Programming LANGUAGE;

THE PYTHON EXPERIENCE

POWERED BY THE PROGRAMMING TENT...

Python Variables

Create a variable
Output both text and a variable
Add a variable to another variable

Python Numbers

Verify the type of an object
Create integers
Create floating point numbers
Create scientific numbers with an "e" to indicate the power of 10
Create complex numbers

Python Casting

Casting - Integers
Casting - Floats
Casting – Strings

Python Strings

Get the character at position 1 of a string
Substring. Get the characters from position 2 to position 5 (not included)
Remove whitespace from the beginning or at the end of a string
Return the length of a string
Convert a string to lower case
Convert a string to upper case
Replace a string with another string
Split a string into substrings

Python Operators

Addition operator
Subtraction operator
Multiplication operator
Division operator
Modulus operator
Assignment operator

Python Lists

Create a list
Access list items
Change the value of a list item
Loop through a list
Check if a list item exists

Get the length of a list
Add an item to the end of a list
Add an item at a specified index
Remove an item
Remove the last item
Remove an item at a specified index
Empty a list
Using the list() constructor to make a list

Python Tuples

Create a tuple
Access tuple items
Change tuple values
Loop through a tuple
Check if a tuple item exists
Get the length of a tuple
Delete a tuple
Using the tuple() constructor to create a tuple

Python Sets

Create a set
Loop through a set
Check if an item exists
Add an item to a set
Add multiple items to a set
Get the length of a set
Remove an item in a set
Remove an item in a set by using the discard() method
Remove the last item in a set by using the pop() method
Empty a set
Delete a set
Using the set() constructor to create a set

Python Dictionaries

Create a dictionary
Access the items of a dictionary
Change the value of a specific item in a dictionary
Print all key names in a dictionary, one by one
Print all values in a dictionary, one by one
Using the values() function to return values of a dictionary
Loop through both keys an values, by using the items() function
Check if a key exists
Get the length of a dictionary
Add an item to a dictionary
Remove an item from a dictionary
Empty a dictionary
Using the dict() constructor to create a dictionary

Python If ... Else

The if statement
The elif statement
The else statement
Short hand if
Short hand if ... else
The and keyword
The or keyword

Python While Loop

The while loop
Using the break statement in a while loop
Using the continue statement in a while loop

Python For Loop

The for loop
Loop through a string
Using the break statement in a for loop
Using the continue statement in a for loop
Using the range() function in a for loop
Else in for loop
Nested for loop

Python Functions

Create and call a function
Function parameters
Default parameter value
Let a function return a value
Recursion

Python Lambda

A lambda function that adds 10 to the number passed in as an argument
A lambda function that multiplies argument a with argument b
A lambda function that sums argument a, b, and c

Python Arrays

Create an array
Access the elements of an array
Change the value of an array element
Get the length of an array
Loop through all elements of an array
Add an element to an array
Remove an element from an array

Python Classes and Objects

Create a class
Create an object
The __init__() Function
Create object methods
The self parameter
Modify object properties
Delete object properties
Delete an object

Python Iterators

Return an iterator from a tuple
Return an iterator from a string
Loop through an iterator
Create an iterator
Stop iteration

Python Modules

Use a module
Variables in module
Re-naming a module
Built-in modules
Using the dir() function
Import from module

Python Dates

Import the datetime module and display the current date
Return the year and name of weekday
Create a date object
The strftime() Method

Python JSON

Convert from JSON to Python
Convert from Python to JSON
Convert Python objects into JSON strings
Convert a Python object containing all the legal data types
Use the indent parameter to define the numbers of indents
Use the separators parameter to change the default separator
Use the sort_keys parameter to specify if the result should be sorted or not

Python RegEx

Search a string to see if it starts with "The" and ends with "Spain"
Using the findall() function
Using the search() function
Using the split() function
Using the sub() function

Python PIP

Using a package

Python Try Except

When an error occurs, print a message
Many exceptions
Use the else keyword to define a block of code to be executed if no errors were raised
Use the finally block to execute code regardless if the try block raises an error or not

---

Python Files

Read a file
Read only parts of a file
Read one line of a file
Loop through the lines of a file to read the whole file, line by line

Python MySQL

Create a connection to a database
Create a database in MySQL
Check if a database exist
Create a table
Check if a table exist
Create primary key when creating a table
Insert a record in a table
Insert multiple rows
Get inserted ID
Select all records from a table
Select only some of the columns in a table
Use the fetchone() method to fetch only one row in a table
Select with a filter
Wildcards characters
Prevent SQL injection
Sort the result of a table alphabetically
Sort the result in a descending order (reverse alphabetically)
Delete records from an existing table
Prevent SQL injection
Delete an existing table
Delete a table if it exist
Update existing records in a table
Prevent SQL injection
Limit the number of records returned from a query
Combine rows from two or more tables, based on a related column between them
LEFT JOIN
RIGHT JOIN

Python MongoDB

Create a database
Check if a database exist
Create a collection
Check if a collection exist
Insert into collection
Return the id field
Insert multiple documents
Insert multiple documents with specified IDs
Find the first document in the selection
Find all documents in the selection
Find only some fields
Filter the result
Advanced query
Filter with regular expressions
Sort the result alphabetically
Sort the result descending (reverse alphabetically)
Delete document
Delete many documents
Delete all documents in a collection
Delete a collection
Update a document
Update many/all documents
Limit the result

START LEARNING

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-orientated 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 (VS code). 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 this purpose.

Python Getting Started

Python Install

Many PCs and Macs will have python already installed.

To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name>python --version
```

To check if you have python installed on a Linux or Mac, then on linux open the command line or on Mac open the Terminal and type:

```
python --version
```

If you find that you do not have python installed on your computer, then you can download it for free from the following website: https://www.python.org/

Python Quickstart

Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.

The way to run a python file is like this on the command line:

```
C:\Users\Your Name>python helloworld.py
```

Where "helloworld.py" is the name of your python file.

Let's write our first Python file, called helloworld.py, which can be done in any text editor.

helloworld.py

```
print("Hello, World!")
```

Simple as that. Save your file. Open your command line, navigate to the directory where you saved your file, and run:

```
C:\Users\Your Name>python helloworld.py
```

The output should read:

```
Hello, World!
```

Congratulations, you have written and executed your first Python program.

The Python Command Line

To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.

Type the following on the Windows, Mac or Linux command line:

```
C:\Users\Your Name>python
```

Or, if the "python" command did not work, you can try "py":

```
C:\Users\Your Name>py
```

From there you can write any python, including our hello world example from earlier in the tutorial:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
```

Which will write "Hello, World!" in the command line:

```
C:\Users\Your Name>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
```

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

```
exit()
```

Python Syntax

Execute Python Syntax

As we learned in the previous page, Python syntax can be executed by writing directly in the Command Line:

```
>>> print("Hello, World!")
Hello, World!
```

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line:

```
C:\Users\Your Name>python myfile.py
```

Python Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

Example

Syntax Error:

```
if 5 > 2:
print("Five is greater than two!")
```

The number of spaces is up to you as a programmer, but it has to be at least one.

Example

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

Example

Syntax Error:

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

Python Variables

In Python variables are created the moment you assign a value to it:

Example

Variables in Python:

```
x = 5
y = "Hello, World!"
```

Python has no command for declaring a variable.

You will learn more about variables in the Python Variables chapter.

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:

Example

Comments in Python:

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

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.

Creating a Comment

Comments starts with a #, and Python will ignore them:

Example

```
#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
```

Comments does not have to be text to explain the code, it can also be used to prevent Python from executing code:

Example

```
#print("Hello, World!")
print("Cheers, Mate!")
```

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!")
```

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!")
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Python Variables

Creating Variables

Variables are containers for storing data values.

Unlike other programming languages, 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)
```

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

Example

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

Variable Names

A 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)

Remember that variable names are case-sensitive

Assign Value 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)
```

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 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)
```

Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example

Create a variable inside a function, with the same name as the global variable

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

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

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the `global` keyword.

