PYTHON

Aman Tripathi

INDEX

Python Introduction

What is Python?

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

It is used for:

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

What can Python do?

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

Why Python?

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

Good to know

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

Python Syntax compared to other programming languages

- Python was designed for readability and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.

- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly brackets for this purpose.

Code:

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

Output:

```
Hello, World!
```

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.

File name:

helloworld.py

Code:

```
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
```

Output:

```
Hello, World!
```

Python Version

To check the Python version of the editor, you can find it by importing the sys module:

Code:

```
import sys

print(sys.version)
```

Output:

```
3.12.0 (tags/v3.12.0:0fb18b0, Oct  2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)]
```

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\Aman Tripathi>python
Python 3.8.8 (tags/v3.8.8:024d805, Feb 19 2021, 13:18:16) [MSC v.1928 64 bit
(AMD64)] 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\Aman Tripathi>python
Python 3.8.8 (tags/v3.8.8:024d805, Feb 19 2021, 13:18:16) [MSC v.1928 64 bit
(AMD64)] 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.

Code1:

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

Output:

```
Five is greater than two!
```

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

Code2:

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

Output:

```
File "c:\Users\Aman Tripathi\Desktop\Python code\test4.py", line 2    print("Five
is greater than two!")
^
IndentationError: expected an indented block after 'if' statement on line 1
```

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

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

Code3:

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

Output:

```
File "c:\Users\Aman Tripathi\Desktop\Python code\test5.py", line 3    print("Five
is greater than two!")
IndentationError: unexpected indent
```

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:

Code4:

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

Output:

```
Hello, World!
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:
A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:

Code5:

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

Output:

```
Cheers, Mate!
```

Multiline Comments

Python does not really have a syntax for multiline comments.
To add a multiline comment you could insert a # for each line:

Code6:

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Output:

```
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:

Code7:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Output:

```
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

Variables

Variables are containers for storing data values.

Creating Variables

Python has no command for declaring a variable.

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

Code8:

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

Output:

```
5
John
```

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

Code9:

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

Output:

```
Sally
```

Casting

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

Code10:

```
x = str(3)    #x will be ‘3’
y = int(3)    #y will be 3
z = float(3)  #z will be 3.0
print(x)
print(y)
print(z)
```

Output:

```
3
3
3.0
```

Get the Type

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

Code11:

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

```
print(type(y))
```

Output:

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

Single or Double Quotes?

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

Code12:

```
x = "John"
print(x)
#double quotes are the same as single quotes:
x = 'John'
print(x)
```

Output:

```
John
John
```

Case-Sensitive

Variable names are case-sensitive.

Code13:

```
a = 4
A = "Sally"
#A will not overwrite a
print(a)
print(A)
```

Output:

```
4
Sally
```

Python - Variable Names

Variable Names

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

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- A variable name cannot be any of the Python keywords.

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:

Keyword	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 return a list of values from a generator

Legal variable names:

Code14:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"

print(myvar)
print(my_var)
print(_my_var)
print(myVar)
print(MYVAR)
print(myvar2)
```

Output:

```
John
John
John
John
John
John
```

Illegal variable names:

Code15:

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

#This example will produce an error in the result
```

Output:

```
File "c:\Users\Aman Tripathi\Desktop\Python code\pr16.py", line 1    2myvar =
"John"
  ^
SyntaxError: invalid decimal literal
```

Remember that variable names are case-sensitive

Multi Words Variable Names

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

Camel Case

Each word, except the first, starts with a capital letter:
myVariableName = "John"

Pascal Case

Each word starts with a capital letter: MyVariableName = "John"

Snake Case

Each word is separated by an underscore character:
my_variable_name = "John"

Python Variables - Assign Multiple Values

Many Values to Multiple Variables

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

Code16:

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

Output:

```
Orange
Banana
Cherry
```

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

One Value to Multiple Variables

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

Code17:

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Output:

```
Orange
Orange
Orange
```

Unpack a Collection

If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called unpacking.

Unpack a list:

Code18:

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

Output:

```
apple
banana
cherry
```

Python - Output Variables

Output Variables

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

Code19:

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

Output:

```
Python is awesome
```

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

Code20:

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

Output:

```
Python is awesome
```

You can also use the + operator to output multiple variables:

Code21:

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

Output:

```
Python is awesome
```

Notice the space character after "Python " and "is ", without them the result would be "Pythonisawesome".

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

Code22:

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

Output:

```
15
```

In the print() function, when you try to combine a string and a number with the + operator, Python will give you an error:

Code23:

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

Output:

```
Traceback (most recent call last):
  File "c:\Users\Aman Tripathi\Desktop\Python code\pr24.py", line 3, in
<module>
    print(x + y)
          ~~^~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The best way to output multiple variables in the print() function is to separate them with commas, which even support different data types:

Code24:

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

Output:

```
5 John
```

Python - Global Variables

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.

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

Code25:

```
x = "awesome"

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

myfunc()
```

Output:

```
Python is awesome
```

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.

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

Code26:

```
x = "awesome"

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

Output:

```
Python is fantastic
Python is awesome
```

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.

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

Code27:

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

Output:

```
Python is fantastic
```

Also, use the global keyword if you want to change a global variable inside a function. To change the value of a global variable inside a function, refer to the variable by using the global keyword:

Code28:

```
x = "awesome"

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

myfunc()

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

Output:

```
Python is fantastic
```

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
None Type:	NoneType

Getting the Data Type

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

Code29:

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

Output:

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

Setting the Data Type

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

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

x = memoryview(bytes(5))	memoryview
x = None	NoneType

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

There are three numeric types in Python:

- int
- float
- complex

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

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

Int

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

Code30:

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

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

Output:

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

Float

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

Code31:

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

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

Output:

```
<class 'float'>
<class 'float'>
<class 'float'>
```

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

Code32:

```
x = 35e3
y = 12E4                    #12*10^4
z = -87.7e100

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

Output:

```
<class 'float'>
<class 'float'>
<class 'float'>
```

Complex

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

Code33:

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

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

Output:

```
<class 'complex'>
<class 'complex'>
<class 'complex'>
```

Type Conversion

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

Convert from one type to another:

Code34:

```
#convert from int to float:
x = float(1)

#convert from float to int:
y = int(2.8)

#convert from int to complex:
z = complex(1)

print(x)
print(y)
```

```
print(z)

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

Output:

```
1.0
2
(1+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

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:

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

Code35:

```
import random

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

Output:

```
4
```

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 based on the Exponential distribution (used in statistics)
gammavariate()	Returns a random float number based on the Gamma distribution (used in statistics)
gauss()	Returns a random float number based on the Gaussian distribution (used in probability theories)
lognormvariate()	Returns a random float number based on a log-normal distribution (used in probability theories)
normalvariate()	Returns a random float number based on the normal distribution (used in probability theories)
vonmisesvariate()	Returns a random float number based on the von Mises distribution (used in directional statistics)
paretovariate()	Returns a random float number based on the Pareto distribution (used in probability theories)
weibullvariate()	Returns a random float number based on the Weibull distribution (used in statistics)

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 removing all decimals), or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Integers:

Code36:

```
x = int(1)
y = int(2.8)
z = int("3")
print(x)
print(y)
print(z)
```

Output:

```
1
2
3
```

Floats:

Code37:

```
x = float(1)
y = float(2.8)
z = float("3")
w = float("4.2")
print(x)
print(y)
print(z)
print(w)
```

Output:

```
1.0
2.8
3.0
4.2
```

Strings:

Code38:

```
x = str("s1")
y = str(2)
z = str(3.0)
print(x)
print(y)
print(z)
```

Output:

```
s1
2
3.0
```

Python Strings

Strings

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

'hello' is the same as "hello".

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

Code39:

```
#You can use double or single quotes:

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

Output:

```
Hello
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:

Code40:

```
a = "Hello"
print(a)
```

Output:

```
Hello
```

Multiline Strings

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

You can use three double quotes:

Code41:

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

Output:

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

Or three single quotes:

Code42:

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

Output:

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

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

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

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

Code43:

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

Output:

```
e
```

Looping Through a String

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

Loop through the letters in the word "banana":

Code44:

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

Output:

```
b
a
n
a
n
a
```

String Length

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

Code45:

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

Output:

```
13
```

Check String

To check if a certain phrase or character is present in a string, we can use the keyword in.
Check if "free" is present in the following text:

Code46:

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

Output:

```
True
```

Use it in an if statement:
Print only if "free" is present:

Code47:

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

Output:

```
Yes, 'free' is present.
```

Check if NOT

To check if a certain phrase or character is NOT present in a string, we can use the keyword not in.
Check if "expensive" is NOT present in the following text:

Code48:

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

Output:

```
True
```

Use it in an if statement:
print only if "expensive" is NOT present:

Code49:

```
txt = "The best things in life are free!"
if "expensive" not in txt:
  print("No, 'expensive' is NOT present.")
```

Output:

```
No, 'expensive' is NOT present.
```

Python - Slicing Strings

Slicing

You can return a range of characters by using the slice syntax.
Specify the start index and the end index, separated by a colon, to return a part of the string.
Get the characters from position 2 to position 5 (not included):

Code50:

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

Output:

```
llo
```

Note: The first character has index 0.

Slice From the Start

By leaving out the start index, the range will start at the first character:
Get the characters from the start to position 5 (not included):

Code51:

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

Output:

```
Hello
```

Slice To the End

By leaving out the end index, the range will go to the end:
Get the characters from position 2, and all the way to the end:

Code52:

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

Output:

```
llo, World!
```

Negative Indexing

Use negative indexes to start the slice from the end of the string:
Get the characters:
From: "o" in "World!" (position -5)
To, but not included: "d" in "World!" (position -2):

Code53:

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

Output:

```
orl
```

Python - Modify Strings

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

Upper Case

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

Code54:

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

Output:

```
HELLO, WORLD!
```

Lower Case

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

Code55:

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

Output:

```
hello, world!
```

Remove Whitespace

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

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

Code56:

```
a = " Hello, World! "
print(a.strip())   # returns "Hello, World!"
Try it Yourself »
```

Output:

```
Hello, World!
```

Replace String

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

Code57:

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

Output:

```
Jello, World!
```

Split String

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

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

Code58:

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

Output:

```
['Hello', ' World!']
```

Python - String Concatenation

String Concatenation

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

Merge variable a with variable b into variable c:

Code59:

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

Output:

```
HelloWorld
```

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

Code60:

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

Output:

```
Hello World
```

Python - Format – Strings

String Format

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

Code61:

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

Output:

```
Traceback (most recent call last):
  File "c:\Users\Aman Tripathi\Desktop\Python code\pr62.py", line 2, in <module>
    txt = "My name is John, I am " + age
          ~~~~~~~~~~~~~~~~~~~~~~~~^~~~~
TypeError: can only concatenate str (not "int") to str
```

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

F-Strings

F-String was introduced in Python 3.6, and is now the preferred way of formatting strings.

To specify a string as an f-string, simply put an f in front of the string literal, and add curly brackets {} as placeholders for variables and other operations.

Create an f-string:

Code62:

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

Output:

```
My name is John, I am 36
```

Placeholders and Modifiers

A placeholder can contain variables, operations, functions, and modifiers to format the value.

Add a placeholder for the price variable:

Code63:

```
price = 59
txt = f"The price is {price} dollars"
print(txt)
```

Output:

```
The price is 59 dollars
```

A placeholder can include a modifier to format the value.

