Unit 1

Python Introduction

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.

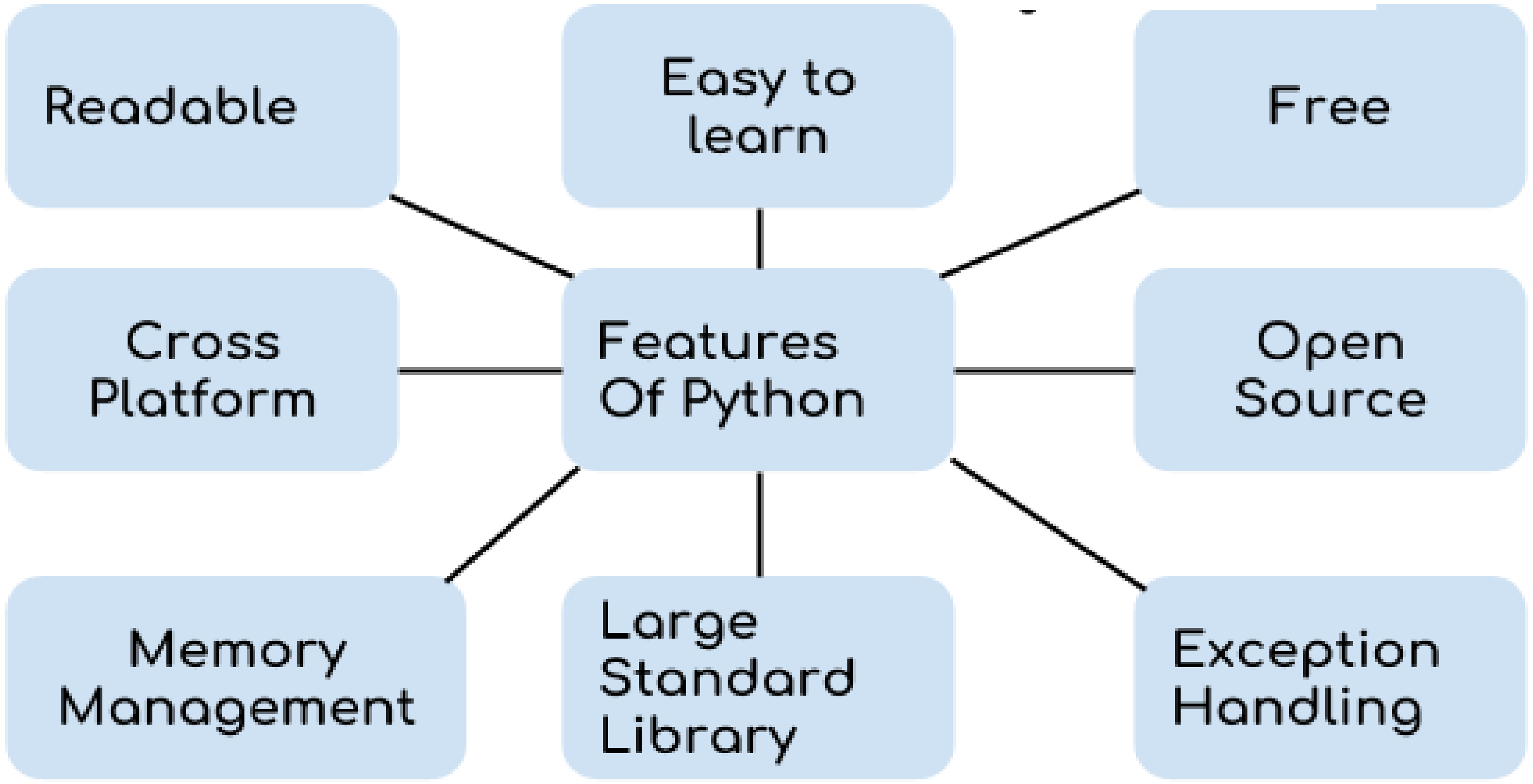
Why Python?

- 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.

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 this purpose.

Features of Python programming language



1. **Readable:** Python is a very readable language.

2. **Easy to Learn:** Learning python is easy as this is a expressive and high level programming language, which means it is easy to understand the language and thus easy to learn.

3. **Cross platform:** Python is available and can run on various operating systems such as Mac, Windows, Linux, Unix etc. This makes it a cross platform and portable language.