Example

If you use the `global` keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

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

Also, use the `global` keyword if you want to change a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword:

```
x = "awesome"
```

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

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

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`, `range`
Mapping Type:	`Dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`Bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`

Getting the Data Type

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

Example

Print the data type of the variable x:

```
x = 5
print(type(x))
```

Setting the Data Type

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

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

ɘ	**Data Type**
lo World"	str
	int
	float

	complex
ɔle", "banana", "cherry"]	list
ɔle", "banana", "cherry")	tuple
ɘ(6)	range
me" : "John", "age" : 36}	dict
ple", "banana", "cherry"}	set
nset({"apple", "banana", "cherry"})	frozenset
	bool
ello"	bytes
array(5)	bytearray
oryview(bytes(5))	memoryview

Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

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

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 = 1j    # 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))
```

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))
```

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

Example

Floats:

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

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

Complex

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

Example

Complex:

```python
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 one type to another:

```python
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))
```

Note: You cannot convert complex numbers into another number type.

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

Import the random module, and display a random number between 1 and 9:

```python
import random

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

Python Casting

Specify a Variable Type

There may be times when you want to specify a type on to a variable. This can be done with 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 rounding down to the previous whole number), 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

Integers:

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

Example

Floats:

```
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
```

Example

Strings:

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

Python Strings

String Literals

String literals 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)
```

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)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

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

Get the character at position 1 (remember that the first character has the position 0):

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

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])
```

Negative Indexing

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

Example

Get the characters from position 5 to position 1, starting the count from the end of the string:

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

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))
```

String Methods

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

Example

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

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

Example

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

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

Example

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

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

Example

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

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

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!']
```

Check String

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

Example

Check if the phrase "ain" is present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x)
```

Example

Check if the phrase "ain" is NOT present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" not in txt
print(x)
```

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)
```

Example

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

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

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))
```

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))
```

String Methods

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

Note: All string methods returns new values. They do not change the original string.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found

isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations

partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case

zfill()	Fills the string with a specified number of 0 values at the beginning

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")
```

Evaluate Values and Variables

The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return,

Example

Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
```

Example

Evaluate two variables:

```
x = "Hello"
y = 15

print(bool(x))
print(bool(y))
```

Most Values are True

Almost any value is evaluated to `True` if it has some sort of content.

Any string is `True`, except empty strings.

Any number is `True`, except `0`.

Any list, tuple, set, and dictionary are `True`, except empty ones.

Example

The following will return True:

```
bool("abc")
bool(123)
bool(["apple", "cherry", "banana"])
```

Some Values are False

In fact, there are not many values that evaluates to `False`, except empty values, such as `()`, `[]`, `{}`, `""`, the number `0`, and the value `None`. And of course the value `False` evaluates to `False`.

Example

The following will return False:

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

One more value, or object in this case, evaluates to `False`, and that is if you have an objects that are made from a class with a `__len__` function that returns `0` or `False`:

Example

```
class myclass():
  def __len__(self):
    return 0

myobj = myclass()
print(bool(myobj))
```

Functions can Return a Boolean

Python also has many built-in functions that returns a boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type:

Example

Check if an object is an integer or not:

```
x = 200
print(isinstance(x, int))
```

Python Operators

Python Operators

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

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

Python Comparison Operators

Comparison operators are used to compare two values:

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

Python Logical Operators

Logical operators are used to combine conditional statements:

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)

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**
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

Python 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

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	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

Python Lists

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** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Example

Create a List:

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

Access Items

You access the list items by referring to the index number:

Example

Print the second item of the list:

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

Negative Indexing

Negative indexing means beginning 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])
```

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])
```

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

Remember that the first item has index 0.

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 index -4 (included) to index -1 (excluded)

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

```
print(thislist[-4:-1])
```

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] = "blackcurrant"
print(thislist)
```

Loop Through a List

You can loop through the list items by using a `for` loop:

Example

Print all items in the list, one by one:

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

You will learn more about `for` loops in our Python For Loops Chapter.

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")
```

List Length

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

Example

Print the number of items in the list:

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

Add 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)
```

To add an item at the specified index, use the `insert()` method:

Example

Insert an item as the second position:

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

Remove Item

There are several methods to remove items from a list:

Example

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

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

Example

The `pop()` method removes the specified index, (or the last item if index is not specified):

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

Example

The `del` keyword removes the specified index:

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

Example

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

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

Example

The `clear()` method empties the list:

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

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)
```

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)
```

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)
```

Another way to join two lists are 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)
```

Or you can use the `extend()` method, which 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)
```

The list() Constructor

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

Example

Using the `list()` constructor to make a List:

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

List Methods

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

Method	**Description**
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position

remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Python Tuples

Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

Example

Create a Tuple:

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

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])
```

Negative Indexing

Negative indexing means beginning 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])
```

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

Remember that the first item has index 0.

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])
```

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)
```

Loop Through a Tuple

You can loop through the tuple items by using a `for` loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

You will learn more about `for` loops in our Python For Loops Chapter.

Check if Item Exists

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

Example

Check if "apple" is present in the tuple:

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

Tuple Length

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

Example

Print the number of items in the tuple:

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

Add Items

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.

Example

You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

Create Tuple With One Item

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

Example

One item tuple, remember the commma:

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Remove Items

Note: You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but 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
```

Join Two Tuples

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

Example

Join two tuples:

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

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

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")) # note the double round-
brackets
print(thistuple)
```

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	**Description**
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

---

Python Sets

Set

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

Example

Create a Set:

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

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

Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

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)
```

Example

Check if "banana" is present in the set:

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

print("banana" in thisset)
```

Change Items

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

Add Items

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

To add more than one item to a set use the `update()` method.

Example

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

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

thisset.add("orange")

print(thisset)
```

Example

Add multiple items to a set, using the `update()` method:

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

thisset.update(["orange", "mango", "grapes"])

print(thisset)
```

Get the Length of a Set

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

Example

Get the number of items in a set:

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

print(len(thisset))
```

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)
```

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)
```

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 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

Remove the last item by using the `pop()` method:

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

x = thisset.pop()

print(x)

print(thisset)
```

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

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 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)
```

Example

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

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

set1.update(set2)
print(set1)
```

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

There are other methods that joins two sets and keeps ONLY the duplicates, or NEVER the duplicates, check the full list of set methods in the bottom of this page.

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")) # note the double round-
brackets
print(thisset)
```

Set Methods

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

Method	**Description**
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not

issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Python Dictionaries

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Example

Create and print a dictionary:

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

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:

```
x = thisdict["model"]
```

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

Example

Get the value of the "model" key:

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

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
```

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:

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

Example

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

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

Example

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

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

Example

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

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

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")
```

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the `len()` method.

Example

Print the number of items in the dictionary:

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

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)
```

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)
```

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)
```

Example

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

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

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()` keyword empties the dictionary:

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

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)
```

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

Example

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

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

Nested Dictionaries

A dictionary can also contain many 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
  }
}
```

Or, if you want to nest three dictionaries that already exists as dictionaries:

Example

Create three dictionaries, than 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
}
```

The dict() Constructor

It is also possible to use the `dict()` constructor to make a new dictionary:

Example

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Dictionary Methods

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

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing the a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Python If ... Else

Python Conditions and If statements

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`

These conditions can be used in several ways, most commonly in "if statements" and loops.

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

Example

If statement:

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

In this example we use two variables, `a` and `b`, which are used as part of the if statement to test whether `b` is greater than `a`. As `a` is `33`, and `b` is `200`, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

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")
```

In this example `a` is equal to `b`, so the first condition is not true, but the `elif` condition is true, so we print to screen that "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")
```

In this example `a` is greater than `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".

You can also have an `else` without the `elif`:

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Short Hand If

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

Example

One line if statement:

```
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

One line if else statement:

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

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

Example

One line if else statement, with 3 conditions:

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

And

The `and` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, AND if `c` is greater than `a`:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

Example

Test if `a` is greater than `b`, OR if `a` is greater than `c`:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

Nested If

You can have `if` statements inside `if` statements, this is called *nested* `if` statements.

Example

```
x = 41

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

Python While Loops

Python Loops

Python has two primitive loop commands:

- `while` loops
- `for` loops

The while Loop

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

Example

Print i as long as i is less than 6:

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

Note: remember to increment i, or else the loop will continue forever.