A modifier is included by adding a colon : followed by a legal formatting type, like .2f which means fixed point number with 2 decimals:

Display the price with 2 decimals:

Code64:

```
price = 59
txt = f"The price is {price:.2f} dollars"
print(txt)
```

Output:

```
The price is 59.00 dollars
```

A placeholder can contain Python code, like math operations:

Perform a math operation in the placeholder, and return the result:

Code65:

```
txt = f"The price is {20 * 5} dollars"
print(txt)
```

Output:

```
The price is 100 dollars
```

You can perform math operations on variables:

Add taxes before displaying the price:

Code66:

```
price = 59
tax = 0.25
txt = f"The price is {price + (price * tax)} dollars"
print(txt)
```

Output:

```
The price is 73.75 dollars
```

You can perform if...else statements inside the placeholders:

Return "Expensive" if the price is over 50, otherwise return "Cheap":

Code67:

```
price = 49
txt = f"It is very {'Expensive' if price>50 else 'Cheap'}"

print(txt)
```

Output:

```
It is very Cheap
```

Execute Functions in F-Strings

You can execute functions inside the placeholder:

Use the string method upper()to convert a value into upper case letters:

Code68:

```
fruit = "apples"
txt = f"I love {fruit.upper()}"
print(txt)
```

Output:

```
I love APPLES
```

The function does not have to be a built-in Python method, you can create your own functions and use them:

Create a function that converts feet into meters:

Code69:

```
def myconverter(x):
  return x * 0.3048

txt = f"The plane is flying at a {myconverter(30000)} meter altitude"
print(txt)
```

Output:

```
The plane is flying at a 9144.0 meter altitude
```

More Modifiers

At the beginning of this chapter we explained how to use the .2f modifier to format a number into a fixed point number with 2 decimals.

There are several other modifiers that can be used to format values:

Use a comma as a thousand separator:

Code70:

```
price = 59000
txt = f"The price is {price:,} dollars"
print(txt)
```

Output:

```
The price is 59,000 dollars
```

Here is a list of all the formatting types.

:<	Left aligns the result (within the available space)
:>	Right aligns the result (within the available space)
:^	Center aligns the result (within the available space)
:=	Places the sign to the left most position
:+	Use a plus sign to indicate if the result is positive or negative
:-	Use a minus sign for negative values only
:	Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers)
:,	Use a comma as a thousand separator
:_	Use a underscore as a thousand separator

:b	Binary format
:c	Converts the value into the corresponding Unicode character
:d	Decimal format
:e	Scientific format, with a lower case e
:E	Scientific format, with an upper case E
:f	Fix point number format
:F	Fix point number format, in uppercase format (show inf and nan as INF and NAN)
:g	General format
:G	General format (using a upper case E for scientific notations)
:o	Octal format
:x	Hex format, lower case
:X	Hex format, upper case
:n	Number format
:%	Percentage format

String format()

Before Python 3.6 we used the format() method to format strings.

The format() method can still be used, but f-strings are faster and the preferred way to format strings.

The next examples in this page demonstrates how to format strings with the format() method.

The format() method also uses curly brackets as placeholders {}, but the syntax is slightly different:

Add a placeholder where you want to display the price:

Code71:

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

Output:

```
The price is 49 dollars
```

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

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

Code72:

```
price = 49
txt = "The price is {:.2f} dollars"
print(txt.format(price))
```

Output:

```
The price is 49.00 dollars
```

Multiple Values

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

Code73:

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

Output:

```
I want 3 pieces of item number 567 for 49.00 dollars.
```

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:

Code74:

```
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))
```

Output:

```
I want 3 pieces of item number 567 for 49.00 dollars.
```

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

Code75:

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

Output:

```
His name is John. John is 36 years old.
```

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"):

Code76:

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

Output:

```
I have a Ford, it is a Mustang.
```

Python - Escape Characters

Escape Character

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

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

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

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

Code77:

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

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

Output:

```
File "c:\Users\Aman Tripathi\Desktop\Python code\pr78.py", line 1    txt = "We are
the so-called "Vikings" from the north."                                ^^^^^^^
SyntaxError: invalid syntax
```

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

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

Code78:

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

Output:

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

Escape Characters

Other escape characters used in Python:

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

Python - String Methods

String Methods

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

Note: All string methods return 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
isascii()	Returns True if all characters in the string are ascii characters
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:

Code79:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

Output:

```
True
False
False
```

When you run a condition in an if statement, Python returns True or False:
Print a message based on whether the condition is True or False:

Code80:

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

Output:

```
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,
Evaluate a string and a number:

Code81:

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

Output:

```
True
True
```

Evaluate two variables:

Code82:

```
x = "Hello"
y = 15
print(bool(x))
print(bool(y))
```

Output:

```
True
True
```

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.
The following will return True:

Code83:

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

Output:

```
True
True
True
```

Some Values are False

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

Code84:

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

Output:

```
False
False
False
False
False
False
False
```

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

Code85:

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

```
    return 0
myobj = myclass()
print(bool(myobj))
```

Output:

```
False
```

Functions can Return a Boolean

You can create functions that returns a Boolean Value
Print the answer of a function:

Code86:

```
def myFunction() :
  return True
print(myFunction())
```

Output:

```
True
```

You can execute code based on the Boolean answer of a function:
Print "YES!" if the function returns True, otherwise print "NO!":

Code87:

```
def myFunction() :
  return True
if myFunction():
  print("YES!")
else:
  print("NO!")
```

Output:

```
YES!
```

Python also has many built-in functions that return a boolean value, like the isinstance() function, which can be used to determine if an object is of a certain data type:
Check if an object is an integer or not:

Code88:

```
x = "200"
print(isinstance(x, str))
```

Output:

```
True
```

Python Operators

Python Operators

Operators are used to perform operations on variables and values.
In the example below, we use the + operator to add together two values:

Code89:

```
print(10 + 5)
```

Output:

15

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
:=	print(x := 3)	x = 3 print(x)

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	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

Operator Precedence

Operator precedence describes the order in which operations are performed.

Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first:

Code90:

```
print((6 + 3) - (6 + 3))

"""
Parenthesis have the highest precedence, and need to be evaluated first.
The calculation above reads 9 - 9 = 0
"""
```

Output:

```
0
```

Multiplication * has higher precedence than addition +, and therefor multiplications are evaluated before additions:

Code91:

```
print(100 + 5 * 3)

"""
Multiplication has higher precedence than addition, and needs to be evaluated first.
The calculation above reads 100 + 15 = 115
"""
```

Output:

```
115
```

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	**Description**
()	Parentheses
**	Exponentiation
+x -x ~x	Unary plus, unary minus, and bitwise NOT
* / // %	Multiplication, division, floor division, and modulus
+ -	Addition and subtraction
<< >>	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR

== != > >= < <= is is not in not in	Comparisons, identity, and membership operators
not	Logical NOT
and	AND
or	OR

If two operators have the same precedence, the expression is evaluated from left to right.

Addition + and subtraction - has the same precedence, and therefor we evaluate the expression from left to right:

Code92:

```
print(5 + 4 - 7 + 3)

"""
Additions and subtractions have the same precedence, and we need to calculate from left to right.
The calculation above reads:
5 + 4 = 9
9 - 7 = 2
2 + 3 = 5
"""
```

Output:

```
5
```

Python Lists

List

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

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

Create a List:

Code93:

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

Output:

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

List Items

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

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

Ordered

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

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

Note: There are some list methods that will change the order, but in general: the order of the items will not change.

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

Changeable

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

Allow Duplicates

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

Lists allow duplicate values:

Code94:

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

print(thislist)
```

Output:

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

List Length

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

Print the number of items in the list:

Code95:

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

Output:

```
3
```

List Items - Data Types

List items can be of any data type:

String, int and boolean data types:

Code96:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]

print(list1)
print(list2)
print(list3)
```

Output:

```
['apple', 'banana', 'cherry']
[1, 5, 7, 9, 3]
[True, False, False]
```

A list can contain different data types:

A list with strings, integers and boolean values:

Code97:

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

Output:

```
['abc', 34, True, 40, 'male']
```

type()

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

<class 'list'>

What is the data type of a list?

Code98:

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

Output:

```
<class 'list'>
```

The list() Constructor

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

Using the list() constructor to make a List:

Code99:

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

Output:

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

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

*Set items are unchangeable, but you can remove and/or add items whenever you like.
**As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

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.

Python - Access List Items

Access Items

List items are indexed and you can access them by referring to the index number:
Print the second item of the list:

Code100:

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

Output:

```
Banana
```

Note: The first item has index 0.

Negative Indexing

Negative indexing means start from the end
-1 refers to the last item, -2 refers to the second last item etc.
Print the last item of the list:

Code101:

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

Output:

```
Cherry
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new list with the specified items.
Return the third, fourth, and fifth item:

Code102:

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

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

#Remember that the first item is position 0,
#and note that the item in position 5 is NOT included
```

Output:

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

Note: The search will start at index 2 (included) and end at index 5 (not included).
Remember that the first item has index 0.
By leaving out the start value, the range will start at the first item:
This example returns the items from the beginning to, but NOT including, "kiwi":

Code103:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])

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

#Remember that index 0 is the first item, and index 4 is the fifth item
#Remember that the item in index 4 is NOT included
```

Output:

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

By leaving out the end value, the range will go on to the end of the list:
This example returns the items from "cherry" to the end:

Code104:

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

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

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

Output:

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

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the list:
This example returns the items from "orange" (-4) to, but NOT including "mango" (-1):

Code105:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])

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

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

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

Output:

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

Check if Item Exists

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

Code106:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Output:

```
Yes, 'apple' is in the fruits list
```

Python - Change List Items

Change Item Value

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

Change the second item:

Code107:

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

print(thislist)
```

Output:

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

Change a Range of Item Values

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

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

Code108:

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

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

print(thislist)
```

Output:

```
['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

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

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

Code109:

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

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

print(thislist)
```

Output:

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

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

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

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

Code110:

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

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

print(thislist)
```

Output:

```
['apple', 'watermelon', 'Mango']
```

Insert Items

To insert a new list item, without replacing any of the existing values, we can use the insert() method.
The insert() method inserts an item at the specified index:
Insert "watermelon" as the third item:

Code111:

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

thislist.insert(2, "watermelon")

print(thislist)
```

Output:

```
['apple', 'banana', 'watermelon', 'cherry']
```

Note: As a result of the example above, the list will now contain 4 items.

Python - Add List Items

Append Items

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

Code112:

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

thislist.append("orange")

print(thislist)
```

Output:

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

Extend List

To append elements from another list to the current list, use the extend() method.
Add the elements of tropical to thislist:

Code113:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]

thislist.extend(tropical)

print(thislist)
```

Output:

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

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

Add Any Iterable

The extend() method does not have to append lists, you can add any iterable object (tuples, sets, dictionaries etc.).
Add elements of a tuple to a list:

Code114:

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")

thislist.extend(thistuple)

print(thislist)
```

Output:

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

Python - Remove List Items

Remove Specified Item

The remove() method removes the specified item.
Remove "banana":

Code115:

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

Output:

```
['apple', 'cherry']
```

If there are more than one item with the specified value, the remove() method removes the first occurrence:
Remove the first occurrence of "banana":

Code116:

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

Output:

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

Remove Specified Index

The pop() method removes the specified index.
Remove the second item:

Code117:

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

Output:

```
['apple', 'cherry']
```

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

Code118:

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

Output:

```
['apple', 'banana']
```

The del keyword also removes the specified index:
Remove the first item:

Code119:

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

Output:

```
['banana', 'cherry']
```

The del keyword can also delete the list completely.
Delete the entire list:

Code120:

```
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist) #this will cause an error because you have succsesfully deleted
"thislist".
```

Output:

```
Traceback (most recent call last):
  File "c:\Users\Aman Tripathi\Desktop\Python code\pr121.py", line 3, in <module>
    print(thislist) #this will cause an error because you have succsesfully
deleted "thislist".
          ^^^^^^^^
NameError: name 'thislist' is not defined
```

Clear the List

The clear() method empties the list.
The list still remains, but it has no content.
Clear the list content:

Code121:

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

Output:

```
[]
```

Python - Loop Lists

Loop Through a List

You can loop through the list items by using a for loop:
Print all items in the list, one by one:

Code122:

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

Output:

```
Apple
banana
cherry
```

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.
Use the range() and len() functions to create a suitable iterable.
Print all items by referring to their index number:

Code123:

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

Output:

```
Apple
banana
cherry
```

The iterable created in the example above is [0, 1, 2].

Using a While Loop

You can loop through the list items by using a while loop.
Use the len() function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes.
Remember to increase the index by 1 after each iteration.
Print all items, using a while loop to go through all the index numbers

Code124:

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

Output:

```
Apple
banana
cherry
```

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

A short hand for loop that will print all items in a list:

Code125:

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

Output:

```
apple
banana
cherry
```

Python - List Comprehension

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
Example:
Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
Without list comprehension you will have to write a for statement with a conditional test inside:

Code126:

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

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

print(newlist)
```