4. **Open Source:** Python is a open source programming language.

5. **Large standard library:** Python comes with a large standard library that has some handy codes and functions which we can use while writing code in Python.

6. **Free:** Python is free to download and use. This means you can download it for free and use it in your application. Python is an example of a FLOSS (Free/Libre Open Source Software), which means you can freely distribute copies of this software, read its source code and modify it.

7. **Supports exception handling:** If you are new, you may wonder what is an exception? An exception is an event that can occur during program exception and can disrupt the normal flow of program. Python supports exception handling which means we can write less error prone code and can test various scenarios that can cause an exception later on.

8. **Automatic memory management:** Python supports automatic memory management which means the memory is cleared and freed automatically. You do not have to bother clearing the memory.

Dynamic Semantic :

Do not need to initialize anything before using it.

Python Comments

- 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.

1. Single Line Comments :

Comments starts with a '#'

Example :

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

2. Multi Line Comments :

- Python does not really have a syntax for multi line 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!")
```

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 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)
```

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))
```

Single or Double Quotes?

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

Example

```
x = "John"
# is the same as
x = 'John'
```

Case-Sensitive

Variable names are case-sensitive.

Example

This will create two variables:

```
a = 4
A = "Sally"
#A will not overwrite a
```

Python - 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)

Multi Words Variable Names

- Variable names with more than one word can be difficult to read.
- There are several techniques you can use to make them more readable:

Camel Case

Each word, except the first, starts with a capital letter:

```
myVariableName = "John"
```

Pascal Case

Each word starts with a capital letter:

```
MyVariableName = "John"
```

Snake Case

Each word is separated by an underscore character:

```
my_variable_name = "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)
```

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)
```

Unpack a Collection

If you have a collection of values in a list, tuple etc.

Python allows you extract the values into variables. This is called *unpacking*.

Example

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

Python - Output Variables

The Python `print` statement is often used to output variables.

To combine both text and a variable, Python uses the `+` character:

Example

```
x = "awesome"
print("Python is " + x)
```

You can also use the `+` character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z =  x + y
print(z)
```

For numbers, the `+` character works as a mathematical operator:

Example

```
x = 5
y = 10
print(x + y)
```

If you try to combine a string and a number, Python will give you an error:

Example

```
x = 5
y = "John"
print(x + y)
```

Python Data Types

Built-in Data Types

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

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

Setting the Data Type

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

Python 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 = 1j   # complex
```

1. Int :

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

Example

```
x = 1
y = 35656222554887711
z = -3255522

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

2. Float :

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

Example

```
x = 1.10
y = 1.0
z = -35.59

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

Float can also be scientific numbers with an "e" to indicate the power of 10.

Example

```
x = 35e3
y = 12E4
z = -87.7e100
```

3. Complex :

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

Example

```
x = 3+5j
y = 5j
z = -5j

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


A = x + 2
print(A)
B = x + 2.5
Print(B)
```

Creating Complex Data Type Using complex()

We can create complex number from two real numbers. Syntax for doing this is:

c = complex(a,b)

Example :

```
>>> a=5
>>> b=7
>>> c=complex(a,b)
>>> print(c) (5+7j)
```

Accessing Real and Imaginary Part From Complex Number

we can access real and imaginary part using built-in data descriptors `real` and `imag`.

```
>>> a = 5+6j
>>> a.real
5.0
>>> a.imag
6.0
```

Reading Complex Number From User

We can read complex number directly from user using built-in function input(). Since function input() returns STRING we must convert result to complex using function complex()

```
a = complex(input('Enter complex number:’))
print('Given complex number is:',a)
```

Type Conversion

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

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

Random Number

Python does not have a `random()` function to make a random number, but Python has a built-in module called `random` that can be used to make random numbers:

Example

```python
import random

print(random.randrange(1, 10))
```

Python Casting

- Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.
- Casting in python is therefore done using constructor functions:

`int()` - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)

`float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)

`str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Example

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3

x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2

x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

Python Operators

Python divides the operators in the following groups:

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

Arithmetic Operators

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

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Assignment Operators

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 &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

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
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Example

```
a = [1,2,3]
b = [1,2,3]

a is b
b is a

c = a
a is c
 a.append(4)

a is c
```

Membership Operators

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

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Example

a = [1,2,3,4]

2 in a

 3 not in a

6 in a

Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operato r	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

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, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

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

- List items are ordered, changeable (mutable), 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 that will change the order, but in general: the order of the items will not change.