The `while` loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement

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

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

The else Statement

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

Example

Print a message once the condition is false:

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

Python For Loops

Python For Loops

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

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

Print each fruit in a fruit list:

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

The `for` loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word "banana":

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

The break Statement

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

Example

Exit the loop when `x` is "banana":

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

Example

Exit the loop when `x` is "banana", but this time the break comes before the print:

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

The continue Statement

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

Example

Do not print banana:

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

Using the range() function:

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

Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

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

Using the start parameter:

```
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

Increment the sequence with 3 (default is 1):

```
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

Print all numbers from 0 to 5, and print a message when the loop has ended:

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

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the `def` keyword:

Example

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")

my_function()
```

Parameters

Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without parameter, it uses the default value:

Example

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Passing a List as a Parameter

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

Example

```
def my_function(food):
  for x in food:
    print(x)

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

my_function(fruits)
```

Return Values

To let a function return a value, use the `return` statement:

Example

```
def my_function(x):
  return 5 * x
```

```
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Keyword Arguments

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

Arbitrary Arguments

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example

If the number of arguments are unknown, add a `*` before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (`-1`) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example

Recursion Example

```
def tri_recursion(k):
  if(k>0):
    result = k+tri_recursion(k-1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

Example

A lambda function that adds 10 to the number passed in as an argument, and print the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example

A lambda function that multiplies argument a with argument b and print the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example

A lambda function that sums argument a, b, and c and print the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

Example

```
def myfunc(n):
  return lambda a : a * n
```

```
mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

Python Arrays

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

Arrays

Arrays are used to store multiple values in one single variable:

Example

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

What is an Array?

An array is a special variable, which can hold more than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

The solution is an array!

An array can hold many values under a single name, and you can access the values by referring to an index number.

Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
x = cars[0]
```

Example

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

The Length of an Array

Use the `len()` method to return the length of an array (the number of elements in an array).

Example

Return the number of elements in the `cars` array:

```
x = len(cars)
```

Note: The length of an array is always one more than the highest array index.

Looping Array Elements

You can use the `for in` loop to loop through all the elements of an array.

Example

Print each item in the `cars` array:

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

Adding Array Elements

You can use the `append()` method to add an element to an array.

Example

Add one more element to the `cars` array:

```
cars.append("Honda")
```

Removing Array Elements

You can use the `pop()` method to remove an element from the array.

Example

Delete the second element of the `cars` array:

```
cars.pop(1)
```

You can also use the `remove()` method to remove an element from the array.

Example

Delete the element that has the value "Volvo":

```
cars.remove("Volvo")
```

Note: The list's `remove()` method only removes the first occurrence of the specified value.

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list

copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

Python Classes and Objects

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

Now we can use the class named myClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Note: The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named `self` , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example

Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

Example

Delete the p1 object:

```
del p1
```

Python Inheritance

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named `Person`, with `firstname` and `lastname` properties, and a `printname` method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname
method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

Create a class named `Student`, which will inherit the properties and methods from the `Person` class:

```
class Student(Person):
  pass
```

Note: Use the `pass` keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Example

Use the `Student` class to create an object, and then execute the `printname` method:

```
x = Student("Mike", "Olsen")
x.printname()
```

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Example

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Use the super() Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties

Example

Add a property called graduationyear to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function:

Example

Add a year parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods

Example

Add a method called welcome to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Python Iterators

Python Iterators

An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.

All these objects have a `iter()` method which is used to get an iterator:

Example

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

Even strings are iterable objects, and can return an iterator:

Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a `for` loop to iterate through an iterable object:

Example

Iterate the values of a tuple:

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

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

Example

Iterate the characters of a string:

```
mystr = "banana"

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

The `for` loop actually creates an iterator object and executes the next() method for each loop.

Create an Iterator

To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called `__init__()`, which allows you do some initializing when the object is being created.

The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

Example

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

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a `for` loop.

To prevent the iteration to go on forever, we can use the `StopIteration` statement.

In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example

Stop after 20 iterations:

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

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

Python Scope

A variable is only available from inside the region it is created. This is called **scope**.

Local Scope

A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

Example

A variable created inside a function is available inside that function:

```
def myfunc():
  x = 300
  print(x)

myfunc()
```

Function Inside Function

As explained in the example above, the variable `x` is not available outside the function, but it is available for any function inside the function:

Example

The local variable can be accessed from a function within the function:

```
def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()

myfunc()
```

Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

Example

A variable created outside of a function is global and can be used by anyone:

```
x = 300

def myfunc():
  print(x)

myfunc()

print(x)
```

Naming Variables

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Example

The function will print the local `x`, and then the code will print the global `x`:

```
x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the `global` keyword.

The `global` keyword makes the variable global.

Example

If you use the `global` keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = 300

myfunc()

print(x)
```

Also, use the `global` keyword if you want to make a change to a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword:

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

Python Modules

What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

Example

Save this code in a file named `mymodule.py`

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the `import` statement:

Example

Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the syntax: *module_name.function_name*.

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file `mymodule.py`

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Naming a Module

You can name the module file whatever you like, but it must have the file extension `.py`

Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

Example

Create an alias for `mymodule` called `mx`:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example

Import and use the `platform` module:

```
import platform

x = platform.system()
print(x)
```

Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module.
The `dir()` function:

Example

List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

Note: The dir() function can be used on *all* modules, also the ones you create yourself.

Import From Module

You can choose to import only parts from a module, by using the `from` keyword.

Example

The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example

Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module. Example: `person1["age"]`, **not** ~~`mymodule.person1["age"]`~~

Python Datetime

Python Dates

A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.

Example

Import the datetime module and display the current date:

```
import datetime

x = datetime.datetime.now()
print(x)
```

Date Output

When we execute the code from the example above the result will be:

```
2019-10-01 16:41:59.860212
```

The date contains year, month, day, hour, minute, second, and microsecond.

The `datetime` module has many methods to return information about the date object.

Here are a few examples, you will learn more about them later in this chapter:

Example

Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

Creating Date Objects

To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.

The `datetime()` class requires three parameters to create a date: year, month, day.

Example

Create a date object:

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

The `datetime()` class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of `0`, (`None` for timezone).

The strftime() Method

The `datetime` object has a method for formatting date objects into readable strings.

The method is called `strftime()`, and takes one parameter, `format`, to specify the format of the returned string:

Example

Display the name of the month:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

A reference of all the legal format codes:

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec
%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond 000000-999999	548513
%z	UTC offset	+0100
%Z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%

Python JSON

JSON is a syntax for storing and exchanging data.

JSON is text, written with JavaScript object notation.

JSON in Python

Python has a built-in package called `json`, which can be used to work with JSON data.

Example

Import the json module:

```
import json
```

Parse JSON - Convert from JSON to Python

If you have a JSON string, you can parse it by using the `json.loads()` method.

The result will be a Python dictionary.

Example

Convert from JSON to Python:

```
import json

# some JSON:
x =  '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method.

Example

Convert from Python to JSON:

```
import json

# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)
```

You can convert Python objects of the following types, into JSON strings:

- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

Example

Convert Python objects into JSON strings, and print the values:

```
import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

Python	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	True
False	False
None	Null

Example

Convert a Python object containing all the legal data types:

```
import json

x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}
```

```
print(json.dumps(x))
```

Format the Result

The example above prints a JSON string, but it is not very easy to read, with no indentations and line breaks.

The `json.dumps()` method has parameters to make it easier to read the result:

Example

Use the `indent` parameter to define the numbers of indents:

```
json.dumps(x, indent=4)
```

You can also define the separators, default value is (", ", ": "), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:

Example

Use the `separators` parameter to change the default separator:

```
json.dumps(x, indent=4, separators=(". ", " = "))
```

Order the Result

The `json.dumps()` method has parameters to order the keys in the result:

Example

Use the `sort_keys` parameter to specify if the result should be sorted or not:

```
json.dumps(x, indent=4, sort_keys=True)
```

Python RegEx

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

RegEx Module

Python has a built-in package called `re`, which can be used to work with Regular Expressions.