Output:

```
['apple', 'banana', 'mango']
```

With list comprehension you can do all that with only one line of code:

Code127:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]

print(newlist)
```

Output:

```
['apple', 'banana', 'mango']
```

The Syntax

newlist = [expression for item in iterable if condition == True]
The return value is a new list, leaving the old list unchanged.

Condition

The condition is like a filter that only accepts the items that valuate to True.
Only accept items that are not "apple":

Code128:

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

newlist = [x for x in fruits if x != "apple"]

print(newlist)
```

Output:

```
['banana', 'cherry', 'kiwi', 'mango']
```

The condition if x != "apple" will return True for all elements other than "apple", making the new list contain all fruits except "apple".
The condition is optional and can be omitted:
With no if statement:

Code129:

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

newlist = [x for x in fruits]

print(newlist)
```

Output:

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

Iterable

The iterable can be any iterable object, like a list, tuple, set etc.
You can use the range() function to create an iterable:

Code130:

```
newlist = [x for x in range(10)]

print(newlist)
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Same example, but with a condition:
Accept only numbers lower than 5:

Code131:

```
newlist = [x for x in range(10) if x < 5]

print(newlist)
```

Output:

```
[0, 1, 2, 3, 4]
```

Expression

The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:
Set the values in the new list to upper case:

Code132:

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

newlist = [x.upper() for x in fruits]
```

```
print(newlist)
```

Output:

```
['APPLE', 'BANANA', 'CHERRY', 'KIWI', 'MANGO']
```

You can set the outcome to whatever you like:

Set all values in the new list to 'hello':

Code133:

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

newlist = ['hello' for x in fruits]

print(newlist)
```

Output:

```
['hello', 'hello', 'hello', 'hello', 'hello']
```

The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome:

Return "orange" instead of "banana":

Code134:

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

newlist = [x if x != "banana" else "orange" for x in fruits]

print(newlist)
```

Output:

```
['apple', 'orange', 'cherry', 'kiwi', 'mango']
```

The expression in the example above says:

"Return the item if it is not banana, if it is banana return orange".

Python - Sort Lists

Sort List Alphanumerically

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

Code135:

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

fruits.sort()

print(fruits)
```

Output:

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

Sort the list numerically:

Code136:

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

thislist.sort()

print(thislist)
```

Output:

```
[23, 50, 65, 82, 100]
```

Sort Descending

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

Code137:

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

thislist.sort(reverse = True)

print(thislist)
```

Output:

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

Code138:

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

thislist.sort(reverse = True)

print(thislist)
```

Output:

```
[100, 82, 65, 50, 23]
```

Customize Sort Function

You can also customize your own function by using the keyword argument key = *function*.
The function will return a number that will be used to sort the list (the lowest number first):
Sort the list based on how close the number is to 50:

Code139:

```
def myfunc(n):
  return abs(n - 50)

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

thislist.sort(key = myfunc)

print(thislist)
```

Output:

```
[50, 65, 23, 82, 100]
```

Case Insensitive Sort

By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters:
Case sensitive sorting can give an unexpected result:

Code140:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]

thislist.sort()

print(thislist)
```

Output:

```
['Kiwi', 'Orange', 'banana', 'cherry']
```

Luckily we can use built-in functions as key functions when sorting a list.
So if you want a case-insensitive sort function, use str.lower as a key function:
Perform a case-insensitive sort of the list:

Code141:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]

thislist.sort(key = str.lower)
print(thislist)
```

Output:

```
['banana', 'cherry', 'Kiwi', 'Orange']
```

Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?
The reverse() method reverses the current sorting order of the elements.
Reverse the order of the list items:

Code142:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]

thislist.reverse()

print(thislist)
```

Output:

```
['cherry', 'Kiwi', 'Orange', 'banana']
```

Python - Copy Lists

Copy a List

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

Use the copy() method

You can use the built-in List method copy() to copy a list.

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

Code143:

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

Output:

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

Use the list() method

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

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

Code144:

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

Output:

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

Use the slice Operator

You can also make a copy of a list by using the : (slice) operator.

Make a copy of a list with the : operator:

Code145:

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

Output:

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

Python - Join Lists

Join Two Lists

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

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

Join two list:

Code146:

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

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

Output:

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

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

Append list2 into list1:

Code147:

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

for x in list2:
  list1.append(x)

print(list1)
```

Output:

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

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

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

Code148:

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

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

Output:

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

Top of the Document

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

Tuples are used to store multiple items in a single variable.
Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
A tuple is a collection which is ordered and unchangeable.
Tuples are written with round brackets.
Create a Tuple:

Code149:

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

Output:

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

Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.
Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

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

Allow Duplicates

Since tuples are indexed, they can have items with the same value:
Tuples allow duplicate values:

Code150:

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

Output:

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

Tuple Length

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

Code151:

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

Output:

```
3
```

Create Tuple With One Item

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

One item tuple, remember the comma:

Code152:

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

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

Output:

```
<class 'tuple'>
<class 'str'>
```

Tuple Items - Data Types

Tuple items can be of any data type:

String, int and boolean data types:

Code153:

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

print(tuple1)
print(tuple2)
print(tuple3)
```

Output:

```
('apple', 'banana', 'cherry')
(1, 5, 7, 9, 3)
(True, False, False)
```

A tuple can contain different data types:

A tuple with strings, integers and boolean values:

Code154:

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

print(tuple1)
```

Output:

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

type()

From Python's perspective, tuples are defined as objects with the data type 'tuple':

<class 'tuple'>

What is the data type of a tuple?

Code155:

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

print(type(mytuple))
```

Output:

```
<class 'tuple'>
```

The tuple() Constructor

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

Code156:

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

Output:

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

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

*Set *items* are unchangeable, but you can remove and/or add items whenever you like.
**As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

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.

Python - Access Tuple Items

Access Tuple Items

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

Code157:

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

Output:

```
Banana
```

Note: The first item has index 0.

Negative Indexing

Negative indexing means start from the end.
-1 refers to the last item, -2 refers to the second last item etc.
Print the last item of the tuple:

Code158:

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

Output:

```
Cherry
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new tuple with the specified items.
Return the third, fourth, and fifth item:

Code159:

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

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

#Remember that the first item is position 0,
#and note that the item in position 5 is NOT included
```

Output:

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

Note: The search will start at index 2 (included) and end at index 5 (not included).
Remember that the first item has index 0.
By leaving out the start value, the range will start at the first item:
This example returns the items from the beginning to, but NOT included, "kiwi":

Code160:

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

print(thistuple[:4])
```

Output:

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

By leaving out the end value, the range will go on to the end of the tuple:
This example returns the items from "cherry" and to the end:

Code161:

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

print(thistuple[2:])
```

Output:

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

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:
This example returns the items from index -4 (included) to index -1 (excluded)

Code162:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])

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

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

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

Output:

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

Check if Item Exists

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

Code163:

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

Output:

```
Yes, 'apple' is in the fruits tuple
```

Python - Update Tuples

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

But there are some workarounds.

Change Tuple Values

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

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

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

Code164:

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

Output:

```
('apple', 'kiwi', 'cherry')
```

Add Items

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

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

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

Code165:

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

Output:

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

2. **Add tuple to a tuple**. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Create a new tuple with the value "orange", and add that tuple:

Note: When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple.

Code166:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y
print(thistuple)
```

Output:

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

Remove Items

Note: You cannot remove items in a tuple.

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

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

Code167:

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

Output:

```
('banana', 'cherry')
```

Or you can delete the tuple completely:

The del keyword can delete the tuple completely:

Code168:

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

Output:

```
Traceback (most recent call last):
  File "c:\Users\Aman Tripathi\Desktop\Python code\pr168.py", line 3, in <module>
    print(thistuple) #this will raise an error because the tuple no longer exists
          ^^^^^^^^^
NameError: name 'thistuple' is not defined
```

Python - Unpack Tuples

Unpacking a Tuple

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

Packing a tuple:

Code169:

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

print(fruits)
```

Output:

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

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

Unpacking a tuple:

Code170:

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

(green, yellow, red) = fruits

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

Output:

```
apple
banana
cherry
```

Note: The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Using Asterisk*

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

Assign the rest of the values as a list called "red":

Code171:

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

(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)
print(type(red))
print(type(yellow))
```

Output:

```
Apple
banana
['cherry', 'strawberry', 'raspberry']
<class 'list'>
<class 'str'>
```

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

Add a list of values the "tropic" variable:

Code172:

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")

(green, *tropic, red) = fruits

print(green)
print(tropic)
print(red)
```

Output:

```
apple
['mango', 'papaya', 'pineapple']
Cherry
```

Python - Loop Tuples

Loop Through a Tuple

You can loop through the tuple items by using a for loop.
Iterate through the items and print the values:

Code173:

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

Output:

```
apple
banana
cherry
```

Loop Through the Index Numbers

You can also loop through the tuple items by referring to their index number.
Use the range() and len() functions to create a suitable iterable.
Print all items by referring to their index number:

Code174:

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

Output:

```
apple
banana
cherry
```

Using a While Loop

You can loop through the tuple items by using a while loop.
Use the len() function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by referring to their indexes.
Remember to increase the index by 1 after each iteration.
Print all items, using a while loop to go through all the index numbers:

Code175:

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

Output:

```
apple
banana
cherry
```

Python - Join Tuples

Join Two Tuples

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

Code176:

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

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

Output:

```
('a', 'b', 'c', 1, 2, 3)
```

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the * operator:
Multiply the fruits tuple by 2:

Code177:

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

print(mytuple)
```

Output:

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

Python - Tuple Methods

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

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

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

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

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

Sets are written with curly brackets.

Create a Set:

Code178:

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

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

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

Output:

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

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

Set Items

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

Unordered

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

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

Unchangeable

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

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

Duplicates Not Allowed

Sets cannot have two items with the same value.

Duplicate values will be ignored:

Code179:

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

print(thisset)
```

Output:

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

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

Code180:

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

print(thisset)
```

Output:

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

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

Code181:

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

print(thisset)
```

Output:

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

Get the Length of a Set

To determine how many items a set has, use the len() function.
Get the number of items in a set:

Code182:

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

print(len(thisset))
```

Output:

```
3
```

Set Items - Data Types

Set items can be of any data type:
String, int and boolean data types:

Code183:

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

print(set1)
print(set2)
print(set3)
```

Output:

```
{'cherry', 'apple', 'banana'}
{1, 3, 5, 7, 9}
{False, True}
```

A set can contain different data types:

A set with strings, integers and boolean values:

Code184:

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

print(set1)
```

Output:

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

type()

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

```
<class 'set'>
```

What is the data type of a set?

Code185:

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

print(type(myset))
```

Output:

```
<class 'set'>
```

The set() Constructor

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

Using the set() constructor to make a set:

Code186:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
# Note: the set list is unordered, so the result will display the items in a random order.
```

Output:

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

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

*Set items are unchangeable, but you can remove items and add new items.
**As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

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.

Python - Access Set Items

Access Items

You cannot access items in a set by referring to an index or a key.
But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.
Loop through the set, and print the values:

Code187:

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

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

Output:

```
banana
apple
cherry
```

Check if "banana" is present in the set:

Code188:

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

print("banana" in thisset)
```

Output:

```
True
```

Check if "banana" is NOT present in the set:

Code189:

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

print("banana" not in thisset)
```

Output:

```
False
```

Change Items

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

Python - Add Set Items

Add Items

Once a set is created, you cannot change its items, but you can add new items.
To add one item to a set use the add() method.
Add an item to a set, using the add() method:

Code190:

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

thisset.add("orange")

print(thisset)
```

Output:

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

Add Sets

To add items from another set into the current set, use the update() method.
Add elements from tropical into thisset:

Code191:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

Output:

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

Add Any Iterable

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

Code192:

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]

thisset.update(mylist)

print(thisset)
```

Output:

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

Python - Remove Set Items

Remove Item

To remove an item in a set, use the remove(), or the discard() method.
Remove "banana" by using the remove() method:

Code193:

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

thisset.remove("banana")

print(thisset)
```

Output:

```
{'apple', 'cherry'}
```

Note: If the item to remove does not exist, remove() will raise an error.
Remove "banana" by using the discard() method:

Code194:

