherry'}

## Add Sets

To add items from another set into the current set, use the update() method.

### **Example**

Add elements from tropical into thisset:

thisset = {"apple", "banana", "cherry"}

tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)

{'apple', 'mango', 'cherry', 'pineapple', 'banana', 'papaya'}

## Add Any Iterable

The object in the update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

### **Example**

Add elements of a list to at set:

thisset = {"apple", "banana", "cherry"}

mylist = ["kiwi", "orange"]

thisset.update(mylist)

print(thisset)

{'banana', 'cherry', 'apple', 'orange', 'kiwi'}

# Python - Remove Set Items

## Remove Item

To remove an item in a set, use the remove(), or the discard() method.

### **Example**

Remove "banana" by using the remove() method:

thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)

{'apple', 'cherry'}

**Note:** If the item to remove does not exist, remove() will raise an error.

### **Example**

Remove "banana" by using the discard() method:

thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)

{'apple', 'cherry'}

**Note:** If the item to remove does not exist, discard() will **NOT** raise an error.

You can also use the pop() method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed.

The return value of the pop() method is the removed item.

### **Example**

Remove a random item by using the pop() method:

thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x) #removed item

print(thisset) #the set after removal

banana  
{'apple', 'cherry'}

**Note:** Sets are unordered, so when using the pop() method, you do not know which item that gets removed.

### **Example**

The clear() method empties the set:

thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)

set()

### **Example**

The del keyword will delete the set completely:

thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset) #this will raise an error because the set no longer exists

Traceback (most recent call last):  
  File "demo\_set\_del.py", line 5, in <module>  
    print(thisset) #this will raise an error because the set no longer exists  
NameError: name 'thisset' is not defined

# Python - Join Sets

## Join Two Sets

There are several ways to join two or more sets in Python.

You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another:

### **Example**

The union() method returns a new set with all items from both sets:

set1 = {"a", "b" , "c"}

set2 = {1, 2, 3}

set3 = set1.union(set2)

print(set3)

{'c', 'a', 3, 1, 2, 'b'}

### **Example**

The update() method inserts the items in set2 into set1:

set1 = {"a", "b" , "c"}

set2 = {1, 2, 3}

set1.update(set2)

print(set1)

{1, 'a', 'c', 2, 3, 'b'}

**Note:** Both union() and update() will exclude any duplicate items.

## Keep ONLY the Duplicates

The intersection\_update() method will keep only the items that are present in both sets.

### **Example**

Keep the items that exist in both set x, and set y:

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

x.intersection\_update(y)

print(x)

{'apple'}

The intersection() method will return a new set, that only contains the items that are present in both sets.

### **Example**

Return a set that contains the items that exist in both set x, and set y:

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

z = x.intersection(y)

print(z)

{'apple'}

## Keep All, But NOT the Duplicates

The symmetric\_difference\_update() method will keep only the elements that are NOT present in both sets.

### **Example**

Keep the items that are not present in both sets:

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

x.symmetric\_difference\_update(y)

print(x)

{'google', 'banana', 'microsoft', 'cherry'}

The symmetric\_difference() method will return a new set, that contains only the elements that are NOT present in both sets.

### **Example**

Return a set that contains all items from both sets, except items that are present in both:

x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}

z = x.symmetric\_difference(y)

print(z)

{'google', 'banana', 'microsoft', 'cherry'}

**Note:** The values True and 1 are considered the same value in sets, and are treated as duplicates:

### **Example**

True and 1 is considered the same value:

x = {"apple", "banana", "cherry", True}

y = {"google", 1, "apple", 2}

z = x.symmetric\_difference(y)

print(z)

{2, 'google', 'cherry', 'banana'}

# Python Dictionaries

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

## Dictionary

Dictionaries are used to store data values in key:value pairs.

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

Dictionaries are written with curly brackets, and have keys and values:

### **Example**

Create and print a dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

print(thisdict)

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

## Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

### **Example**

Print the "brand" value of the dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964}

print(thisdict["brand"])

Ford

Ordered or Unordered?

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

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

### **Example**

Duplicate values will overwrite existing values:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964,

"year": 2020

}

print(thisdict)

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

## Dictionary Length

To determine how many items a dictionary has, use the len() function:

### **Example**

Print the number of items in the dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964,

"year": 2020

}

print(len(thisdict))

3

## Dictionary Items - Data Types

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

### **Example**

String, int, boolean, and list data types:

thisdict = {

"brand": "Ford",

"electric": False,

"year": 1964,

"colors": ["red", "white", "blue"]

}

print(thisdict)

{'brand': 'Ford', 'electric': False, 'year': 1964, 'colors': ['red', 'white', 'blue']}

## The dict() Constructor

It is also possible to use the dict() constructor to make a dictionary.