Import the `re` module:

```
import re
```

RegEx in Python

When you have imported the `re` module, you can start using regular expressions:

Example

Search the string to see if it starts with "The" and ends with "Spain":

```
import re

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
```

RegEx Functions

The `re` module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Metacharacters

Metacharacters are characters with a special meaning:

er	Description	Example
	A set of characters	"[a-m]"
	Signals a special sequence (can also be used to escape special characters)	"\d"
	Any character (except newline character)	"he..o"
	Starts with	"^hello"
	Ends with	"world$"
	Zero or more occurrences	"aix*"
	One or more occurrences	"aix+"
	Exactly the specified number of occurrences	"al{2}"
	Either or	"falls|stays"
	Capture and group	

Special Sequences

A special sequence is a `\` followed by one of the characters in the list below, and has a special meaning:

Character	Description	Try it
\A	Returns a match if the specified characters are at the beginning	Try it »

	of the string	
\b	Returns a match where the specified characters are at the beginning or at the end of a word	Try it » Try it »
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word	Try it » Try it »
\d	Returns a match where the string contains digits (numbers from 0-9)	Try it »
\D	Returns a match where the string DOES NOT contain digits	Try it »
\s	Returns a match where the string contains a white space character	Try it »
\S	Returns a match where the string DOES NOT contain a white space character	Try it »
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	Try it »
\W	Returns a match where the string DOES NOT contain any word characters	Try it »
\Z	Returns a match if the specified characters are at the end of the string	Try it »

Sets

A set is a set of characters inside a pair of square brackets `[]` with a special meaning:

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9

[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

The findall() Function

The `findall()` function returns a list containing all matches.

Example

Print a list of all matches:

```
import re

str = "The rain in Spain"
x = re.findall("ai", str)
print(x)
```

The list contains the matches in the order they are found.

If no matches are found, an empty list is returned:

Example

Return an empty list if no match was found:

```
import re

str = "The rain in Spain"
x = re.findall("Portugal", str)
print(x)
```

The search() Function

The `search()` function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example

Search for the first white-space character in the string:

```
import re

str = "The rain in Spain"
x = re.search("\s", str)

print("The first white-space character is located in position:", x.start())
```

If no matches are found, the value `None` is returned:

Example

Make a search that returns no match:

```
import re

str = "The rain in Spain"
x = re.search("Portugal", str)
print(x)
```

The split() Function

The `split()` function returns a list where the string has been split at each match:

Example

Split at each white-space character:

```
import re

str = "The rain in Spain"
x = re.split("\s", str)
print(x)
```

You can control the number of occurrences by specifying the `maxsplit` parameter:

Example

Split the string only at the first occurrence:

```
import re

str = "The rain in Spain"
x = re.split("\s", str, 1)
print(x)
```

The sub() Function

The `sub()` function replaces the matches with the text of your choice:

Example

Replace every white-space character with the number 9:

```
import re

str = "The rain in Spain"
x = re.sub("\s", "9", str)
print(x)
```

You can control the number of replacements by specifying the `count` parameter:

Example

Replace the first 2 occurrences:

```
import re

str = "The rain in Spain"
x = re.sub("\s", "9", str, 2)
print(x)
```

Match Object

A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value `None` will be returned, instead of the Match Object.

Example

Do a search that will return a Match Object:

```
import re

str = "The rain in Spain"
x = re.search("ai", str)
print(x) #this will print an object
```

The Match object has properties and methods used to retrieve information about the search, and the result:

`.span()` returns a tuple containing the start-, and end positions of the match.
`.string` returns the string passed into the function
`.group()` returns the part of the string where there was a match

Example

Print the position (start- and end-position) of the first match occurrence.

The regular expression looks for any words that starts with an upper case "S":

```
import re

str = "The rain in Spain"
x = re.search(r"\bS\w+", str)
print(x.span())
```

Example

Print the string passed into the function:

```
import re

str = "The rain in Spain"
x = re.search(r"\bS\w+", str)
print(x.string)
```

Example

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

```
import re

str = "The rain in Spain"
x = re.search(r"\bS\w+", str)
print(x.group())
```

Note: If there is no match, the value `None` will be returned, instead of the Match Object.

Python PIP

What is PIP?

PIP is a package manager for Python packages, or modules if you like.

Note: If you have Python version 3.4 or later, PIP is included by default.

What is a Package?

A package contains all the files you need for a module.

Modules are Python code libraries you can include in your project.

Check if PIP is Installed

Navigate your command line to the location of Python's script directory, and type the following:

Example

Check PIP version:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip --version
```

Install PIP

If you do not have PIP installed, you can download and install it from this page: https://pypi.org/project/pip/

Download a Package

Downloading a package is very easy.

Open the command line interface and tell PIP to download the package you want.

Navigate your command line to the location of Python's script directory, and type the following:

Example

Download a package named "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip
install camelcase
```

Now you have downloaded and installed your first package!

Using a Package

Once the package is installed, it is ready to use.

Import the "camelcase" package into your project.

Example

Import and use "camelcase":

```
import camelcase

c = camelcase.CamelCase()

txt = "hello world"

print(c.hump(txt))
```

Find Packages

Find more packages at https://pypi.org/.

Remove a Package

Use the `uninstall` command to remove a package:

Example

Uninstall the package named "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip
uninstall camelcase
```

The PIP Package Manager will ask you to confirm that you want to remove the camelcase package:

```
Uninstalling camelcase-02.1:
  Would remove:
    c:\users\Your Name\appdata\local\programs\python\python36-32\lib\site-
packages\camecase-0.2-py3.6.egg-info
    c:\users\Your Name\appdata\local\programs\python\python36-32\lib\site-
packages\camecase\*
Proceed (y/n)?
```

Press `y` and the package will be removed.

List Packages

Use the `list` command to list all the packages installed on your system:

Example

List installed packages:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip
list
```

Result:

```
Package         Version
-----------------------
camelcase       0.2
mysql-connector 2.1.6
pip             18.1
pymongo         3.6.1
setuptools      39.0.1
```

Python Try Except

The `try` block lets you test a block of code for errors.

The `except` block lets you handle the error.

The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

Example

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

Example

This statement will raise an error, because `x` is not defined:

```
print(x)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a `NameError` and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Else

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
```

```
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

The program can continue, without leaving the file object open.

Python Strings

Command Line Input

Python allows for command line input.

That means we are able to ask the user for input.

The method is a bit different in Python 3.6 than Python 2.7.

Python 3.6 uses the `input()` method.

Python 2.7 uses the `raw_input()` method.

The following example asks for the user's name, and when you entered the name, the name gets printed to the screen:

Python 3.6

```
print("Enter your name:")
x = input()
print("Hello ", x)
```

Python 2.7

```
print("Enter your name:")
x = raw_input()
print("Hello ", x)
```

Save this file as `demo_string_input.py`, and load it through the command line:

```
C:\Users\Your Name>python demo_string_input.py
```

Our program will prompt the user for a string:

```
Enter your name:
```

The user now enters a name:

```
Linus
```

Then, the program prints it to screen with a little message:

```
Hello Linus
```

Python String Formatting

To make sure a string will display as expected, we can format the result with the `format()` method.

String format()

The `format()` method allows you to format selected parts of a string.

Sometimes there are parts of a text that you do not control, maybe they come from a database, or user input?

To control such values, add placeholders (curly brackets `{}`) in the text, and run the values through the `format()` method:

Example

Add a placeholder where you want to display the price:

```
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

You can add parameters inside the curly brackets to specify how to convert the value:

Example

Format the price to be displayed as a number with two decimals:

```
txt = "The price is {:.2f} dollars"
```

Multiple Values

If you want to use more values, just add more values to the format() method:

```
print(txt.format(price, itemno, count))
```

And add more placeholders:

Example

```
quantity = 3
itemno = 567
price = 49
```

```
myorder = "I want {} pieces of item number {} for {:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

Index Numbers

You can use index numbers (a number inside the curly brackets `{0}`) to be sure the values are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

Also, if you want to refer to the same value more than once, use the index number:

Example

```
age = 36
name = "John"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name))
```

Named Indexes

You can also use named indexes by entering a name inside the curly brackets `{carname}`, but then you must use names when you pass the parameter values `txt.format(carname = "Ford")`:

Example