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

thisset.discard("banana")

print(thisset)
```

Output:

```
{'cherry', 'apple'}
```

Note: If the item to remove does not exist, discard() will NOT raise an error.
You can also use the pop() method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed.
The return value of the pop() method is the removed item.
Remove a random item by using the pop() method:

Code195:

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

x = thisset.pop()

print(x) #removed item

print(thisset) #the set after removal
```

Output:

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

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

The clear() method empties the set:

Code196:

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

thisset.clear()

print(thisset)
```

Output:

```
set()
```

The del keyword will delete the set completely:

Code197:

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

del thisset

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

Output:

```
Traceback (most recent call last):
  File "c:\Users\Aman Tripathi\Desktop\Python code\pr197.py", line 5, in <module>
    print(thisset) #this will raise an error because the set no longer exists
          ^^^^^^^
NameError: name 'thisset' is not defined
```

Python - Loop Sets

Loop Items

You can loop through the set items by using a for loop:
Loop through the set, and print the values:

Code198:

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

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

Output:

```
cherry
banana
apple
```

Python - Join Sets

Join Sets

There are several ways to join two or more sets in Python.
The union() and update() methods joins all items from both sets.
The intersection() method keeps ONLY the duplicates.
The difference() method keeps the items from the first set that are not in the other set(s).
The symmetric_difference() method keeps all items EXCEPT the duplicates.

Union

The union() method returns a new set with all items from both sets.
Join set1 and set2 into a new set:

Code199:

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

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

Output:

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

You can use the | operator instead of the union() method, and you will get the same result. Use | to join two sets:

Code200:

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

set3 = set1 | set2
print(set3)
```

Output:

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

Join Multiple Sets

All the joining methods and operators can be used to join multiple sets.
When using a method, just add more sets in the parentheses, separated by commas:
Join multiple sets with the union() method:

Code201:

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = {"John", "Elena"}
set4 = {"apple", "bananas", "cherry"}
myset = set1.union(set2, set3, set4)
print(myset)
```

Output:

```
{1, 2, 3, 'a', 'bananas', 'b', 'c', 'John', 'Elena', 'cherry', 'apple'}
```

When using the | operator, separate the sets with more | operators:

Use | to join two sets:

Code202:

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = {"John", "Elena"}
set4 = {"apple", "bananas", "cherry"}

myset = set1 | set2 | set3 |set4
print(myset)
```

Output:

```
{'Elena', 1, 2, 3, 'a', 'c', 'cherry', 'bananas', 'John', 'apple', 'b'}
```

Join a Set and a Tuple

The union() method allows you to join a set with other data types, like lists or tuples.
The result will be a set.
Join a set with a tuple:

Code203:

```
x = {"a", "b", "c"}
y = (1, 2, 3)

z = x.union(y)
print(z)
```

Output:

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

Note: The | operator only allows you to join sets with sets, and not with other data types like you can with the union() method.

Update

The update() method inserts all items from one set into another.
The update() changes the original set, and does not return a new set.
The update() method inserts the items in set2 into set1:

Code204:

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

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

Output:

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

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

Intersection

Keep ONLY the duplicates

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

Join set1 and set2, but keep only the duplicates:

Code205:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}

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

Output:

```
{'apple'}
```

You can use the & operator instead of the intersection() method, and you will get the same result.

Use & to join two sets:

Code206:

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}

set3 = set1 & set2
print(set3)
```

Output:

```
{'apple'}
```

Note: The & operator only allows you to join sets with sets, and not with other data types like you can with the intersection() method.

The intersection_update() method will also keep ONLY the duplicates, but it will change the original set instead of returning a new set.

Keep the items that exist in both set1, and set2:

Code207:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}

set1.intersection_update(set2)
print(set1)
```

Output:

```
{'apple'}
```

The values True and 1 are considered the same value. The same goes for False and 0.

Join sets that contains the values True, False, 1, and 0, and see what is considered as duplicates:

Code208:

```
set1 = {"apple", 1, "banana", 0, "cherry"}
set2 = {False, "google", "microsoft", "apple", True}

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

Output:

```
{False, True, 'apple'}
```

Difference

The difference() method will return a new set that will contain only the items from the first set that are not present in the other set.
Keep all items from set1 that are not in set2:

Code209:

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}

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

Output:

```
{'banana', 'cherry'}
```

You can use the - operator instead of the difference() method, and you will get the same result.
Use - to join two sets:

Code210:

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}

set3 = set1 - set2
print(set3)
```

Output:

```
{'cherry', 'banana'}
```

Note: The - operator only allows you to join sets with sets, and not with other data types like you can with the difference() method.
The difference_update() method will also keep the items from the first set that are not in the other set, but it will change the original set instead of returning a new set.
Use the difference_update() method to keep the items that are not present in both sets:

Code211:

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}
set1.difference_update(set2)
print(set1)
```

Output:

```
{'cherry', 'banana'}
```

Symmetric Differences

The symmetric_difference() method will keep only the elements that are NOT present in both sets. Keep the items that are not present in both sets:

Code212:

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1.symmetric_difference(set2)
print(set3)
```

Output:

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

You can use the ^ operator instead of the symmetric_difference() method, and you will get the same result.
Use ^ to join two sets:

Code213:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1 ^ set2
print(set3)
```

Output:

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

Note: The ^ operator only allows you to join sets with sets, and not with other data types like you can with the symmetric_difference() method.
The symmetric_difference_update() method will also keep all but the duplicates, but it will change the original set instead of returning a new set.
Use the symmetric_difference_update() method to keep the items that are not present in both sets:

Code214:

```
set1 = {"apple", "banana" , "cherry"}
set2 = {"google", "microsoft", "apple"}

set1.symmetric_difference_update(set2)
print(set1)
```

Output:

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

Python - Set Methods

Set Methods

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

Method	Shortcut	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
	<	Returns whether all items in this set is present in other, specified set(s)
issuperset()	>=	Returns whether this set contains another set or not
	>	Returns whether all items in other, specified set(s) is present in this set
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

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

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

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

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

Create and print a dictionary:

Code215:

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

Output:

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

Dictionary Items

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

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

Print the "brand" value of the dictionary:

Code216:

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

Output:

```
Ford
```

Ordered or Unordered?

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

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

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

Changeable

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

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Duplicate values will overwrite existing values:

Code217:

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

Output:

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

Dictionary Length

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

Print the number of items in the dictionary:

Code218:

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

Output:

```
3
```

Dictionary Items - Data Types

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

String, int, boolean, and list data types:

Code219:

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

Output:

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

type()

From Python's perspective, dictionaries are defined as objects with the data type 'dict': <class 'dict'>

Print the data type of a dictionary:

Code220:

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

Output:

```
<class 'dict'>
```

The dict() Constructor

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

Using the dict() method to make a dictionary:

Code221:

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

print(thisdict)
```

Output:

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

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

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

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

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.

Python - Access Dictionary Items

Accessing Items

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

Get the value of the "model" key:

Code222:

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

Output:

```
Mustang
```

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

Get the value of the "model" key:

Code223:

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

Output:

```
Mustang
```

Get Keys

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

Get a list of the keys:

Code224:

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

x = thisdict.keys()

print(x)
```

Output:

```
dict_keys(['brand', 'model', 'year'])
```

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

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

Code225:

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

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

Output:

```
dict_keys(['brand', 'model', 'year'])
dict_keys(['brand', 'model', 'year', 'color'])
```

Get Values

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

Get a list of the values:

Code226:

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

x = thisdict.values()

print(x)
```

Output:

```
dict_values(['Ford', 'Mustang', 1964])
```

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

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

Code227:

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

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Output:

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 2020])
```

Get Items

The items() method will return each item in a dictionary, as tuples in a list.
Get a list of the key:value pairs

Code228:

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

x = thisdict.items()

print(x)
```

Output:

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.
Make a change in the original dictionary, and see that the items list gets updated as well:

Code229:

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

```
x = car.items()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

Output:

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2020)])
```

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

Code230:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.items()
print(x) #before the change
car["color"] = "red"

print(x) #after the change
```

Output:

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964), ('color',
'red')])
```

Check if Key Exists

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

Check if "model" is present in the dictionary:

Code231:

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

Output:

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

Python - Change Dictionary Items

Change Values

You can change the value of a specific item by referring to its key name:
Change the "year" to 2018:

Code232:

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

thisdict["year"] = 2018

print(thisdict)
```

Output:

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

Update Dictionary

The update() method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs. Update the "year" of the car by using the update() method:

Code233:

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

print(thisdict)
```

Output:

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

Python - Add Dictionary Items

Adding Items

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

Code234:

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

Output:

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

Update Dictionary

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

The argument must be a dictionary, or an iterable object with key:value pairs. Add a color item to the dictionary by using the update() method:

Code235:

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

print(thisdict)
```

Output:

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

Python - Remove Dictionary Items

Removing Items

There are several methods to remove items from a dictionary:

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

Code236:

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

Output:

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

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

Code237:

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

Output:

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

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

Code238:

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

Output:

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

The del keyword can also delete the dictionary completely:

Code239:

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

Output:

```
Traceback (most recent call last):
  File "c:\Users\Aman Tripathi\Desktop\Python code\pr239.py", line 7, in <module>
    print(thisdict) #this will cause an error because "thisdict" no longer exists.
^^^^^^^^
NameError: name 'thisdict' is not defined
```

The clear() method empties the dictionary:

Code240:

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

Output:

```
{}
```

Python - Loop Dictionaries

Loop Through a Dictionary

You can loop through a dictionary by using a for loop.
When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.
Print all key names in the dictionary, one by one:

Code241:

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

Output:

```
brand
model
year
```

Print all values in the dictionary, one by one:

Code242:

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

Output:

```
Ford
Mustang
1964
```

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

Code243:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.values():
  print(x)
```

Output:

```
Ford
Mustang
1964
```

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

Code244:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict.keys():
  print(x)
```

Output:

```
brand
model
year
```

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

Code245:

```
thisdict =  {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x, y in thisdict.items():
  print(x, y)
```

Output:

```
brand Ford
model Mustang
year 1964
```

Python - Copy Dictionaries

Copy a Dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.
There are ways to make a copy, one way is to use the built-in Dictionary method copy().
Make a copy of a dictionary with the copy() method:

Code246:

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

Output:

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

Another way to make a copy is to use the built-in function dict().
Make a copy of a dictionary with the dict() function:

Code247:

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

Output:

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

Python - Nested Dictionaries

Nested Dictionaries

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

Create a dictionary that contain three dictionaries:

Code248:

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

print(myfamily)
```

Output:

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

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

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

Code249:

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

myfamily = {
```

```
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}

print(myfamily)
```

Output:

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

Access Items in Nested Dictionaries

To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary: Print the name of child 2:

Code250:

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

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

Output:

```
Tobias
```

Loop Through Nested Dictionaries

You can loop through a dictionary by using the items() method like this:
Loop through the keys and values of all nested dictionaries:

Code251:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
```

```
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}

for x, obj in myfamily.items():
    print(x)

    for y in obj:
        print(y + ':', obj[y])
```

Output:

```
child1
name: Emil
year: 2004
child2
name: Tobias
year: 2007
child3
name: Linus
year: 2011
```

Python Dictionary Methods

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 value
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 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.

If statement:

Code252:

```
a = 33
b = 200

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

Output:

```
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.

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

Code253:

```
a = 33
b = 200

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

Output:

```
File "c:\Users\Aman Tripathi\Desktop\Python code\pr253.py", line 5    print("b is greater than a")    ^
IndentationError: expected an indented block after 'if' statement on line 4
```

Elif

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

Code254:

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

Output:

```
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.