Changeable / Mutable

- 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

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

Length

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

Example

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

A list can contain different data types:

Example

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

The list() Constructor

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

Example

```
thislist = list(("apple", "banana", "cherry")) # note the double round-
brackets
print(thislist)
```

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])
```

- **Negative Indexing**

`-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])
```

Nested List :

Example :

```
My_list = [1,2,3,4,[5,6,7]]
My_list[4]
My_list[4][0]

X = [1,2.0,3+4j,”List”]
X[3]
X[3][0]
```

Slicing / 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.
- Start index is included and end index is excluded while slicing.

Example

Return the third, fourth, and fifth item:

```
thislist =["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

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

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])
```

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:])
```

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])
```

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")
```

Change List Items

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

Example

Change the second item:

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

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 "blackcurrant" and "watermelon":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

- 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)
```

- 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

```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

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:

Syntax

```
list.insert(pos, elmnt)
```

Example

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

Append Items

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

Syntax

list.append(*elmnt*)

Example

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Extend List

Syntax

list.extend(*iterable*)

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

Example

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

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)
```

Remove List Items

Syntax

list.remove(*elmnt*)

Remove Specified Item

The `remove()` method removes the specified item.

Example

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Remove Specified Index

Syntax

list.pop(*pos*)

The `pop()` method removes the specified index.

Example

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

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

Example

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

The del keyword also removes the specified index:

Example

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

The del keyword can also delete the list completely.

Example

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

Clear the List

Syntax

```
list.clear()
```

The `clear()` method empties the list.

The list still remains, but it has no content.

Example

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

Loop Lists

(1) Using for loop:

Example

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

(2) Loop Through the Index Numbers :

You can also loop through the list items by referring to their index number.

Use the `range()` and `len()` functions to create a suitable iterable.

Example

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

(3) Using while loop :

You can loop through the list items by using a `while` loop.

Example

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

Sort Lists

Syntax

```
list.sort(reverse=True|False, key=myFunc)
```

Sort List Alphanumerically

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

Example

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

Sort Descending

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

Example

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

Case Insensitive Sort

By default the `sort()` method is case sensitive, resulting in all capital letters being sorted before lower case letters:

if you want a case-insensitive sort function, use str.lower as a key function:

Example

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?

The `reverse()` method reverses the current sorting order of the elements.

Example

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

Copy Lists

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().`

Syntax

list.copy()

Example

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

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

Example

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

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

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

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

Example

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)
```

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

Example

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

index() Method

The `index()` method returns the position at the first occurrence of the specified value.

Syntax

list.index(*elmnt*)

Example

```
fruits = [4, 55, 64, 32, 16, 32]

x = fruits.index(32)
```

Note: The `index()` method only returns the *first* occurrence of the value.

If … Else

- Python supports the usual logical conditions from mathematics:
- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

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

Example

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

Elif

The `elif` keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

Else

The `else` keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

```
a = 2
b = 330
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

Example

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

Nested If

Example

```
x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

The pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

Example

```
a = 33
b = 200

if b > a:
  pass
```

Loops

1. The while Loop :

With the `while` loop we can execute a set of statements as long as a condition is true.

Example :

```
i = 1
while i < 6:
  print(i)
  i += 1
```

The else Statement

With the `else` statement we can run a block of code once when the condition no longer is true:

Example

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

2. The For Loop :

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

Example

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Looping Through a String

Example

```
for x in "banana":
  print(x)
```

The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function.

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

```
for x in range(6):
  print(x)
```

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

Example

```
for x in range(2, 6):
  print(x)
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3)

Example

```
for x in range(2, 30, 3):
  print(x)
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

The break Statement

With the `break` statement we can stop the loop before it has looped through all the items:

Example

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Example

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

The `else` block will NOT be executed if the loop is stopped by a `break` statement.

Example

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

The pass Statement

`for` loops cannot be empty, but if you for some reason have a `for` loop with no content, put in the `pass` statement to avoid getting an error.

Example

```
for x in [0, 1, 2]:
  pass
```

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

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

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

Example

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

Tuple Length

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

Example

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

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

```
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

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

A tuple can contain different data types:

Example

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

The tuple() Constructor

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

Example

```
thistuple = tuple(("apple", "banana", "cherry"))
                                        # note the double round-brackets
print(thistuple)
```

Access Tuple Items

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

Example

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

Negative Indexing

Negative indexing means start from the end.