```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname = "Ford", model = "Mustang"))
```

Python File Open

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

File Handling

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

`"r"` - Read - Default value. Opens a file for reading, error if the file does not exist

`"a"` - Append - Opens a file for appending, creates the file if it does not exist

`"w"` - Write - Opens a file for writing, creates the file if it does not exist

`"x"` - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

`"t"` - Text - Default value. Text mode

`"b"` - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because `"r"` for read, and `"t"` for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Python File Open

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in `open()` function.

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the `readline()` method:

Example

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling `readline()` two times, you can read the two first lines:

Example

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

Example

Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Python File Write

Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

`"a"` - Append - will append to the end of the file

`"w"` - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:

`"x"` - Create - will create a file, returns an error if the file exist

`"a"` - Append - will create a file if the specified file does not exist

`"w"` - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Python Delete File

Delete a File

To delete a file, you must import the OS module, and run its `os.remove()` function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

Example

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

Python MySQL

Python can be used in database applications.

One of the most popular databases is MySQL.

MySQL Database

To be able to experiment with the code examples in this tutorial, you should have MySQL installed on your computer.

You can download a free MySQL database at https://www.mysql.com/downloads/.

Install MySQL Driver

Python needs a MySQL driver to access the MySQL database.

In this tutorial we will use the driver "MySQL Connector".

We recommend that you use PIP to install "MySQL Connector".

PIP is most likely already installed in your Python environment.

Navigate your command line to the location of PIP, and type the following:

Download and install "MySQL Connector":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>python -m pip install mysql-connector
```

Now you have downloaded and installed a MySQL driver.

Test MySQL Connector

To test if the installation was successful, or if you already have "MySQL Connector" installed, create a Python page with the following content:

demo_mysql_test.py:

```python
import mysql.connector
```

If the above code was executed with no errors, "MySQL Connector" is installed and ready to be used.

Create Connection

Start by creating a connection to the database.

Use the username and password from your MySQL database:

demo_mysql_connection.py:

```python
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)

print(mydb)
```

Now you can start querying the database using SQL statements.

Python MySQL Create Database

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

Example

create a database named "mydatabase":

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE DATABASE mydatabase")
```

If the above code was executed with no errors, you have successfully created a database.

Check if Database Exists

You can check if a database exist by listing all databases in your system by using the "SHOW DATABASES" statement:

Example

Return a list of your system's databases:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)

mycursor = mydb.cursor()

mycursor.execute("SHOW DATABASES")

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

Or you can try to access the database when making the connection:

Example

Try connecting to the database "mydatabase":

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
```
If the database does not exist, you will get an error.

Python MySQL Create Table

Creating a Table

To create a table in MySQL, use the "CREATE TABLE" statement.

Make sure you define the name of the database when you create the connection

Example

Create a table named "customers":

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address
VARCHAR(255))")
```

If the above code was executed with no errors, you have now successfully created a table.

Check if Table Exists

You can check if a table exist by listing all tables in your database with the "SHOW TABLES" statement:

Example

Return a list of your system's databases:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SHOW TABLES")

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

Primary Key

When creating a table, you should also create a column with a unique key for each record.

This can be done by defining a PRIMARY KEY.

We use the statement "INT AUTO_INCREMENT PRIMARY KEY" which will insert a unique number for each record. Starting at 1, and increased by one for each record.

Example

Create primary key when creating the table:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(255), address VARCHAR(255))")
```

If the table already exists, use the ALTER TABLE keyword:

Example

Create primary key on an existing table:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
```

```
mycursor = mydb.cursor()

mycursor.execute("ALTER TABLE customers ADD COLUMN id INT AUTO_INCREMENT
PRIMARY KEY")
```

Python MySQL Insert Into Table

Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

Example

Insert a record in the "customers" table:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)

mydb.commit()

print(mycursor.rowcount, "record inserted.")
```

Important!: Notice the statement: `mydb.commit()`. It is required to make the changes, otherwise no changes are made to the table.

Insert Multiple Rows

To insert multiple rows into a table, use the `executemany()` method.

The second parameter of the `executemany()` method is a list of tuples, containing the data you want to insert:

Example

Fill the "customers" table with data:

```python
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = [
  ('Peter', 'Lowstreet 4'),
  ('Amy', 'Apple st 652'),
  ('Hannah', 'Mountain 21'),
  ('Michael', 'Valley 345'),
  ('Sandy', 'Ocean blvd 2'),
  ('Betty', 'Green Grass 1'),
  ('Richard', 'Sky st 331'),
  ('Susan', 'One way 98'),
  ('Vicky', 'Yellow Garden 2'),
  ('Ben', 'Park Lane 38'),
  ('William', 'Central st 954'),
  ('Chuck', 'Main Road 989'),
  ('Viola', 'Sideway 1633')
]

mycursor.executemany(sql, val)

mydb.commit()

print(mycursor.rowcount, "was inserted.")
```

Get Inserted ID

You can get the id of the row you just inserted by asking the cursor object.

Note: If you insert more that one row, the id of the last inserted row is returned.

Example

Insert one row, and return the ID:

```python
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)
```

```
mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("Michelle", "Blue Village")
mycursor.execute(sql, val)

mydb.commit()

print("1 record inserted, ID:", mycursor.lastrowid)
```

Python MySQL Select From

Select From a Table

To select from a table in MySQL, use the "SELECT" statement:

Example

Select all records from the "customers" table, and display the result:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchall()

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

Note: We use the `fetchall()` method, which fetches all rows from the last executed statement.

Selecting Columns

To select only some of the columns in a table, use the "SELECT" statement followed by the column name(s):

Example

Select only the name and address columns:

```python
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT name, address FROM customers")

myresult = mycursor.fetchall()

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

Using the fetchone() Method

If you are only interested in one row, you can use the `fetchone()` method.

The `fetchone()` method will return the first row of the result:

Example

Fetch only one row:

```python
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchone()

print(myresult)
```

Python MySQL Where

Select With a Filter

When selecting records from a table, you can filter the selection by using the "WHERE" statement:

Example

Select record(s) where the address is "Park Lane 38": result:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address ='Park Lane 38'"

mycursor.execute(sql)

myresult = mycursor.fetchall()

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

Wildcard Characters

You can also select the records that starts, includes, or ends with a given letter or phrase.

Use the `%` to represent wildcard characters:

Example

Select records where the address contains the word "way":

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address LIKE '%way%'"

mycursor.execute(sql)
```

```
myresult = mycursor.fetchall()

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

Prevent SQL Injection

When query values are provided by the user, you should escape the values.

This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database.

The mysql.connector module has methods to escape query values:

Example

Escape query values by using the placholder `%s` method:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )

mycursor.execute(sql, adr)

myresult = mycursor.fetchall()

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

Python MySQL Order By

Sort the Result

Use the ORDER BY statement to sort the result in ascending or descending order.

The ORDER BY keyword sorts the result ascending by default. To sort the result in descending order, use the DESC keyword.

Example

Sort the result alphabetically by name: result:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT * FROM customers ORDER BY name"

mycursor.execute(sql)

myresult = mycursor.fetchall()

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

ORDER BY DESC

Use the DESC keyword to sort the result in a descending order.

Example

Sort the result reverse alphabetically by name:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()
```

```
sql = "SELECT * FROM customers ORDER BY name DESC"

mycursor.execute(sql)

myresult = mycursor.fetchall()

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

Python MySQL Delete From By

Delete Record

You can delete records from an existing table by using the "DELETE FROM" statement:

Example

Delete any record where the address is "Mountain 21":

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "DELETE FROM customers WHERE address = 'Mountain 21'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

Important!: Notice the statement: `mydb.commit()`. It is required to make the changes, otherwise no changes are made to the table.

Notice the WHERE clause in the DELETE syntax: The WHERE clause specifies which record(s) that should be deleted. If you omit the WHERE clause, all records will be deleted!

Prevent SQL Injection

It is considered a good practice to escape the values of any query, also in delete statements.

This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database.

The mysql.connector module uses the placeholder `%s` to escape values in the delete statement:

Example

Escape values by using the placeholder `%s` method:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "DELETE FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )

mycursor.execute(sql, adr)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

Python MySQL Drop Table

Delete a Table

You can delete an existing table by using the "DROP TABLE" statement:

Example

Delete the table "customers":

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "DROP TABLE customers"
```

```
mycursor.execute(sql)
```

Drop Only if Exist

If the the table you want to delete is already deleted, or for any other reason does not exist, you can use the IF EXISTS keyword to avoid getting an error.