Code255:

```
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")
```

Output:

```
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:

Short Hand If

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

One line if statement:

Code256:

```
a = 200
b = 33

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

Output:

```
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:

One line if else statement:

Code257:

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

Output:

```
B
```

This technique is known as Ternary Operators, or Conditional Expressions.

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

One line if else statement, with 3 conditions:

Code258:

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

Output:

```
=
```

And

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

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

Code259:

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

Output:

```
Both conditions are True
```

Or

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

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

Code260:

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

Output:

```
At least one of the conditions is True
```

Not

The not keyword is a logical operator, and is used to reverse the result of the conditional statement:
Test if a is NOT greater than b:

Code261:

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

Output:

```
a is NOT greater than b
```

Nested If

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

Code262:

```
x = 41

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

Output:

```
Above ten,
and also above 20!
```

The pass Statement

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

Code263:

```
a = 33
b = 200
if b < a:
  pass

# having an empty if statement like this, would raise an error without the pass statement
```

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.
Print i as long as i is less than 6:

Code264:

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

Output:

```
1
2
3
4
5
```

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:
Exit the loop when i is 3:

Code265:

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

Output:

```
1
2
3
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:
Continue to the next iteration if i is 3:

Code266:

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

# Note that number 3 is missing in the result
```

Output:

```
1
2
4
5
6
```

The else Statement

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

Print a message once the condition is false:

Code267:

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

Output:

```
1
2
3
4
5
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.
Print each fruit in a fruit list:

Code268:

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

Output:

```
apple
banana
cherry
```

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:
Loop through the letters in the word "banana":

Code269:

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

Output:

```
b
a
n
a
n
a
```

The break Statement

With the break statement we can stop the loop before it has looped through all the items:
Exit the loop when x is "banana":

Code270:

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

Output:

```
apple
banana
```

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

The continue Statement

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

Do not print banana:

Code271:

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

Output:

```
apple
cherry
```

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.

Using the range() function:

Code272:

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

Output:

```
0
1
2
3
4
5
```

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):

Using the start parameter:

Code273:

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

Output:

```
2
3
4
5
```

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**):
Increment the sequence with 3 (default is 1):

Code274:

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

Output:

```
2
5
8
11
14
17
20
23
26
29
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished: Print all numbers from 0 to 5, and print a message when the loop has ended:

Code275:

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

Output:

```
0
1
2
3
4
5
Finally finished!
```

Note: The else block will NOT be executed if the loop is stopped by a break statement.
Break the loop when x is 3, and see what happens with the else block:

Code276:

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

#If the loop breaks, the else block is not executed.
```

Output:

```
0
1
2
```

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":
Print each adjective for every fruit:

Code277:

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

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

Output:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

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

Code278:

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

Output:

```
apple
banana
cherry
```

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

Code279:

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

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

Output:

```
b
a
n
a
n
a
```

Looping Through an Iterator

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

Iterate the values of a tuple:

Code280:

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

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

Output:

```
apple
banana
cherry
```

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 to 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.

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

Code281:

```
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))
```

Output:

```
1
2
3
4
5
```

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 from going 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:
Stop after 10 iterations:

Code282:

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

  def __next__(self):
    if self.a <= 10:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

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

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Python Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Function Polymorphism

An example of a Python function that can be used on different objects is the len() function.

String

For strings len() returns the number of characters:

Code283:

```
x = "Hello World!"
print(len(x))
```

Output:

12

Tuple

For tuples len() returns the number of items in the tuple:

Code284:

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

Output:

3

Dictionary

For dictionaries len() returns the number of key/value pairs in the dictionary:

Code285:

```
thisdict = {
  "brand": "Mercedes",
  "model": "MayBach",
  "year": 1968,
}
print(len(thisdict))
```

Output:

3

Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move():

Code286:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Drive!")

class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Sail!")

class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747")     #Create a Plane object
for x in (car1, boat1, plane1):
  x.move()
```

Output:

```
Drive!
Sail!
Fly!
```

Look at the for loop at the end. Because of polymorphism we can execute the same method for all three classes.

Inheritance Class Polymorphism

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called Vehicle, and make Car, Boat, Plane child classes of Vehicle, the child classes inherits the Vehicle methods, but can override them:

Create a class called Vehicle and make Car, Boat, Plane child classes of Vehicle:

Code287:

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model
  def move(self):
    print("Move!")
```

```
class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

Output:

```
Ford
Mustang
Move!
Ibiza
Touring 20
Sail!
Boeing
747
Fly!
```

Child classes inherits the properties and methods from the parent class.

In the example above you can see that the Car class is empty, but it inherits brand, model, and move() from Vehicle.

The Boat and Plane classes also inherit brand, model, and move() from Vehicle, but they both override the move() method.

Because of polymorphism we can execute the same method for all classes.

Python Arrays

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

Arrays

Note: This page shows you how to use LISTS as ARRAYS, however, to work with arrays in Python you will have to import a library, like the NumPy library.

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

Create an array containing car names:

Code288:

```
cars = ["Ford", "Volvo", "BMW"]
print(cars)
```

Output:

```
['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*.

Get the value of the first array item:

Code289:

```
cars = ["Ford", "Volvo", "BMW"]
x = cars[0]
print(x)
```

Output:

```
Ford
```

Modify the value of the first array item:

Code290:

```
cars = ["Ford", "Volvo", "BMW"]
cars[0] = "Toyota"
print(cars)
```

Output:

```
['Toyota', 'Volvo', 'BMW']
```

The Length of an Array

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

Return the number of elements in the cars array:

Code291:

```
cars = ["Ford", "Volvo", "BMW"]
x = len(cars)
print(x)
```

Output:

```
3
```

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.

Print each item in the cars array:

Code292:

```
cars = ["Ford", "Volvo", "BMW"]
for x in cars:
  print(x)
```

Output:

```
Ford
Volvo
BMW
```

Adding Array Elements

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

Add one more element to the cars array:

Code293:

```
cars = ["Ford", "Volvo", "BMW"]
cars.append("Honda")
print(cars)
```

Output:

```
['Ford', 'Volvo', 'BMW', 'Honda']
```

Removing Array Elements

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

Delete the second element of the cars array:

Code294:

```
cars = ["Ford", "Volvo", "BMW"]
cars.pop(1)
print(cars)
```

Output:

```
['Ford', 'BMW']
```

You can also use the remove() method to remove an element from the array.
Delete the element that has the value "Volvo":

Code295:

```
cars = ["Ford", "Volvo", "BMW"]
cars.remove("BMW")
print(cars)
```

Output:

```
['Ford', '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:

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

Code296:

```
class MyClass:
  x = 5
print(MyClass)
```

Output:

```
<class '__main__.MyClass'>
```

Create Object

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

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

Code297:

```
class MyClass:
  x = 5
p1 = MyClass()
print(p1.x)
```

Output:

```
5
```

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:

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

Code298:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
```

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

Output:

```
John
36
```

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

The __str__() Function

The __str__() function controls what should be returned when the class object is represented as a string.

If the __str__() function is not set, the string representation of the object is returned:

The string representation of an object WITHOUT the __str__() function:

Code299:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1)
```

Output:

```
<__main__.Person object at 0x00000247DDCD7880>
```

The string representation of an object WITH the __str__() function:

Code300:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def __str__(self):
    return f"{self.name}({self.age})"
p1 = Person("John", 36)
print(p1)
```

Output:

```
John(36)
```

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:

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

Code301:

```
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()
```

Output:

```
Hello my name is John
```

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 belong 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:

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

Code302:

```
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()
```

Output:

```
Hello my name is John
```

Modify Object Properties

You can modify properties on objects like this:

Set the age of p1 to 40:

Code303:

```
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.age = 40
print(p1.age)
```

Output:

```
40
```

Delete Object Properties

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

Delete the age property from the p1 object:

Code304:

```
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)
del p1.age
print(p1.age)
```

Output:

```
Traceback (most recent call last):
  File "pr304.py", line 9, in <module>
    print(p1.age)
AttributeError: 'Person' object has no attribute 'age'
```

Delete Objects

You can delete objects by using the del keyword:

Delete the p1 object:

Code305:

```
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)
del p1
print(p1)
```

Output:

```
Traceback (most recent call last):
  File "pr305.py", line 9, in <module>
    print(p1)
NameError: name 'p1' is not defined
```

The pass Statement

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

Code306:

```
class Person:
  pass
# having an empty class definition like this, would raise an error without the pass
statement
```

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:

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

Code307:

```
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()
```

Output:

```
John Doe
```

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:

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

Code308:

```
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.

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

Code309:

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

```
  pass

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

Output:

```
Mike Olsen
```

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.

Add the __init__() function to the Student class:

Code310:

```
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:

Code311:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
x = Student("Mike", "Olsen")
x.printname()
```

Output:

```
Mike Olsen
```

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:

Code312:

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

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

Output:

```
Mike Olsen
```

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

Add a property called graduationyear to the Student class:

Code313:

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

x = Student("Mike", "Olsen")
print(x.graduationyear)
```

Output:

```
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:

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

Code314:

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

```
    print(self.firstname, self.lastname)
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year
x = Student("Mike", "Olsen", 2019)
print(x.graduationyear)
```

Output:

```
2019
```

Add Methods

Add a method called welcome to the Student class:

Code315:

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

x = Student("Mike", "Olsen", 2024)
x.welcome()
```

Output:

```
Welcome Mike Olsen to the class of 2024
```

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 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.

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

Code316:

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

Output:

```
300
```

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:

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

Code317:

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

Output:

```
300
```

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.

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

Code318:

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

Output:

```
300
300
```

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):

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

Code319:

```
x = 300
def myfunc():
  x = 200
  print(x)
myfunc()
print(x)
```

Output:

```
200
300
```

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.

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

Code320:

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

Output:

```
300
```

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

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

Code321:

```
x = 300
def myfunc():
  global x
  x = 200
myfunc()
print(x)
```

Output:

```
200
```

Nonlocal Keyword

The nonlocal keyword is used to work with variables inside nested functions.
The nonlocal keyword makes the variable belong to the outer function.
If you use the nonlocal keyword, the variable will belong to the outer function:

Code322:

```
def myfunc1():
  x = "Jane"
  def myfunc2():
    nonlocal x
    x = "hello"
  myfunc2()
  return x

print(myfunc1())
```

Output:

```
Hello
```

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:

Code323:

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

Calling a Function

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

Code324:

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

Output:

```
Hello from a function
```

Arguments

Information can be passed into functions as arguments.
Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Code325:

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

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

Output:

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Arguments are often shortened to args in Python documentations.

Parameters or Arguments?

The terms parameter and argument can be used for the same thing: information that are passed into a function.
From a function's perspective:
A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

This function expects 2 arguments, and gets 2 arguments:

Code326:

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil", "Refsnes")
```

Output:

```
Emil Refsnes
```

If you try to call the function with 1 or 3 arguments, you will get an error:

This function expects 2 arguments, but gets only 1:

Code327:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

Output:

```
Traceback (most recent call last):
  File "pr327.py", line 4, in <module>
    my_function("Emil")
TypeError: my_function() missing 1 required positional argument: 'lname'
```

Arbitrary Arguments, *args

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:

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

Code328:

```
def my_function(*kids):
  print("The youngest child is " + kids[0])

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

Output:

```
The youngest child is Emil
```

Arbitrary Arguments are often shortened to *args in Python documentations.

Keyword Arguments

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

This way the order of the arguments does not matter.

Code329:

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

Output:

```
The youngest child is Linus
```

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

Arbitrary Keyword Arguments, **kwargs

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

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

If the number of keyword arguments is unknown, add a double ** before the parameter name:

Code330:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```

Output:

```
His last name is Refsnes
```

Arbitrary Kword Arguments are often shortened to **kwargs in Python documentations.

Default Parameter Value

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

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

Code331:

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

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

Output:

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Passing a List as an Argument

You can send any data types of argument 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 an argument, it will still be a List when it reaches the function:

Code332:

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

Output:

```
Apple
banana
cherry
```

Return Values

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

Code333:

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

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

Output:

```
15
25
45
```

The pass Statement

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

Code334:

```
def myfunction():
  pass

# having an empty function definition like this, would raise an error without the pass statement
```

Positional-Only Arguments

You can specify that a function can have ONLY positional arguments, or ONLY keyword arguments.