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

Example

```
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

```
thistuple =
("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Check if Item Exists

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

Example

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

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

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Add Items

Once a tuple is created, you cannot add items to it.

Example

```
thistuple = ("apple", "banana", "cherry")
thistuple.append("orange") # This will raise an error
print(thistuple)
```

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

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

Remove Items

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

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

Or you can delete the tuple completely:

Example

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer
exists
```

Unpack Tuples

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

Example

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

Using Asterix *

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list:

Example

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)
```

If the asterix is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Example

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
(green, *tropic, red) = fruits
print(green)
print(tropic)
print(red)
```

Join Tuples

To join two or more tuples you can use the + operator:

Example

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Example

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```

count() Method

The `count()` method returns the number of times a specified value appears in the tuple.

Syntax

```
tuple.count(value)
```

Example

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)
print(x)
```

index() Method

The `index()` method finds the first occurrence of the specified value.

The `index()` method raises an exception if the value is not found.

Syntax

```
tuple.index(value)
```

Example

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.index(8)
print(x)
```

Sets

- Sets are used to store multiple items in a single variable.
- A set is a collection which is both **unordered** and **unindexed**.
- Sets are written with **curly brackets**.

Example

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

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

- Sets 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 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)
```

Get the Length of a Set

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

Example

```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))
```

Set Items - Data Types

Set items can be of any data type:

Example

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

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

type()

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

Example

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

The set() Constructor

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

Example

```
thisset = set(("apple", "banana", "cherry")) # note the double round-
brackets
print(thisset)
```

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

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

for x in thisset:
  print(x)
```

Change Items

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

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Add Sets

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

Example

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)
```

Add Any Iterable

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

Example

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)
```

Remove Set Items

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

Example

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

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

Example

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

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 the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

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

Example

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

x = thisset.pop()

print(x)

print(thisset)
```

Example

The `clear()` method empties the set:

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

Example

The `del` keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

Join 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

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

Keep ONLY the Duplicates

The `intersection_update()` method will keep only the items that are present in both sets.

Example

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.intersection_update(y)

print(x)
```

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

Example

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.intersection(y)

print(z)
```

Keep All, But NOT the Duplicates

The `symmetric_difference_update()` method will keep only the elements that are NOT present in both sets.

Example

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.symmetric_difference_update(y)

print(x)
```

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

Example

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.symmetric_difference(y)

print(z)
```

difference() Method

The `difference()` method returns a set that contains the difference between two sets.

The returned set contains items that exist only in the first set, and not in both sets.

Syntax

```
set.difference(set)
```

Example

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.difference(y)

print(z)
```

difference_update() Method

The `difference_update()` method removes the items that exist in both sets.

The `difference_update()` method is different from the `difference()` method, because the `difference()` method *returns a new set*, without the unwanted items, and the `difference_update()` method *removes* the unwanted items from the original set.

Syntax

```
set.difference_update(set)
```

Example

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

x.difference_update(y)

print(x)
```

isdisjoint() Method

The `isdisjoint()` method returns True if none of the items are present in both sets, otherwise it returns False.

Syntax

```
set.isdisjoint(set)
```

Example

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.isdisjoint(y)

print(z)
```

issubset() Method

The `issubset()` method returns True if all items in the set exists in the specified set, otherwise it retuns False.

Syntax

```
set.issubset(set)
```

Example

```
x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b"}

z = x.issubset(y)

print(z)
```

issuperset() Method

- The `issuperset()` method returns True if all items in the specified set exists in the original set, otherwise it retuns False.

Syntax

```
set.issuperset(set)
```

Example

```
x = {"f", "e", "d", "c", "b", "a"}
y = {"a", "b", "c"}

z = x.issuperset(y)

print(z)
```

Dictionary

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and does not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values:

Example

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

Ordered or 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:

Dictionary Length

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

Example

```
print(len(thisdict))
```

Dictionary Items - Data Types

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

Example

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

Accessing Items

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

Example

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

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

Example

```
x = thisdict.get("model")
```

Get Keys

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

Example

```
x = thisdict.keys()
```

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

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

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

Get Values

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

Example

```
x = thisdict.values()
```

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

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

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Get Items

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

Example

```
x = thisdict.items()
```

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

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

x = car.items()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Check if Key Exists

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

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Change Dictionary Items

Change Values

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

Example

-
  ```
  thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
  }
  thisdict["year"] = 2018
  ```

  Try it Yourself »
-

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

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

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)
```

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

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

Removing Items

There are several methods to remove items from a dictionary:

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

Example

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

The `popitem()` method removes the last inserted item.