Example

Delete the table "customers" if it exists:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "DROP TABLE IF EXISTS customers"

mycursor.execute(sql)
```

Python MySQL Update Table

Update Table

You can update existing records in a table by using the "UPDATE" statement:

Example

Overwrite the address column from "Valley 345" to "Canyoun 123":

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley
345'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) affected")
```

Important!: Notice the statement: `mydb.commit()`. It is required to make the changes, otherwise no changes are made to the table.

Notice the WHERE clause in the UPDATE syntax: The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

Prevent SQL Injection

It is considered a good practice to escape the values of any query, also in update statements.

This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database.

The mysql.connector module uses the placeholder `%s` to escape values in the delete statement:

Example

Escape values by using the placholder `%s` method:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "UPDATE customers SET address = %s WHERE address = %s"
val = ("Valley 345", "Canyon 123")

mycursor.execute(sql, val)

mydb.commit()

print(mycursor.rowcount, "record(s) affected")
```

Python MySQL Limit

Limit the Result

You can limit the number of records returned from the query, by using the "LIMIT" statement:

Example

Select the 5 first records in the "customers" table:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers LIMIT 5")

myresult = mycursor.fetchall()

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

Start From Another Position

If you want to return five records, starting from the third record, you can use the "OFFSET" keyword:

Example

Start from position 3, and return 5 records:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers LIMIT 5 OFFSET 2")

myresult = mycursor.fetchall()

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

Python MySQL Join

Join Two or More Tables

You can combine rows from two or more tables, based on a related column between them, by using a JOIN statement.

Consider you have a "users" table and a "products" table:

users

```
{ id: 1, name: 'John', fav: 154},
{ id: 2, name: 'Peter', fav: 154},
{ id: 3, name: 'Amy', fav: 155},
{ id: 4, name: 'Hannah', fav:},
{ id: 5, name: 'Michael', fav:}
```

products

```
{ id: 154, name: 'Chocolate Heaven' },
{ id: 155, name: 'Tasty Lemons' },
{ id: 156, name: 'Vanilla Dreams' }
```

These two tables can be combined by using users' `fav` field and products' `id` field.

Example

Join users and products to see the name of the users favorite product:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  INNER JOIN products ON users.fav = products.id"

mycursor.execute(sql)

myresult = mycursor.fetchall()

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

Note: You can use JOIN instead of INNER JOIN. They will both give you the same result.

LEFT JOIN

In the example above, Hannah, and Michael were excluded from the result, that is because INNER JOIN only shows the records where there is a match.

If you want to show all users, even if they do not have a favorite product, use the LEFT JOIN statement:

Example

Select all users and their favorite product:

```
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  LEFT JOIN products ON users.fav = products.id"
```

RIGHT JOIN

If you want to return all products, and the users who have them as their favorite, even if no user have them as their favorite, use the RIGHT JOIN statement:

Example

Select all products, and the user(s) who have them as their favorite:

```
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  RIGHT JOIN products ON users.fav = products.id"
```

Note: Hannah and Michael, who have no favorite product, are not included in the result.

Python MongoDB

Python can be used in database applications.

One of the most popular NoSQL database is MongoDB.

MongoDB

MongoDB stores data in JSON-like documents, which makes the database very flexible and scalable.

To be able to experiment with the code examples in this tutorial, you will need access to a MongoDB database.

You can download a free MongoDB database at https://www.mongodb.com.

PyMongo

Python needs a MongoDB driver to access the MongoDB database.

In this tutorial we will use the MongoDB driver "PyMongo".

We recommend that you use PIP to install "PyMongo".

PIP is most likely already installed in your Python environment.

Navigate your command line to the location of PIP, and type the following:

Download and install "PyMongo":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>python
-m pip install pymongo
```

Now you have downloaded and installed a mongoDB driver.

Test PyMongo

To test if the installation was successful, or if you already have "pymongo" installed, create a Python page with the following content:

demo_mongodb_test.py:

```python
import pymongo
```

If the above code was executed with no errors, "pymongo" is installed and ready to be used.

Python MongoDB Create Database

Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

Example

Create a database called "mydatabase":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")

mydb = myclient["mydatabase"]
```

Important: In MongoDB, a database is not created until it gets content!

MongoDB waits until you have created a collection (table), with at least one document (record) before it actually creates the database (and collection).

Check if Database Exists

Remember: In MongoDB, a database is not created until it gets content, so if this is your first time creating a database, you should complete the next two chapters (create collection and create document) before you check if the database exists!

You can check if a database exist by listing all databases in you system:

Example

Return a list of your system's databases:

```
print(myclient.list_database_names())
```

Or you can check a specific database by name:

Example

Check if "mydatabase" exists:

```
dblist = myclient.list_database_names()
if "mydatabase" in dblist:
  print("The database exists.")
```

Python MongoDB Create Collection

A **collection** in MongoDB is the same as a **table** in SQL databases.

Creating a Collection

To create a collection in MongoDB, use database object and specify the name of the collection you want to create.

MongoDB will create the collection if it does not exist.

Example

Create a collection called "customers":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]

mycol = mydb["customers"]
```

Important: In MongoDB, a collection is not created until it gets content!

MongoDB waits until you have inserted a document before it actually creates the collection.

Check if Collection Exists

Remember: In MongoDB, a collection is not created until it gets content, so if this is your first time creating a collection, you should complete the next chapter (create document) before you check if the collection exists!

You can check if a collection exist in a database by listing all collections:

Example

Return a list of all collections in your database:

```
print(mydb.list_collection_names())
```

Or you can check a specific collection by name:

Example

Check if the "customers" collection exists:

```
collist = mydb.list_collection_names()
if "customers" in collist:
  print("The collection exists.")
```

Python MongoDB Insert Document

A **document** in MongoDB is the same as a **record** in SQL databases.

Insert Into Collection

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the `insert_one()` method.

The first parameter of the `insert_one()` method is a dictionary containing the name(s) and value(s) of each field in the document you want to insert.

Example

Insert a record in the "customers" collection:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydict = { "name": "John", "address": "Highway 37" }

x = mycol.insert_one(mydict)
```

Return the _id Field

The `insert_one()` method returns a InsertOneResult object, which has a property, `inserted_id`, that holds the id of the inserted document.

Example

Insert another record in the "customers" collection, and return the value of the `_id` field:

```
mydict = { "name": "Peter", "address": "Lowstreet 27" }

x = mycol.insert_one(mydict)
```

```
print(x.inserted_id)
```

If you do not specify an `_id` field, then MongoDB will add one for you and assign a unique id for each document.

In the example above no `_id` field was specified, so MongoDB assigned a unique _id for the record (document).

Insert Multiple Documents

To insert multiple documents into a collection in MongoDB, we use the `insert_many()` method.

The first parameter of the `insert_many()` method is a list containing dictionaries with the data you want to insert:

Example

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
  { "name": "Amy", "address": "Apple st 652"},
  { "name": "Hannah", "address": "Mountain 21"},
  { "name": "Michael", "address": "Valley 345"},
  { "name": "Sandy", "address": "Ocean blvd 2"},
  { "name": "Betty", "address": "Green Grass 1"},
  { "name": "Richard", "address": "Sky st 331"},
  { "name": "Susan", "address": "One way 98"},
  { "name": "Vicky", "address": "Yellow Garden 2"},
  { "name": "Ben", "address": "Park Lane 38"},
  { "name": "William", "address": "Central st 954"},
  { "name": "Chuck", "address": "Main Road 989"},
  { "name": "Viola", "address": "Sideway 1633"}
]

x = mycol.insert_many(mylist)

#print list of the _id values of the inserted documents:
print(x.inserted_ids)
```

The `insert_many()` method returns a InsertManyResult object, which has a property, `inserted_ids`, that holds the ids of the inserted documents.

Insert Multiple Documents, with Specified IDs

If you do not want MongoDB to assign unique ids for you document, you can specify the _id field when you insert the document(s).