### **Example**

Using the dict() method to make a dictionary:

thisdict = dict(name = "John", age = 36, country = "Norway")

print(thisdict)

{'name': 'John', 'age': 36, 'country': 'Norway'}

# Python - Access Dictionary Items

## Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

### **Example**

Get the value of the "model" key:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = thisdict["model"]

print(x)

Mustang

There is also a method called get() that will give you the same result:

### **Example**

Get the value of the "model" key:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = thisdict.get("model")

print(x)

Mustang

## Get Keys

The keys() method will return a list of all the keys in the dictionary.

### **Example**

Get a list of the keys:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = thisdict.keys()

print(x)

dict\_keys(['brand', 'model', 'year'])

The list of the keys is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

### **Example**

Add a new item to the original dictionary, and see that the keys list gets updated as well:

car = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change

dict\_keys(['brand', 'model', 'year'])  
dict\_keys(['brand', 'model', 'year', 'color'])

## Get Values

The values() method will return a list of all the values in the dictionary.

### **Example**

Get a list of the values:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = thisdict.values()

print(x)

The list of the values is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

### **Example**

Make a change in the original dictionary, and see that the values list gets updated as well:

car = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change

dict\_values(['Ford', 'Mustang', 1964])  
dict\_values(['Ford', 'Mustang', 2020])

### **Example**

Add a new item to the original dictionary, and see that the values list gets updated as well:

car = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = car.values()

print(x) #before the change

car["color"] = "red"

print(x) #after the change

dict\_values(['Ford', 'Mustang', 1964])  
dict\_values(['Ford', 'Mustang', 1964, 'red'])

## Get Items

The items() method will return each item in a dictionary, as tuples in a list.

### **Example**

Get a list of the key:value pairs

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = thisdict.items()

print(x)

dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

The returned list is a view of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

### **Example**

Make a change in the original dictionary, and see that the items list gets updated as well:

car = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = car.items()

print(x) #before the change

car["year"] = 2020

print(x) #after the change

dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])  
dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2020)])

### **Example**

Add a new item to the original dictionary, and see that the items list gets updated as well:

car = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

x = car.items()

print(x) #before the change

car["color"] = "red"

print(x) #after the change

dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])  
dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964), ('color', 'red')])

## Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword:

### **Example**

Check if "model" is present in the dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

if "model" in thisdict:

print("Yes, 'model' is one of the keys in the thisdict dictionary")

Yes, 'model' is one of the keys in the thisdict dictionary

# Python - Change Dictionary Items

## Change Values

You can change the value of a specific item by referring to its key name:

### **Example**

Change the "year" to 2018:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

thisdict["year"] = 2018

print(thisdict)

{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}

## Update Dictionary

The update() method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

### **Example**

Update the "year" of the car by using the update() method:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

thisdict.update({"year": 2020})

print(thisdict)

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

# Python - Add Dictionary Items

## Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

### **Example**

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

thisdict["color"] = "red"

print(thisdict)

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}

## Update Dictionary

The update() method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

### **Example**

Add a color item to the dictionary by using the update() method:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

thisdict.update({"color": "red"})

print(thisdict)

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}

# Python - Remove Dictionary Items

## Removing Items

There are several methods to remove items from a dictionary:

### **Example**

The pop() method removes the item with the specified key name:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

thisdict.pop("model")

print(thisdict)

{'brand': 'Ford', 'year': 1964}

### **Example**

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

thisdict.popitem()

print(thisdict)

{'brand': 'Ford', 'model': 'Mustang'}

### **Example**

The del keyword removes the item with the specified key name:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

del thisdict["model"]

print(thisdict)

{'brand': 'Ford', 'year': 1964}

### **Example**

The del keyword can also delete the dictionary completely:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
del thisdict  
print(thisdict) #this will cause an error because "thisdict" no longer exists.

### **Example**

The clear() method empties the dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

thisdict.clear()

print(thisdict)

{}

# Python - Loop Dictionaries

## Loop Through a Dictionary

You can loop through a dictionary by using a for loop.