To specify that a function can have only positional arguments, add , / after the arguments:

Code335:

```
def my_function(x, /):
  print(x)
my_function(3)
```

Output:

```
3
```

Without the , / you are actually allowed to use keyword arguments even if the function expects positional arguments:

Code336:

```
def my_function(x):
  print(x)
my_function(x = 3)
```

Output:

```
3
```

But when adding the , / you will get an error if you try to send a keyword argument:

Code337:

```
def my_function(x, /):
  print(x)

my_function(x = 3)
```

Output:

```
Traceback (most recent call last):
  File "pr337.py", line 4, in <module>
    my_function(x = 3)
TypeError: my_function() got some positional-only arguments passed as keyword
arguments: 'x'
```

Keyword-Only Arguments

To specify that a function can have only keyword arguments, add *, before the arguments:

Code338:

```
def my_function(*, x):
  print(x)

my_function(x = 3)
```

Output:

```
3
```

Without the *, you are allowed to use positionale arguments even if the function expects keyword arguments:

Code339:

```
def my_function(x):
  print(x)

my_function(3)
```

Output:

```
3
```

But with the *, you will get an error if you try to send a positional argument:

Code340:

```
def my_function(*, x):
  print(x)

my_function(3)
```

Output:

```
Traceback (most recent call last):
  File "pr340.py", line 4, in <module>
    my_function(3)
TypeError: my_function() takes 0 positional arguments but 1 was given
```

Combine Positional-Only and Keyword-Only

You can combine the two argument types in the same function.

Any argument before the / , are positional-only, and any argument after the *, are keyword-only.

Code341:

```
def my_function(a, b, /, *, c, d):
  print(a + b + c + d)

my_function(5, 6, c = 7, d = 8)
```

Output:

```
26
```

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.

Code342:

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

```
print("Recursion Example Results:")
tri_recursion(6)
```

Output:

```
Recursion Example Results:
1
3
6
10
15
21
```

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:

Add 10 to argument a, and return the result:

Code343:

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

Output:

```
15
```

Lambda functions can take any number of arguments:

Multiply argument a with argument b and return the result:

Code344:

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

Output:

```
30
```

Summarize argument a, b, and c and return the result:

Code345:

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

Output:

```
13
```

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:

Code346:

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

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

Code347:

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```

Output:

```
22
```

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

Code348:

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

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

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

Output:

```
22
33
```

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

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:
Save this code in a file named Code349.py

Code349:

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

Use a Module

Now we can use the module we just created, by using the import statement:
Import the module named mymodule, and call the greeting function:

Code350:

```
import Code349

Code349.greeting("Aman")
```

Output:

```
Hello, Aman
```

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):
Save this code in the file Code351.py

Code351:

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

Import the module named Code351, and access the person1 dictionary:

Code352:

```
import Code351

a = Code351.person1
print(a)
```

Output:

```
{'name': 'Aman', 'age': 22, 'country': 'India'}
```

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:
Create an alias for Code351 called cd:

Code353:

```
import Code351 as cd

a = cd.person1["name"]
print(a)
```

Output:

```
Aman
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.
Import and use the platform module:

Code354:

```
import platform

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

Output:

```
Windows
```

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:
List all the defined names belonging to the platform module:

Code355:

```
import platform

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

Output:

```
['_WIN32_CLIENT_RELEASES', '_WIN32_SERVER_RELEASES', '__builtins__', '__cached__',
'__copyright__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', '__version__', '_comparable_version', '_component_re',
'_default_architecture', '_follow_symlinks', '_ironpython26_sys_version_parser',
'_ironpython_sys_version_parser', '_java_getprop', '_libc_search', '_mac_ver_xml',
'_node', '_norm_version', '_platform', '_platform_cache',
'_pypy_sys_version_parser', '_sys_version', '_sys_version_cache',
'_sys_version_parser', '_syscmd_file', '_syscmd_uname', '_syscmd_ver',
'_uname_cache', '_ver_output', '_ver_stages', 'architecture', 'collections',
```

```
'java_ver', 'libc_ver', 'mac_ver', 'machine', 'node', 'os', 'platform',
'processor', 'python_branch', 'python_build', 'python_compiler',
'python_implementation', 'python_revision', 'python_version',
'python_version_tuple', 're', 'release', 'sys', 'system', 'system_alias', 'uname',
'uname_result', 'version', 'win32_edition', 'win32_is_iot', 'win32_ver']
```

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.
The module named Code356 has one function and one dictionary:

Code356:

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

person1 = {
  "name": "Aman",
  "age": 22,
  "country": "India"
}
```

Import only the person1 dictionary from the module:

Code357:

```
from Code356 import person1
from Code356 import greeting

print(person1["name"])
print(greeting("Aman"))
```

Output:

```
Aman
Hello,Aman
```

Note: When importing using the from keyword, do not use the module name when referring to elements in the module.
Example: person1["age"], not Code356.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.

Import the datetime module and display the current date:

Code358:

```
import datetime

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

Output:

```
2025-04-13 19:58:22.324662
```

Date Output

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

2025-04-13 19:47:21.416505

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:

Return the year and name of weekday:

Code359:

```
import datetime

x = datetime.datetime.now()
print(x.year)
print(x.strftime("%A"))
```

Output:

```
2025
Sunday
```

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.

Create a date object:

Code360:

```
import datetime

x = datetime.datetime(2025, 4, 21)
print(x)
```

Output:

```
2025-04-21 00:00:00
```

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:

Display the name of the month:

Code361:

```
import datetime

x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

Output:

```
June
```

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
%C	Century	20
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%
%G	ISO 8601 year	2018
%u	ISO 8601 weekday (1-7)	1
%V	ISO 8601 weeknumber (01-53)	01

Python Math

Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

Built-in Math Functions

The min() and max() functions can be used to find the lowest or highest value in an iterable:

Code362:

```
x = min(5, 10, 25)
y = max(5, 10, 25)

print(x)
print(y)
```

Output:

```
5
25
```

The abs() function returns the absolute (positive) value of the specified number:

Code363:

```
x = abs(-7.25)

print(x)
```

Output:

```
7.25
```

The pow(x, y) function returns the value of x to the power of y (xy).

Return the value of 4 to the power of 3 (same as 4 * 4 * 4):

Code364:

```
x = pow(4, 3)

print(x)
```

Output:

```
64
```

The Math Module

Python has also a built-in module called math, which extends the list of mathematical functions.

To use it, you must import the math module:

```
import math
```

When you have imported the math module, you can start using methods and constants of the module.

The math.sqrt() method for example, returns the square root of a number:

Code365:

```
import math

x = math.sqrt(64)
```

```
print(x)
```

Output:

```
8.0
```

The math.ceil() method rounds a number upwards to its nearest integer, and the math.floor() method rounds a number downwards to its nearest integer, and returns the result:

Code366:

```
#Import math library
import math

#Round a number upward to its nearest integer
x = math.ceil(1.4)

#Round a number downward to its nearest integer
y = math.floor(1.4)

print(x)
print(y)
```

Output:

```
2
1
```

The math.pi constant, returns the value of PI (3.14...):

Code367:

```
import math

x = math.pi
print(x)
```

Output:

```
3.141592653589793
```

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.

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.

Convert from JSON to Python:

Code368:

```
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"])
```

Output:

```
30
```

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.

Convert from Python to JSON:

Code369:

```
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)
```

Output:

```
{"name": "John", "age": 30, "city": "New York"}
```

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

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

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

Code370:

```
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))
```

Output:

```
{"name": "John", "age": 30}
["apple", "bananas"]
["apple", "bananas"]
"hello"
42
31.76
true
false
null
```

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

Convert a Python object containing all the legal data types:

Code371:

```
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}
  ]
}

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

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

Output:

```
{"name": "John", "age": 30, "married": true, "divorced": false, "children": ["Ann", "Billy"], "pets": null, "cars": [{"model": "BMW 230", "mpg": 27.5}, {"model": "Ford Edge", "mpg": 24.1}]}
```

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:

Use the indent parameter to define the numbers of indents:

Code372:

```
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}
  ]
```

```
}
# use four indents to make it easier to read the result:
print(json.dumps(x, indent=4))
```

Output:

```
{
    "name": "John",
    "age": 30,
    "married": true,
    "divorced": false,
    "children": [
        "Ann",
        "Billy"
    ],
    "pets": null,
    "cars": [
        {
            "model": "BMW 230",
            "mpg": 27.5
        },
        {
            "model": "Ford Edge",
            "mpg": 24.1
        }
    ]
}
```

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:

Use the separators parameter to change the default separator:

Code373:

```
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}
  ]
}
# use . and a space to separate objects, and a space, a = and a space to separate keys
from their values:
print(json.dumps(x, indent=4, separators=(". ", " = ")))
```

Output:

```
{
    "name" = "John".
    "age" = 30.
    "married" = true.
    "divorced" = false.
    "children" = [
        "Ann".
        "Billy"
    ].
    "pets" = null.
    "cars" = [
        {
            "model" = "BMW 230".
            "mpg" = 27.5
        }.
        {
            "model" = "Ford Edge".
            "mpg" = 24.1
        }
    ]
}
```

Order the Result

The json.dumps() method has parameters to order the keys in the result:
Use the sort_keys parameter to specify if the result should be sorted or not:

Code374:

```
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}
  ]
}

# sort the result alphabetically by keys:
print(json.dumps(x, indent=4, sort_keys=True))
```

Output:

```
{
    "age": 30,
    "cars": [
        {
```

```
            "model": "BMW 230",
            "mpg": 27.5
        },
        {
            "model": "Ford Edge",
            "mpg": 24.1
        }
    ],
    "children": [
        "Ann",
        "Billy"
    ],
    "divorced": false,
    "married": true,
    "name": "John",
    "pets": null
}
```

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

Code375:

```
import re

#Check if the string starts with "The" and ends with "Spain":

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

if x:
  print("YES! We have a match!")
else:
  print("No match")
```

Output:

```
YES! We have a match!
```

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:

Character	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	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"
()	Capture and group	

Flags

You can add flags to the pattern when using regular expressions.

Flag	Shorthand	Description
re.ASCII	re.A	Returns only ASCII matches
re.DEBUG		Returns debug information
re.DOTALL	re.S	Makes the . character match all characters (including newline character)
re.IGNORECASE	re.I	Case-insensitive matching
re.MULTILINE	re.M	Returns only matches at the beginning of each line
re.NOFLAG		Specifies that no flag is set for this pattern
re.UNICODE	re.U	Returns Unicode matches. This is default from Python 3. For Python 2: use this flag to return only Unicode matches
re.VERBOSEX	re.X	Allows whitespaces and comments inside patterns. Makes the pattern more readable

Special Sequences

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

Character	Description	Example

\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

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) is 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.

Print a list of all matches:

Code376:

```
import re

#Return a list containing every occurrence of "ai":
txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

Output:

```
['ai', 'ai']
```

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

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

Return an empty list if no match was found:

Code377:

```
import re

txt = "The rain in Spain"

#Check if "Portugal" is in the string:
x = re.findall("Portugal", txt)
print(x)
if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

Output:

```
[]
No match
```

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:

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

Code378:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)
print("The first white-space character is located in position:", x.start())
```

Output:

```
The first white-space character is located in position: 3
```

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

Make a search that returns no match:

Code379:

```
import re

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

Output:

```
None
```

The split() Function

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

Split at each white-space character:

Code380:

```
import re

#Split the string at every white-space character:
txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

Output:

```
['The', 'rain', 'in', 'Spain']
```

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

Split the string only at the first occurrence:

Code381:

```
import re

#Split the string at the first white-space character:
txt = "The rain in Spain"
x = re.split("\s", txt, 1)
print(x)
```

Output:

```
['The', 'rain in Spain']
```

The sub() Function

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

Replace every white-space character with the number 9:

Code382:

```
import re
```

```
#Replace all white-space characters with the digit "9":
txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

Output:

```
The9rain9in9Spain
```

You can control the number of replacements by specifying the count parameter:
Replace the first 2 occurrences:

Code383:

```
import re

#Replace the first two occurrences of a white-space character with the digit 9:

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

Output:

```
The9rain9in Spain
```

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.
Do a search that will return a Match Object:

Code384:

```
import re

#The search() function returns a Match object:
txt = "The rain in Spain"
x = re.search("ai", txt)
print(x)
```

Output:

```
<re.Match object; span=(5, 7), match='ai'>
```

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
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":

Code385:

```
import re

#Search for an upper case "S" character in the beginning of a word, and print its
position:
```

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

Output:

```
(12, 17)
```

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":

Code386:

```
import re

#Search for an upper case "S" character in the beginning of a word, and print the word:
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

Output:

```
Spain
```

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:

Check PIP version:

```
PS C:\Users\Aman Tripathi\AppData\Local\Programs\Python\python38\scripts> pip -version
pip 25.0.1 from c:\users\aman tripathi\appdata\local\programs\python\python38\lib\site-packages\pip (python 3.8)
```

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:

Download a package named "camelcase":

```
PS C:\Users\Aman Tripathi\AppData\Local\Programs\Python\python38\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.