Remember that the values has to be unique. Two documents cannot have the same _id.

Example

```
import pymongo
```

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mylist = [
  { "_id": 1, "name": "John", "address": "Highway 37"},
  { "_id": 2, "name": "Peter", "address": "Lowstreet 27"},
  { "_id": 3, "name": "Amy", "address": "Apple st 652"},
  { "_id": 4, "name": "Hannah", "address": "Mountain 21"},
  { "_id": 5, "name": "Michael", "address": "Valley 345"},
  { "_id": 6, "name": "Sandy", "address": "Ocean blvd 2"},
  { "_id": 7, "name": "Betty", "address": "Green Grass 1"},
  { "_id": 8, "name": "Richard", "address": "Sky st 331"},
  { "_id": 9, "name": "Susan", "address": "One way 98"},
  { "_id": 10, "name": "Vicky", "address": "Yellow Garden 2"},
  { "_id": 11, "name": "Ben", "address": "Park Lane 38"},
  { "_id": 12, "name": "William", "address": "Central st 954"},
  { "_id": 13, "name": "Chuck", "address": "Main Road 989"},
  { "_id": 14, "name": "Viola", "address": "Sideway 1633"}
]

x = mycol.insert_many(mylist)

#print list of the _id values of the inserted documents:
print(x.inserted_ids)
```

Python MongoDB Find

In MongoDB we use the **find** and **findOne** methods to find data in a collection.

Just like the **SELECT** statement is used to find data in a table in a MySQL database.

Find One

To select data from a collection in MongoDB, we can use the `find_one()` method.

The `find_one()` method returns the first occurrence in the selection.

Example

Find the first document in the customers collection:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

x = mycol.find_one()

print(x)
```

Find All

To select data from a table in MongoDB, we can also use the `find()` method.

The `find()` method returns all occurrences in the selection.

The first parameter of the `find()` method is a query object. In this example we use an empty query object, which selects all documents in the collection.

No parameters in the find() method gives you the same result as **SELECT *** in MySQL.

Example

Return all documents in the "customers" collection, and print each document:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find():
  print(x)
```

Return Only Some Fields

The second parameter of the `find()` method is an object describing which fields to include in the result.

This parameter is optional, and if omitted, all fields will be included in the result.

Example

Return only the names and addresses, not the _ids:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({},{ "_id": 0, "name": 1, "address": 1 }):
  print(x)
```

You are not allowed to specify both 0 and 1 values in the same object (except if one of the fields is the _id field). If you specify a field with the value 0, all other fields get the value 1, and vice versa:

Example

This example will exclude "address" from the result:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
for x in mycol.find({},{ "address": 0 }):
  print(x)
```

Example

You get an error if you specify both 0 and 1 values in the same object (except if one of the fields is the _id field):

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({},{ "name": 1, "address": 0 }):
  print(x)
```

Python MongoDB Query

Filter the Result

When finding documents in a collection, you can filter the result by using a query object.

The first argument of the `find()` method is a query object, and is used to limit the search.

Example

Find document(s) with the address "Park Lane 38":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Park Lane 38" }

mydoc = mycol.find(myquery)

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

Advanced Query

To make advanced queries you can use modifiers as values in the query object.

E.g. to find the documents where the "address" field starts with the letter "S" or higher (alphabetically), use the greater than modifier: `{"$gt": "S"}`:

Example

Find documents where the address starts with the letter "S" or higher:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$gt": "S" } }

mydoc = mycol.find(myquery)

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

Filter With Regular Expressions

You can also use regular expressions as a modifier.

Regular expressions can only be used to query *strings*.

To find only the documents where the "address" field starts with the letter "S", use the regular expression `{"$regex": "^S"}`:

Example

Find documents where the address starts with the letter "S":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }

mydoc = mycol.find(myquery)

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

Python MongoDB Sort

Sort the Result

Use the `sort()` method to sort the result in ascending or descending order.

The `sort()` method takes one parameter for "fieldname" and one parameter for "direction" (ascending is the default direction).

Example

Sort the result alphabetically by name:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name")

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

Sort Descending

Use the value -1 as the second parameter to sort descending.

```
sort("name", 1) #ascending
sort("name", -1) #descending
```

Example

Sort the result reverse alphabetically by name:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name", -1)

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

Python MongoDB Delete Document

Delete Document

To delete one document, we use the `delete_one()` method.

The first parameter of the `delete_one()` method is a query object defining which document to delete.

Note: If the query finds more than one document, only the first occurrence is deleted.

Example

Delete the document with the address "Mountain 21":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Mountain 21" }

mycol.delete_one(myquery)
```

Delete Many Documents

To delete more than one document, use the `delete_many()` method.

The first parameter of the `delete_many()` method is a query object defining which documents to delete.

Example

Delete all documents were the address starts with the letter S:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": {"$regex": "^S"} }

x = mycol.delete_many(myquery)

print(x.deleted_count, " documents deleted.")
```

Delete All Documents in a Collection

To delete all documents in a collection, pass an empty query object to the `delete_many()` method:

Example

Delete all documents in the "customers" collection:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
x = mycol.delete_many({})

print(x.deleted_count, " documents deleted.")
```

Python MongoDB Drop Collection

Delete Collection

You can delete a table, or collection as it is called in MongoDB, by using the `drop()` method.

Example

Delete the "customers" collection:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mycol.drop()
```

The `drop()` method returns true if the collection was dropped successfully, and false if the collection does not exist.

Python MongoDB Update

Update Collection

You can update a record, or document as it is called in MongoDB, by using the `update_one()` method.

The first parameter of the `update_one()` method is a query object defining which document to update.

Note: If the query finds more than one record, only the first occurrence is updated.

The second parameter is an object defining the new values of the document.

Example

Change the address from "Valley 345" to "Canyon 123":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Valley 345" }
newvalues = { "$set": { "address": "Canyon 123" } }

mycol.update_one(myquery, newvalues)

#print "customers" after the update:
for x in mycol.find():
  print(x)
```

Update Many

To update *all* documents that meets the criteria of the query, use the `update_many()` method.

Example

Update all documents where the address starts with the letter "S":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }
newvalues = { "$set": { "name": "Minnie" } }

x = mycol.update_many(myquery, newvalues)

print(x.modified_count, "documents updated.")
```

Python MongoDB Limit

Limit the Result

To limit the result in MongoDB, we use the `limit()` method.

The `limit()` method takes one parameter, a number defining how many documents to return.

Consider you have a "customers" collection:

Customers

```
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
{'_id': 6, 'name': 'Sandy', 'address': 'Ocean blvd 2'}
{'_id': 7, 'name': 'Betty', 'address': 'Green Grass 1'}
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 9, 'name': 'Susan', 'address': 'One way 98'}
{'_id': 10, 'name': 'Vicky', 'address': 'Yellow Garden 2'}
{'_id': 11, 'name': 'Ben', 'address': 'Park Lane 38'}
{'_id': 12, 'name': 'William', 'address': 'Central st 954'}
{'_id': 13, 'name': 'Chuck', 'address': 'Main Road 989'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
```

Example

Limit the result to only return 5 documents:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myresult = mycol.find().limit(5)