When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

### **Example**

Print all key names in the dictionary, one by one:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

for x in thisdict:

print(x)

brand  
model  
year

### **Example**

Print all values in the dictionary, one by one:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

for x in thisdict:

print(thisdict[x])

Ford  
Mustang  
1964

### **Example**

You can also use the values() method to return values of a dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

for x in thisdict.values():

print(x)

Ford  
Mustang  
1964

### **Example**

You can use the keys() method to return the keys of a dictionary:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

for x in thisdict.keys():

print(x)

brand  
model  
year

### **Example**

Loop through both keys and values, by using the items() method:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

for x, y in thisdict.items():

print(x, y)

brand Ford  
model Mustang  
year 1964

# Python - Copy Dictionaries

## Copy a Dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

### **Example**

Make a copy of a dictionary with the copy() method:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

mydict = thisdict.copy()

print(mydict)

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

Another way to make a copy is to use the built-in function dict().

### **Example**

Make a copy of a dictionary with the dict() function:

thisdict = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

mydict = dict(thisdict)

print(mydict)

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

# Python - Nested Dictionaries

## Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

### **Example**

Create a dictionary that contain three dictionaries:

myfamily = {

"child1" : {

"name" : "Emil",

"year" : 2004

},

"child2" : {

"name" : "Tobias",

"year" : 2007

},

"child3" : {

"name" : "Linus",

"year" : 2011

}

}

print(myfamily)

{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}

## Access Items in Nested Dictionaries

To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary:

### **Example**

Print the name of child 2:

{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}

Or, if you want to add three dictionaries into a new dictionary:

### **Example**

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

child1 = {

"name" : "Emil",

"year" : 2004

}

child2 = {

"name" : "Tobias",

"year" : 2007

}

child3 = {

"name" : "Linus",

"year" : 2011

}

myfamily = {

"child1" : child1,

"child2" : child2,

"child3" : child3

}

print(myfamily)

{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}

myfamily = {

"child1" : {

"name" : "Emil",

"year" : 2004

},

"child2" : {

"name" : "Tobias",

"year" : 2007

},

"child3" : {

"name" : "Linus",

"year" : 2011

}

}

print(myfamily["child2"]["name"])

Tobias

***Encapsulation***

**Encapsulation** is one of the fundamental concepts in object-oriented programming (OOP), and it is widely used in Python. It refers to the bundling of data (variables) and methods (functions) that operate on the data into a single unit, or class. It also involves restricting access to certain details or implementation of the class from outside code. This is typically done by making variables or methods private (or protected) and providing public methods to access or modify those variables.

### Benefits of Encapsulation:

1. **Data Hiding**: Protects the object's internal state from being directly accessed or modified by external code.
2. **Control**: Provides a controlled interface for interacting with the object's data, such as using getter and setter methods to validate inputs.
3. **Modularity**: Allows for cleaner, more maintainable code by isolating the class implementation.

### Key Concepts in Python Encapsulation:

1. **Private Members**: Members (variables and methods) that are intended to be private to the class are marked by prefixing them with double underscores (\_\_). This "name mangling" helps avoid accidental modification from outside the class.
2. **Public Members**: These are the default members and can be accessed directly from outside the class.
3. **Getter and Setter Methods**: These are methods used to access and modify private members indirectly. These methods control how the data is accessed or modified.

### Example of Encapsulation in Python:

class Person:

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

self. \_\_name = name # Private variable

self. \_\_age = age # Private variable

# Getter method for name

def get\_name(self):

return self.\_\_name

# Setter method for name

def set\_name(self, name):

self.\_\_name = name

# Getter method for age

def get\_age(self):

return self.\_\_age

# Setter method for age

def set\_age(self, age):

if age >= 0: # Validation check

self.\_\_age = age

else:

print("Age cannot be negative.")

# Creating an object of Person class

person = Person("Alice", 30)

# Accessing private variables through getter methods

print(person.get\_name()) # Output: Alice

print(person.get\_age()) # Output: 30

# Trying to modify the private variables using setter methods

person.set\_name("Bob")

person.set\_age(35)

print(person.get\_name()) # Output: Bob

print(person.get\_age()) # Output: 35

# Trying to set an invalid age

person.set\_age(-5) # Output: Age cannot be negative.

### Important Notes:

1. **Name Mangling**: In Python, the double underscores before an attribute name (like \_\_name) causes Python to "mangle" the name, meaning it changes it internally to something like \_Person\_\_name. This prevents direct access to the attribute from outside the class, but it doesn't make it completely private.
2. **Single Underscore \_**: This is used to indicate that an attribute or method is intended for internal use (protected), but it is still accessible from outside the class. It is just a convention and does not enforce any restrictions.

python

Copy code

class MyClass:

def \_\_init\_\_(self):

self.\_protected\_var = 10 # Conventionally protected, but still accessible

### Summary:

Encapsulation in Python helps control access to an object's internal state by using private and public members. You can use getter and setter methods to provide controlled access and modification of private attributes, ensuring that they are validated or processed before being changed. This improves security, flexibility, and maintainability in your code.