Example

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

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

Example

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

The `del` keyword can also delete the dictionary completely:

Example

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

The `clear()` method empties the dictionary:

Example

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

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

```
for x in thisdict:
  print(x)
```

Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

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

```
for x in thisdict.values():
  print(x)
```

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

```
for x in thisdict.keys():
  print(x)
```

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

```
for x, y in thisdict.items():
  print(x, y)
```

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

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Nested Dictionaries

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

Example

```
myfamily = {
  "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:

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

Example

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

fromkeys() Method

The `fromkeys()` method returns a dictionary with the specified keys and the specified value.

Syntax

```
dict.fromkeys(keys, value)
```

Example

```
x = ('key1', 'key2', 'key3')
y = 0
thisdict = dict.fromkeys(x, y)
print(thisdict)
```

Same example as above, but without specifying the value:

```
x = ('key1', 'key2', 'key3')
thisdict = dict.fromkeys(x)
print(thisdict)
```

setdefault() Method

The `setdefault()` method returns the value of the item with the specified key.

If the key does not exist, insert the key, with the specified value

Syntax

```
dictionary.setdefault(keyname, value)
```

Example

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

x = car.setdefault("model", "Bronco")

print(x)
```

Example

```python
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.setdefault("color", "white")

print(x)
```

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

```
a = """This is the string
having multiple line. this is
python programming language."""
print(a)
```

Or three single quotes:

```
a = '' This is the string
having multiple line. this is
python programming language.'''
print(a)
```

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example

```
a = "Hello, World!"
print(a[1])
```

Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example

```
for x in "banana":
  print(x)
```

String Length

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

Example

```
a = "Hello, World!"
print(len(a))
```

Check String

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

Example

```
txt = "The best things in life are free!"
print("free" in txt)
```

Slicing Strings

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

```
b = "Hello, World!"
print(b[2:5])
```

Slice From the Start

```
b = "Hello, World!"
print(b[:5])
```

Slice To the End

```
b = "Hello, World!"
print(b[2:])
```

Negative Indexing

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

Example

```
b = "Hello, World!"
print(b[-5:-2])
```

Modify Strings

Upper Case

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

Example

```
a = "Hello, World!"
print(a.upper())
```

Lower Case

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

Example

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

Remove Whitespace

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

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

Example

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

Replace String

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

Example

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

Split String

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

Example

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

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)
```

String Format

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

```
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

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

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))
```

You 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))
```

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:

Example

You will get an error if you use double quotes inside a string that is surrounded by double quotes:

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

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

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

Escape Characters

Other escape characters used in Python:

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

capitalize() Method

The `capitalize()` method returns a string where the first character is upper case.

Syntax

```
string.capitalize()
```

Example

```
txt = "hello, and welcome to my world."
x = txt.capitalize()
print (x)


txt = "36 is my age."
x = txt.capitalize()
print (x)
```

center() Method

The `center()` method will center align the string, using a specified character (space is default) as the fill character.

Syntax

```
string.center(length, character)
```

Example

```
txt = "banana"
x = txt.center(20)
print(x)
```

```
txt = "banana"
x = txt.center(20, "O")
print(x)
```

count() Method

The `count()` method returns the number of times a specified value appears in the string.

Syntax

```
string.count(value, start, end)
```

Parameter	Description
value	Required. A String. The string to value to search for
start	Optional. An Integer. The position to start the search. Default is 0
end	Optional. An Integer. The position to end the search. Default is the end of the string

Example

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple")
print(x)
```

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple", 10, 24)
print(x)
```

endswith() Method

The `endswith()` method returns True if the string ends with the specified value, otherwise False.