Import and use "camelcase":

Code387:

```
import camelcase

c = camelcase.CamelCase()
txt = "lorem ipsum dolor sit amet"
print(c.hump(txt))
#This method capitalizes the first letter of each word.
```

Output:

```
Lorem Ipsum Dolor Sit Amet
```

Find Packages

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

Remove a Package

Use the uninstall command to remove a package:

Uninstall the package named "camelcase":

```
PS C:\Users\Aman Tripathi\appdata\local\programs\python\python38\Scripts> pip
uninstall camelCase
```

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

```
Found existing installation: camelcase 0.2
Uninstalling camelcase-0.2:
  Would remove:
    c:\users\aman tripathi\appdata\local\programs\python\python38\lib\site-
packages\camelcase-0.2.dist-info\*
    c:\users\aman tripathi\appdata\local\programs\python\python38\lib\site-
packages\camelcase\*
Proceed (Y/n)? Y
  Successfully uninstalled camelcase-0.2
```

List Packages

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

List installed packages:

```
PS C:\Users\Aman Tripathi\appdata\local\programs\python\python38\Scripts> pip list
```

Result:

```
Package            Version
------------------ -------
asgiref            3.8.1
backports.zoneinfo 0.2.1
Django             4.2.16
pip                25.0.1
setuptools         49.2.1
sqlparse           0.5.2
typing_extensions  4.12.2
tzdata             2024.2
```

Python Try Except

The try block lets you test a block of code for errors.
The except block lets you handle the error.
The else block lets you execute code when there is no 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:
The try block will generate an exception, because x is not defined:

Code388:

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

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

Output:

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

Code389:

```
#This will raise an exception, because x is not defined:

print(x)
```

Output:

```
Traceback (most recent call last):
  File "Code389.py", line 3, in <module>
    print(x)
NameError: name 'x' is not defined
```

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

Code390:

```
#The try block will generate a NameError, because x is not defined:

try:
  print(x)
except NameError:
```

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

Output:

```
Variable x is not defined
```

Built-in Exceptions

The table below shows built-in exceptions that are usually raised in Python:

Exception	Description
ArithmeticError	Raised when an error occurs in numeric calculations
AssertionError	Raised when an assert statement fails
AttributeError	Raised when attribute reference or assignment fails
Exception	Base class for all exceptions
EOFError	Raised when the input() method hits an "end of file" condition (EOF)
FloatingPointError	Raised when a floating point calculation fails
GeneratorExit	Raised when a generator is closed (with the close() method)
ImportError	Raised when an imported module does not exist
IndentationError	Raised when indentation is not correct
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
KeyboardInterrupt	Raised when the user presses Ctrl+c, Ctrl+z or Delete
LookupError	Raised when errors raised cant be found
MemoryError	Raised when a program runs out of memory
NameError	Raised when a variable does not exist
NotImplementedError	Raised when an abstract method requires an inherited class to override the method
OSError	Raised when a system related operation causes an error
OverflowError	Raised when the result of a numeric calculation is too large
ReferenceError	Raised when a weak reference object does not exist
RuntimeError	Raised when an error occurs that do not belong to any specific exceptions
StopIteration	Raised when the next() method of an iterator has no further values
SyntaxError	Raised when a syntax error occurs
TabError	Raised when indentation consists of tabs or spaces
SystemError	Raised when a system error occurs
SystemExit	Raised when the sys.exit() function is called
TypeError	Raised when two different types are combined

UnboundLocalError	Raised when a local variable is referenced before assignment
UnicodeError	Raised when a unicode problem occurs
UnicodeEncodeError	Raised when a unicode encoding problem occurs
UnicodeDecodeError	Raised when a unicode decoding problem occurs
UnicodeTranslateError	Raised when a unicode translation problem occurs
ValueError	Raised when there is a wrong value in a specified data type
ZeroDivisionError	Raised when the second operator in a division is zero

Else

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

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

Code391:

```
#The try block does not raise any errors, so the else block is executed:

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

Output:

```
Hello
Nothing went wrong
```

Finally

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

Code392:

```
#The finally block gets executed no matter if the try block raises any errors or not:

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

Output:

```
Something went wrong
The 'try except' is finished
```

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

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

Code393:

```
#The try block will raise an error when trying to write to a read-only file:

try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

Output:

```
Something went wrong when opening the file
```

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

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the raise keyword.

Raise an error and stop the program if x is lower than 0:

Code394:

```
x = -1

if x < 0:
    raise Exception("Sorry, no numbers below zero")
```

Output:

```
Traceback (most recent call last):
  File "Code394.py", line 4, in <module>
    raise Exception("Sorry, no numbers below zero")
Exception: Sorry, no numbers below zero
```

The raise keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Raise a TypeError if x is not an integer:

Code395:

```
x = "hello"
if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

Output:

```
Traceback (most recent call last):
  File "Code395.py", line 4, in <module>
    raise TypeError("Only integers are allowed")
TypeError: Only integers are allowed
```

Python User Input

User Input

Python allows for user 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 username, and when you entered the username, it gets printed on the screen:

Python 3.6

Code396:

```
username = input("Enter username:")
print("Username is: " + username)
print(type(username))
```

Output:

```
Enter username:Aman Kumar Tripathi
Username is: Aman Kumar Tripathi
<class 'str'>
```

Python 2.7

Code397:

```
username = raw_input("Enter username:")
print("Username is: " + username)
```

Output:

```
Enter username:Aman
Username is: Aman
```

Python stops executing when it comes to the input() function, and continues when the user has given some input.

Python Match

The match statement is used to perform different actions based on different conditions.

The Python Match Statement

Instead of writing many if..else statements, you can use the match statement.

The match statement selects one of many code blocks to be executed.

Syntax

```
match expression:
  case x:
    code block
  case y:
    code block
  case z:
    code block
```

This is how it works:

- The match expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.

The example below uses the weekday number to print the weekday name:

Code398:

```
day = 4
match day:
  case 1:
    print("Monday")
  case 2:
    print("Tuesday")
  case 3:
    print("Wednesday")
  case 4:
    print("Thursday")
  case 5:
    print("Friday")
  case 6:
    print("Saturday")
  case 7:
    print("Sunday")
```

Output:

```
Thursday
```

Default Value

Use the underscore character _ as the last case value if you want a code block to execute when there are not other matches:

Code399:

```
day = 4
match day:
  case 6:
```

```
    print("Today is Saturday")
  case 7:
    print("Today is Sunday")
  case _:
    print("Looking forward to the Weekend")
```

Output:

```
Looking forward to the Weekend
```

The value _ will always match, so it is important to place it as the last case to make it beahave as a default case.

Combine Values

Use the pipe character | as an or operator in the case evaluation to check for more than one value match in one case:

Code400:

```
day = 4
match day:
  case 1 | 2 | 3 | 4 | 5:
    print("Today is a weekday")
  case 6 | 7:
    print("I love weekends!")
```

Output:

```
Today is a weekday
```

If Statements as Guards

You can add if statements in the case evaluation as an extra condition-check:

Code401:

```
month = 5
day = 4
match day:
  case 1 | 2 | 3 | 4 | 5 if month == 4:
    print("A weekday in April")
  case 1 | 2 | 3 | 4 | 5 if month == 5:
    print("A weekday in May")
  case _:
    print("No match")
```

Output:

```
A weekday in May
```

Python File Handling

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:

Code402:

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

print(f.read())
```

Output:

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

If the file is located in a different location, you will have to specify the file path, like this:

Open a file on a different location:

Code403:

```
f = open("C:\\Users\Aman Tripathi\Desktop\welcome.txt", "r")

print(f.read())
```

Output:

```
Hii my name is Aman
```

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:

Return the 5 first characters of the file:

Code404:

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

print(f.read(5))
```

Output:

```
Hello
```

Read Lines

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

Read one line of the file:

Code405:

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

```
print(f.readline())
```

Output:

```
Hello! Welcome to demofile.txt
```

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

Read two lines of the file:

Code406:

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

print(f.readline())
print(f.readline())
```

Output:

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

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

Loop through the file line by line:

Code407:

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

Output:

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

Close Files

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

Close the file when you are finished with it:

Code408:

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

print(f.readline())

f.close()
```

Output:

```
Hello! Welcome to demofile.txt
```

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

Code409:

```
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())
```

Output:

```
This is demofile2,Now the file has more content!
```

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

Code410:

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

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

Output:

```
Woops! I have deleted the content!
```

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 exists
"a" - Append - will create a file if the specified file does not exists
"w" - Write - will create a file if the specified file does not exists
Create a file called "myfile.txt":

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

Result: a new empty file is created!

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:

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:

Check if file exists, then delete it:

Code411:

```
import os
if os.path.exists("demofile4.txt"):
  os.remove("demofile4.txt")
else:
  print("The file does not exist")
```

Delete Folder

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

Remove the folder "myfolder":

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

Note: You can only remove empty folders.

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.

Or get started right away with a MongoDB cloud service at https://www.mongodb.com/cloud/atlas.

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:

Code412:

```
import pymongo

#if this page is executed with no errors, you have the "pymongo" module installed.
```

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.

Create a database called "mydatabase":

Code413:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase']
# database created!
```

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:

Return a list of your system's databases:

Code414:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
print(myclient.list_database_names())
```

Output:

```
['MongoDBProject_Database', 'admin', 'config', 'local', 'myschoolDB']
```

Or you can check a specific database by name:

Check if "mydatabase" exists:

Code415:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
dblist = myclient.list_database_names()
if "mydatabase" in dblist:
  print("The database exists.")
else:
  print("The database does not exists")
```

Output:

```
The database does not 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.

Create a collection called "customers":

Code416:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase']
mycol = mydb["customers"]
# collection created!
```

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:

Return a list of all collections in your database:

Code417:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase']
mycol = mydb["customers"]
print(mydb.list_collection_names())
```

Output:

```
[]
```

Or you can check a specific collection by name:

Check if the "customers" collection exists:

Code418:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase']
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.

Insert a record in the "customers" collection:

Code419:

```
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)
print(x)
```

Output:

```
InsertOneResult(ObjectId('6802329b0124ef57ac2ee334'), acknowledged=True)
```

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.

Insert another record in the "customers" collection, and return the value of the _id field:

Code420:

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase']
mycol = mydb["customers"]
mydict = { "name": "Peter", "address": "Lowstreet 27" }
x = mycol.insert_one(mydict)
print(x.inserted_id)
```

Output:

```
680264c07f573399325e02ef
```

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:

Code421:

```
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)
```

Output:

```
[ObjectId('68028de2790cf405f5613c37'), ObjectId('68028de2790cf405f5613c38'),
ObjectId('68028de2790cf405f5613c39'), ObjectId('68028de2790cf405f5613c3a'),
ObjectId('68028de2790cf405f5613c3b'), ObjectId('68028de2790cf405f5613c3c'),
ObjectId('68028de2790cf405f5613c3d'), ObjectId('68028de2790cf405f5613c3e'),
ObjectId('68028de2790cf405f5613c3f'), ObjectId('68028de2790cf405f5613c40'),
ObjectId('68028de2790cf405f5613c41'), ObjectId('68028de2790cf405f5613c42')]
```

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 your 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.

Code422:

```
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 a list of the _id values of the inserted documents:
print(x.inserted_ids)
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Python MongoDB Find

In MongoDB we use the find() and find_one() 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.
Find the first document in the customers collection:

Code423:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
x = mycol.find_one()
print(x)
```

Output:

```
{'_id': ObjectId('6802329b0124ef57ac2ee334'), 'name': 'John', 'address': 'Highway 37'}
```

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.
Return all documents in the "customers" collection, and print each document:

Code424:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
for x in mycol.find():
  print(x)
```

Output:

```
{'_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'}
```

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.