***Polymorphism***

**Polymorphism** is another fundamental concept in object-oriented programming (OOP) that allows different classes to be treated as instances of the same class through a shared interface. The key idea behind polymorphism is that **the same method or operation can behave differently based on the object it is applied to**.

In Python, polymorphism is commonly achieved through **method overriding** and **duck typing**.

### Types of Polymorphism in Python:

1. **Method Overriding (Compile-time Polymorphism or Early Binding)**:
   * This occurs when a subclass provides its own implementation of a method that is already defined in its superclass.
   * The subclass method overrides the parent class method, and the version of the method that gets executed depends on the type of the object (i.e., subclass or superclass).
2. **Duck Typing (Runtime Polymorphism or Late Binding)**:
   * This is a concept unique to dynamically-typed languages like Python.
   * It refers to the idea that if an object implements the required behavior (methods or attributes), it can be treated as an instance of a particular type, regardless of its actual class.

### Examples of Polymorphism in Python:

#### 1. Method Overriding (Inheritance-Based Polymorphism):

python

Copy code

class Animal:

def make\_sound(self):

print("Animal makes a sound")

class Dog(Animal):

def make\_sound(self):

print("Dog barks")

class Cat(Animal):

def make\_sound(self):

print("Cat meows")

# Creating instances of different classes

animal = Animal()

dog = Dog()

cat = Cat()

# Calling the same method on different objects

animal.make\_sound() # Output: Animal makes a sound

dog.make\_sound() # Output: Dog barks

cat.make\_sound() # Output: Cat meows

In this example:

* The method make\_sound() is defined in the Animal class, and it's overridden in the Dog and Cat subclasses.
* When the method is called on instances of Dog and Cat, their respective implementations are executed, demonstrating polymorphism.

#### 2. Duck Typing (Behavioral Polymorphism):

Duck typing is a concept where the suitability of an object is determined by the presence of certain methods or properties, rather than its actual class type. This allows objects of different classes to be treated interchangeably if they implement the required method or behavior.

python

Copy code

class Bird:

def fly(self):

print("Bird is flying")

class Airplane:

def fly(self):

print("Airplane is flying")

def lets\_fly(flyable\_object):

flyable\_object.fly()

# Creating instances of Bird and Airplane

bird = Bird()

airplane = Airplane()

# Passing objects with the same interface (fly method)

lets\_fly(bird) # Output: Bird is flying

lets\_fly(airplane) # Output: Airplane is flying

In this case:

* Even though Bird and Airplane are completely unrelated classes, they both implement the fly() method, so they can be treated interchangeably. This demonstrates duck typing, where polymorphism is achieved based on behavior, not class hierarchy.

#### 3. Polymorphism with Operator Overloading:

In Python, polymorphism can also be demonstrated through **operator overloading**, where the behavior of operators like +, -, etc., can be customized for user-defined classes.

python

Copy code

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

# Overloading the + operator to add two Point objects

def \_\_add\_\_(self, other):

return Point(self.x + other.x, self.y + other.y)

def \_\_repr\_\_(self):

return f"Point({self.x}, {self.y})"

# Creating Point objects

point1 = Point(1, 2)

point2 = Point(3, 4)

# Using the overloaded + operator

result = point1 + point2

print(result) # Output: Point(4, 6)

Here:

* The \_\_add\_\_ method is used to overload the + operator so that adding two Point objects results in a new Point object with summed coordinates.
* This is an example of how polymorphism can also be applied to operators.

### Summary of Polymorphism in Python:

* **Method Overriding**: Polymorphism allows subclasses to provide a different implementation of a method defined in a superclass. The version of the method that gets executed depends on the type of the object.
* **Duck Typing**: In Python, polymorphism can also be achieved without considering the class of an object, as long as it implements the required method or behavior. This is often referred to as "duck typing" because of the saying: "If it looks like a duck and quacks like a duck, it must be a duck."
* **Operator Overloading**: Polymorphism can also be seen in how operators (e.g., +, -, \*) behave differently based on the types of operands they are applied to.

Overall, polymorphism increases flexibility and makes your code more reusable and adaptable to changes.