Syntax

```
string.endswith(value, start, end)
```

Example

```
txt = "Hello, welcome to my world."
x = txt.endswith(".")
print(x)
```

```
txt = "Hello, welcome to my world."
x = txt.endswith("my world.")
print(x)
```

```
txt = "Hello, welcome to my world."
x = txt.endswith("my world.", 5, 11)
print(x)
```

index() Method

The `index()` method finds the first occurrence of the specified value.

The `index()` method raises an exception if the value is not found.

Syntax

```
string.index(value, start, end)
```

Example

```
txt = "Hello, welcome to my world."
x = txt.index("welcome")
print(x)
```

```python
txt = "Hello, welcome to my world."
x = txt.index("e")
print(x)
```

```python
txt = "Hello, welcome to my world."
x = txt.index("e", 5, 10)
print(x)
```

find() Method

The `find()` method finds the first occurrence of the specified value.

The `find()` method returns -1 if the value is not found.

The `find()` method is almost the same as the [`index()`] method, the only difference is that the `index()` method raises an exception if the value is not found.

Syntax

```
string.find(value, start, end)
```

Example

```
txt = "Hello, welcome to my world."
x = txt.find("welcome")
print(x)
```

isalnum() Method

The `isalnum()` method returns True if all the characters are alphanumeric, meaning alphabet letter (a-z) and numbers (0-9).

Example of characters that are not alphanumeric: (space)!#%&? etc.

Syntax

```
string.isalnum()
```

Example

```
txt = "Company12"
x = txt.isalnum()
print(x)
```

```
txt = "Company 12"
x = txt.isalnum()
print(x)
```

isalpha() Method

The `isalpha()` method returns True if all the characters are alphabet letters (a-z).

Example of characters that are not alphabet letters: (space)!#%&? etc.

Syntax

```
string.isalpha()
```

Example

```
txt = "CompanyX"
x = txt.isalpha()
print(x)
```

```
txt = "Company10"
x = txt.isalpha()
print(x)
```

isdecimal() Method

The `isdecimal()` method returns True if all the characters are decimals (0-9).

This method is used on unicode objects.

Syntax

```
string.isdecimal()
```

Example

```
txt = "\u0033" #unicode for 3
x = txt.isdecimal()
print(x)
```

isdigit() Method

The `isdigit()` method returns True if all the characters are digits, otherwise False.

Exponents, like ², are also considered to be a digit.

Syntax

```
string.isdigit()
```

Example

```
txt = "50800"
x = txt.isdigit()
print(x)
```

isidentifier() Method

The `isidentifier()` method returns True if the string is a valid identifier, otherwise False.

A string is considered a valid identifier if it only contains alphanumeric letters (a-z) and (0-9), or underscores (_). A valid identifier cannot start with a number, or contain any spaces.

Syntax

```
string.isidentifier()
```

Example

```
a = "MyFolder"
b = "Demo002"
c = "2bring"
d = "my demo"
print(a.isidentifier())
print(b.isidentifier())
print(c.isidentifier())
print(d.isidentifier())
```

islower() Method

The `islower()` method returns True if all the characters are in lower case, otherwise False.

Numbers, symbols and spaces are not checked, only alphabet characters.

Syntax

```
string.islower()
```

Example

```
a = "Hello world!"
b = "hello 123"
c = "mynameisPeter"

print(a.islower())
print(b.islower())
print(c.islower())
```

isnumeric() Method

The `isnumeric()` method returns True if all the characters are numeric (0-9), otherwise False.

Exponents, like ² and ¾ are also considered to be numeric values.

`"-1“` and `"1.5“` are NOT considered numeric values, because *all* the characters in the string must be numeric, and the `-` and the `.` are not.

Syntax

```
string.isnumeric()
```

Example

```
a = "\u0030" #unicode for 0
z=#unicode for &sup2;
c = "10km2"
d = "-1"
e = "1.5"
print(a.isnumeric())
print(b.isnumeric())
print(c.isnumeric())
print(d.isnumeric())
print(e.isnumeric())
```

isprintable() Method

The `isprintable()` method returns True if all the characters are printable, otherwise False.

Example of none printable character can be carriage return and line feed.

Syntax

```
string.isprintable()
```

Examples

```
txt = "Hello!\nAre you #1?"
x = txt.isprintable()
print(x)
```

isspace() Method

The `isspace()` method returns True if all the characters in a string are whitespaces, otherwise False.