#print the result:
for x in myresult:
  print(x)
```

Python Reference

This section contains a Python reference documentation.

Python Reference

List Methods Dictionary Methods Tuple

Methods Set Methods Keywords

Python Built in Functions

Python has a set of built-in functions.

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number

delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system

hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object

oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
@staticmethod()	Converts a method into a static method
str()	Returns a string object

sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class
tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the __dict__ property of an object
zip()	Returns an iterator, from two or more iterators

Python String Methods

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

Note: All string methods returns new values. They do not change the original string.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string

encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable

isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string

split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Note: All string methods returns new values. They do not change the original string.

Python List/Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list

clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

Python Dictionary Methods

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

Method	Description

clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Python Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Python Set Methods

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

lethod	Description
dd()	Adds an element to the set
lear()	Removes all the elements from the set
opy()	Returns a copy of the set
ifference()	Returns a set containing the difference between two or more sets
ifference_update()	Removes the items in this set that are also included in another, specified set
iscard()	Remove the specified item

ntersection()	Returns a set, that is the intersection of two other sets
ntersection_update()	Removes the items in this set that are not present in other, specified set(s)
sdisjoint()	Returns whether two sets have a intersection or not
ssubset()	Returns whether another set contains this set or not
ssuperset()	Returns whether this set contains another set or not
ɔop()	Removes an element from the set
emove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
ınion()	Return a set containing the union of sets
ıpdate()	Update the set with the union of this set and others

Python File Methods

Python has a set of methods available for the file object.

Method	Description
close()	Closes the file

detach()	Returns the separated raw stream from the buffer
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not
read()	Returns the file content
readable()	Returns whether the file stream can be read or not
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file
seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writeable()	Returns whether the file can be written to or not
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file

Python Keywords

Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers:

Method	Description
and	A logical operator
as	To create an alias
assert	For debugging
break	To break out of a loop
class	To define a class
continue	To continue to the next iteration of a loop
def	To define a function
del	To delete an object
elif	Used in conditional statements, same as else if
else	Used in conditional statements
except	Used with exceptions, what to do when an exception occurs

False	Boolean value, result of comparison operations
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
for	To create a for loop
from	To import specific parts of a module
global	To declare a global variable
if	To make a conditional statement
import	To import a module
in	To check if a value is present in a list, tuple, etc.
is	To test if two variables are equal
lambda	To create an anonymous function
None	Represents a null value
nonlocal	To declare a non-local variable
not	A logical operator
or	A logical operator
pass	A null statement, a statement that will do nothing

raise	To raise an exception
return	To exit a function and return a value
True	Boolean value, result of comparison operations
try	To make a try...except statement
while	To create a while loop
With	Used to simplify exception handling
Yield	To end a function, returns a generator

Python Random Modules

Python has a built-in module that you can use to make random numbers.

The `random` module has a set of methods:

Method	**Description**
seed()	Initialize the random number generator
getstate()	Returns the current internal state of the random number generator
setstate()	Restores the internal state of the random number generator
getrandbits()	Returns a number representing the random bits

randrange()	Returns a random number between the given range
randint()	Returns a random number between the given range
choice()	Returns a random element from the given sequence
choices()	Returns a list with a random selection from the given sequence
shuffle()	Takes a sequence and returns the sequence in a random order
sample()	Returns a given sample of a sequence
random()	Returns a random float number between 0 and 1
uniform()	Returns a random float number between two given parameters
triangular()	Returns a random float number between two given parameters, you can also set a mode parameter to specify the midpoint between the two other parameters
betavariate()	Returns a random float number between 0 and 1 based on the Beta distribution (used in statistics)
expovariate()	Returns a random float number between 0 and 1, or between 0 and -1 if the parameter is negative, based on the Exponential distribution (used in statistics)
gammavariate()	Returns a random float number between 0 and 1 based on the Gamma distribution (used in statistics)
gauss()	Returns a random float number between 0 and 1 based on the Gaussian distribution (used in probability theories)
lognormvariate()	Returns a random float number between 0 and 1 based on a log-normal distribution (used in probability theories)

normalvariate()	Returns a random float number between 0 and 1 based on the normal distribution (used in probability theories)
vonmisesvariate()	Returns a random float number between 0 and 1 based on the von Mises distribution (used in directional statistics)
paretovariate()	Returns a random float number between 0 and 1 based on the Pareto distribution (used in probability theories)
weibullvariate()	Returns a random float number between 0 and 1 based on the Weibull distribution (used in statistics)

Python Requests Modules

Example

Make a request to a web page, and print the response text:

```
import requests

x = requests.get('https://w3schools.com/python/demopage.htm')

print(x.text)
```

Definition and Usage

The `requests` module allows you to send HTTP requests using Python.

The HTTP request returns a Response Object with all the response data (content, encoding, status, etc).

Download and Install the Requests Module

Navigate your command line to the location of PIP, and type the following:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip
install requests
```

Syntax

```
requests.methodname(params)
```

Methods

Method	**Description**
delete(*url*, *args*)	Sends a DELETE request to the specified url
get(*url*, *params, args*)	Sends a GET request to the specified url
head(*url*, *args*)	Sends a HEAD request to the specified url
patch(*url*, *data, args*)	Sends a PATCH request to the specified url
post(*url*, *data, json, args*)	Sends a POST request to the specified url
put(*url*, *data, args*)	Sends a PUT request to the specified url
request(*method*, *url*, *args*)	Sends a request of the specified method to the specified url

How to Remove Duplicates From a Python List

Learn how to remove duplicates from a List in Python.

Example

Remove any duplicates from a List:

```
mylist = ["a", "b", "a", "c", "c"]
mylist = list(dict.fromkeys(mylist))
print(mylist)
```

Example Explained

First we have a List that contains duplicates:

A List with Duplicates

```
mylist = ["a", "b", "a", "c", "c"]
mylist = list(dict.fromkeys(mylist))
print(mylist)
```

Create a dictionary, using the List items as keys. This will automatically remove any duplicates because dictionaries cannot have duplicate keys.

Create a Dictionary

```
mylist = ["a", "b", "a", "c", "c"]
mylist = list( dict.fromkeys(mylist) )
print(mylist)
```

Then, convert the dictionary back into a list:

Convert Into a List

```
mylist = ["a", "b", "a", "c", "c"]
mylist = list( dict.fromkeys(mylist) )
print(mylist)
```

Now we have a List without any duplicates, and it has the same order as the original List.

Print the List to demonstrate the result

Print the List

```
mylist = ["a", "b", "a", "c", "c"]
mylist = list(dict.fromkeys(mylist))
print(mylist)
```

Create a Function

If you like to have a function where you can send your lists, and get them back without duplicates, you can create a function and insert the code from the example above.

Example

```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```

Example Explained

Create a function that takes a List as an argument.

Create a Function

```
def my_function(x):
  return list(dict.fromkeys(x))
```

```
mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```

Create a dictionary, using this List items as keys.

Create a Dictionary

```
def my_function(x):
  return list( dict.fromkeys(x) )

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```

Convert the dictionary into a list.

Convert Into a List

```
def my_function(x):
  return list( dict.fromkeys(x) )

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```

Return the list

Return List

```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```

Call the function, with a list as a parameter:

Call the Function

```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])

print(mylist)
```

Print the result:

Print the Result

```
def my_function(x):
  return list(dict.fromkeys(x))

mylist = my_function(["a", "b", "a", "c", "c"])
```

```
print(mylist)
```

How to Reverse a String in Python

Learn how to reverse a String in Python.

There is no built-in function to reverse a String in Python.

The fastest (and easiest?) way is to use a slice that steps backwards, `-1`.

Example

Reverse the string "Hello World":

```
txt = "Hello World"[::-1]
print(txt)
```

Example Explained

We have a string, "Hello World", which we want to reverse:

The String to Reverse

```
txt = "Hello World" [::-1]
print(txt)
```

Create a slice that starts at the end of the string, and moves backwards.

In this particular example, the slice statement `[::-1]` is the same as `[11:0:-1]` which means start at position 11 (because "Hello "World" has 11 characters), end at position 0, move with the step `-1`, *negative* one, which means one step backwards.

Slice the String

```
txt = "Hello World" [::-1]
print(txt)
```

Now we have a string `txt` that reads "Hello World" backwards.

Print the String to demonstrate the result

Print the List

```
txt = "Hello World"[::-1]
print(txt)
```

Create a Function

If you like to have a function where you can send your strings, and return them backwards, you can create a function and insert the code from the example above.

Example

```
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like backwards")

print(mytxt)
```

Example Explained

Create a function that takes a String as an argument.

Create a Function

```
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like backwards")

print(mytxt)
```

Slice the string starting at the end of the string and move backwards.

Slice the String

```
def my_function(x):
  return x [::-1]

mytxt = my_function("I wonder how this text looks like backwards")

print(mytxt)
```

Return the backward String

Return the String

```
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like backwards")

print(mytxt )
```

Call the function, with a string as a parameter:

Call the Function

```
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like backwards")
```

```python
print(mytxt)
```

Print the result:

Print the Result

```python
def my_function(x):
  return x[::-1]

mytxt = my_function("I wonder how this text looks like backwards")

print(mytxt)
```