Syntax

```
string.isspace()
```

Example

```
txt = "   "
x = txt.isspace()
print(x)


txt = "   s   "
x = txt.isspace()
print(x)
```

istitle() Method

The `istitle()` method returns True if all words in a text start with a upper case letter, AND the rest of the word are lower case letters, otherwise False.

Symbols and numbers are ignored.

Syntax

```
string.istitle()
```

Example

```
a = "HELLO, AND WELCOME TO MY WORLD"
b = "Hello"
c = "22 Names"
d = "This Is %'!?"

print(a.istitle())
print(b.istitle())
print(c.istitle())
print(d.istitle())
```

isupper() Method

The `isupper()` method returns True if all the characters are in upper case, otherwise False.

Numbers, symbols and spaces are not checked, only alphabet characters.

Syntax

```
string.isupper()
```

Example

```
a = "Hello World!"
b = "hello 123"
c = "MY NAME IS PETER"

print(a.isupper())
print(b.isupper())
print(c.isupper())
```

join() Method

The `join()` method takes all items in an iterable and joins them into one string.

A string must be specified as the separator.

Syntax

```
string.join(iterable)
```

Example

```
myTuple = ("John", "Peter", "Vicky")
x = "#".join(myTuple)
print(x)


myDict = {"name": "John", "country": "Norway"}
mySeparator = "TEST"
x = mySeparator.join(myDict)
print(x)
```

ljust() Method

The `ljust()` method will left align the string, using a specified character (space is default) as the fill character.

Syntax

```
string.ljust(length, character)
```

Parameter	Description
length	Required. The length of the returned string
character	Optional. A character to fill the missing space (to the right of the string). Default is " " (space).

Example

```
txt = "banana"
x = txt.ljust(20)
print(x, "is my favorite fruit.")
```

```
txt = "banana"
x = txt.ljust(20, "O")
print(x)
```

partition() Method

The `partition()` method searches for a specified string, and splits the string into a tuple containing three elements.

The first element contains the part before the specified string.

The second element contains the specified string.

The third element contains the part after the string.

Syntax

```
string.partition(value)
```

Example

```
txt = "I could eat bananas all day"
x = txt.partition("bananas")
print(x)
```

```
txt = "I could eat bananas all day"
x = txt.partition("apples")
print(x)
```

replace() Method

The `replace()` method replaces a specified phrase with another specified phrase.

Syntax

```
string.replace(oldvalue, newvalue, count)
```

Example

```
txt = "I like bananas"
x = txt.replace("bananas", "apples")
print(x)
```

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three")
print(x)
```

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three", 2)
print(x)
```

rjust() Method

The `rjust()` method will right align the string, using a specified character (space is default) as the fill character.

Syntax

```
string.rjust(length, character)
```

Example

```
txt = "banana"
x = txt.rjust(20)
print(x, "is my favorite fruit.")


txt = "banana"
x = txt.rjust(20, "O")
      print(x)
```

startswith() Method

The `startswith()` method returns True if the string starts with the specified value, otherwise False.

Syntax

```
string.startswith(value, start, end)
```

Example

```
txt = "Hello, welcome to my world."
x = txt.startswith("Hello")
print(x)
```

```
txt = "Hello, welcome to my world."
x = txt.startswith("wel", 7, 20)
print(x)
```

swapcase() Method

The `swapcase()` method returns a string where all the upper case letters are lower case and vice versa.

Syntax

```
string.swapcase()
```

Example

```
txt = "Hello My Name Is PETER"
x = txt.swapcase()
print(x)
```

title() Method

The `title()` method returns a string where the first character in every word is upper case. Like a header, or a title.

If the word contains a number or a symbol, the first letter after that will be converted to upper case.

Syntax

```
string.title()
```

Example

```
txt = "Welcome to my world"
x = txt.title()
print(x)


txt = "Welcome to my 2nd world"
x = txt.title()
print(x)
```

zfill() Method

The `zfill()` method adds zeros (0) at the beginning of the string, until it reaches the specified length.

If the value of the len parameter is less than the length of the string, no filling is done.

Syntax

```
string.zfill(len)
```

Example

```
a = "hello"
b = "welcome to the jungle"
c = "10.000"

print(a.zfill(10))
print(b.zfill(10))
print(c.zfill(10))
```