Return only the names and addresses, not the _ids:

Code425:

```
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)
```

Output:

```
{'name': 'John', 'address': 'Highway 37'}
{'name': 'Peter', 'address': 'Lowstreet 27'}
{'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'}
```

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:

This example will exclude "address" from the result:

Code426:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
for x in mycol.find({},{ "address": 0 }):
  print(x)
```

Output:

```
{'_id': 1, 'name': 'John'}
{'_id': 2, 'name': 'Peter'}
{'_id': 3, 'name': 'Amy'}
{'_id': 4, 'name': 'Hannah'}
{'_id': 5, 'name': 'Michael'}
```

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):

Code427:

```
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.
Find document(s) with the address "Park Lane 38":

Code428:

```
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)
```

Output:

```
{'_id': 11, 'name': 'Ben', 'address': 'Park Lane 38'}
```

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"}:
Find documents where the address starts with the letter "S" or higher:

Code429:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
#address greater than S:
myquery = { "address": {"$gt": "S"} }
mydoc = mycol.find(myquery)
for x in mydoc:
  print(x)
```

Output:

```
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 10, 'name': 'Vicky', 'address': 'Yellow Garden 2'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
```

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"}:
Find documents where the address starts with the letter "S":

Code430:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
#address starts with S:
myquery = { "address": { "$regex": "^S" } }
mydoc = mycol.find(myquery)
for x in mydoc:
  print(x)
```

Output:

```
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
```

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).
Sort the result alphabetically by name:

Code431:

```
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)
```

Output:

```
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_id': 11, 'name': 'Ben', 'address': 'Park Lane 38'}
{'_id': 7, 'name': 'Betty', 'address': 'Green Grass 1'}
{'_id': 13, 'name': 'Chuck', 'address': 'Main Road 989'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 1, 'name': 'John', 'address': 'Highway 37'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 6, 'name': 'Sandy', 'address': 'Ocean blvd 2'}
{'_id': 9, 'name': 'Susan', 'address': 'One way 98'}
{'_id': 10, 'name': 'Vicky', 'address': 'Yellow Garden 2'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
{'_id': 12, 'name': 'William', 'address': 'Central st 954'}
```

Sort Descending

Use the value -1 as the second parameter to sort descending.
sort("name", 1) #ascending
sort("name", -1) #descending
Sort the result reverse alphabetically by name:

Code432:

```
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)
```

Output:

```
{'_id': 12, 'name': 'William', 'address': 'Central st 954'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
{'_id': 10, 'name': 'Vicky', 'address': 'Yellow Garden 2'}
{'_id': 9, 'name': 'Susan', 'address': 'One way 98'}
{'_id': 6, 'name': 'Sandy', 'address': 'Ocean blvd 2'}
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
{'_id': 1, 'name': 'John', 'address': 'Highway 37'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 13, 'name': 'Chuck', 'address': 'Main Road 989'}
{'_id': 7, 'name': 'Betty', 'address': 'Green Grass 1'}
{'_id': 11, 'name': 'Ben', 'address': 'Park Lane 38'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
```

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.

Delete the document with the address "Mountain 21":

Code433:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
myquery = { "address": "Mountain 21" }
mycol.delete_one(myquery)
#print the customers collection after the deletion:
for x in mycol.find():
  print(x)
```

Output:

```
{'_id': 1, 'name': 'John', 'address': 'Highway 37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_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'}
```

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.

Delete all documents were the address starts with the letter S:

Code434:

```
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")
```

Output:

```
2 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:

Delete all documents in the "customers" collection:

Code435:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
x = mycol.delete_many({})
print(x.deleted_count, "documents deleted")
```

Output:

```
11 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.

Delete the "customers" collection:

Code436:

```
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.

Change the address from "Valley 345" to "Canyon 123":

Code437:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
myquery = { "name": "John" }
newvalues = { "$set": { "name": "Aman" } }
mycol.update_one(myquery, newvalues)
#print "customers" after the update:
for x in mycol.find({"name":"Aman"}):
  print(x)
```

Output:

```
{'_id': 1, 'name': 'Aman', 'address': 'Highway 37'}
```

Update Many

To update all documents that meets the criteria of the query, use the update_many() method.

Update all documents where the address starts with the letter "S":

Code438:

```
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.")
```

Output:

```
2 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'}

Limit the result to only return 5 documents:

Code439:

```
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)
```

Output:

```
{'_id': 1, 'name': 'Aman', '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': 'Canyon 123'}
```

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 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":

```
PS C:\Users\Aman Tripathi\Appdata\local\programs\python\python312\scripts> python -m pip install mysql-connector-python
```

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:

Code440:

```
import mysql.connector

#if this page is executed with no errors, you have the "mysql.connector" module installed.
```

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:

Code441:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321"
)
print(mydb)
```

Python MySQL Create Database

Creating a Database

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

Code442:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE DATABASE mydatabase")
#If this page is executed with no error, 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:

Return a list of your system's databases:

Code443:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321"
)
mycursor = mydb.cursor()
mycursor.execute("SHOW DATABASES")
for x in mycursor:
  print(x)
```

Output:

```
('amandb',)
('information_schema',)
('mydatabase',)
('mysql',)
('performance_schema',)
('sys',)
```

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

Try connecting to the database "mydatabase":

Code444:

```
import mysql.connector

mydb = mysql.connector.connect(
```

```
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
#If this page is executed with no error, the database "mydatabase" exists in your system
```

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

Create a table named "customers":

Code445:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))")
#If this page is executed with no error, you have successfully created a table named
"customers".
```

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:

Return a list of your system's databases:

Code446:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SHOW TABLES")
for x in mycursor:
     print(x)
```

Output:

```
('customers',)
```

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.

Create primary key when creating the table:

Code447:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY, name
VARCHAR(255), address VARCHAR(255))")
#If this page is executed with no error, the table "customers" now has a primary key
```

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

Create primary key on an existing table:

Code448:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("ALTER TABLE customers ADD COLUMN id INT AUTO_INCREMENT PRIMARY KEY")
#If this page is executed with no error, the table "customers" now has a primary key
```

Python MySQL Insert Into Table

Insert Into Table

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

Insert a record in the "customers" table:

Code449:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  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.")
```

Output:

```
1 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:

Fill the "customers" table with data:

Code450:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  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'),
```

```
]
mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "record was inserted.")
```

Output:

```
5 record 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 than one row, the id of the last inserted row is returned.

Insert one row, and return the ID:

Code451:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  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)
```

Output:

```
1 record inserted, ID: 7
```

Python MySQL Select From

Select From a Table

To select from a table in MySQL, use the "SELECT" statement:

Select all records from the "customers" table, and display the result:

Code452:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('John', 'Highway 21', 1)
('Peter', 'Lowstreet 4', 2)
('Amy', 'Apple st 652', 3)
('Hannah', 'Mountain 21', 4)
('Michael', 'Valley 345', 5)
('Sandy', 'Ocean blvd 2', 6)
('Michelle', 'Blue Village', 7)
```

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):

Select only the name and address columns:

Code453:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT name, address FROM customers")
myresult = mycursor.fetchall()
for x in myresult:
```

```
  print(x)
```

Output:

```
('John', 'Highway 21')
('Peter', 'Lowstreet 4')
('Amy', 'Apple st 652')
('Hannah', 'Mountain 21')
('Michael', 'Valley 345')
('Sandy', 'Ocean blvd 2')
('Michelle', 'Blue Village')
```

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:
Fetch only one row:

Code454:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchone()
print(myresult)
```

Output:

```
('John', 'Highway 21', 1)
```

Python MySQL Where

Select With a Filter

When selecting records from a table, you can filter the selection by using the "WHERE" statement:

Select record(s) where the address is "Mountain 21": result:

Code455:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "SELECT * FROM customers WHERE address = 'Mountain 21'"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('Hannah', 'Mountain 21', 4)
```

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:

Select records where the address contains the word "way":

Code456:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  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)
```

Output:

```
('John', 'Highway 21', 1)
```

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:

Escape query values by using the placholder %s method:

Code457:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Lowstreet 4", )
mycursor.execute(sql, adr)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('Peter', 'Lowstreet 4', 2)
```

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.
Sort the result alphabetically by name: result:

Code458:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "SELECT * FROM customers ORDER BY name"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('Amy', 'Apple st 652', 3)
('Hannah', 'Mountain 21', 4)
('John', 'Highway 21', 1)
('Michael', 'Valley 345', 5)
('Michelle', 'Blue Village', 7)
('Peter', 'Lowstreet 4', 2)
('Sandy', 'Ocean blvd 2', 6)
```

ORDER BY DESC

Use the DESC keyword to sort the result in a descending order.
Sort the result reverse alphabetically by name:

Code459:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  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)
```

Output:

```
('Sandy', 'Ocean blvd 2', 6)
('Peter', 'Lowstreet 4', 2)
('Michelle', 'Blue Village', 7)
('Michael', 'Valley 345', 5)
('John', 'Highway 21', 1)
('Hannah', 'Mountain 21', 4)
('Amy', 'Apple st 652', 3)
```

Python MySQL Delete From By

Delete Record

You can delete records from an existing table by using the "DELETE FROM" statement:
Delete any record where the address is "Mountain 21":

Code460:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "DELETE FROM customers WHERE address = 'Mountain 21'"
mycursor.execute(sql)
mydb.commit()
print(mycursor.rowcount, "record(s) deleted")
```

Output:

```
1 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:

Escape values by using the placeholder %s method:

Code461:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "DELETE FROM customers WHERE address = %s"
adr = ("Lowstreet 4", )
```

```
mycursor.execute(sql, adr)
mydb.commit()
print(mycursor.rowcount, "record(s) deleted")
```

Output:

```
1 record(s) deleted
```

Python MySQL Drop Table

Delete a Table

You can delete an existing table by using the "DROP TABLE" statement:

Delete the table "customers":

Code462:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "DROP TABLE customers"
mycursor.execute(sql)
#If this page was executed with no error(s), you have successfully deleted the "customers"
table.
```

Drop Only if Exist

If 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.

Delete the table "customers" if it exists:

Code463:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "DROP TABLE IF EXISTS customers"
mycursor.execute(sql)
#If this page was executed with no error(s), you have successfully deleted the "customers"
table.
```

Python MySQL Update Table

Update Table

You can update existing records in a table by using the "UPDATE" statement:
Overwrite the address column from "Valley 345" to "Canyon 123":

Code464:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  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")
```

Output:

```
1 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 update statement:

Escape values by using the placeholder %s method:

Code465:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  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")
```

Output:

```
1 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:

Select the 5 first records in the "customers" table:

Code466:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers LIMIT 2")
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('Peter', 'Lowstreet 4')
('Amy', 'Apple st 652')
```

Start From Another Position

If you want to return five records, starting from the third record, you can use the "OFFSET" keyword:

Start from position 3, and return 3 records:

Code467:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers LIMIT 3 OFFSET 2")
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('Hannah', 'Mountain 21')
('Michael', 'Valley 345')
('Sandy', 'Ocean blvd 2')
```

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:}
```

Code468:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE person (name VARCHAR(255), fav VARCHAR(255))")
sql = "INSERT INTO person (name, fav) VALUES (%s, %s)"
val = [
 ('John', '154'),
 ('Peter', '154'),
 ('Amy', '155'),
 ('Hannah', ''),
 ('Michael', ''),
]
mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "record was inserted.")
```

Output:

```
5 record was inserted.
```

```
Products
{ id: 154, name: 'Chocolate Heaven' },
{ id: 155, name: 'Tasty Lemons' },
{ id: 156, name: 'Vanilla Dreams' }
```

Code469:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
```

```
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE product (name VARCHAR(255), id VARCHAR(255))")
sql = "INSERT INTO product (name, id) VALUES (%s, %s)"
val = [
 ('Chocolate Heaven', '154'),
 ('Tasty Lemons', '155'),
 ('Vanilla Dreams', '156'),
]
mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "record was inserted.")
```

Output:

```
3 record was inserted.
```

These two tables can be combined by using users' fav field and products' id field.
Join users and products to see the name of the users favorite product:

Code470:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "SELECT \
  person.name AS person, \
  product.name AS favorite \
  FROM person \
  INNER JOIN product ON person.fav = product.id"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('John', 'Chocolate Heaven')
('Peter', 'Chocolate Heaven')
('Amy', 'Tasty Lemons')
```

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:

Select all users and their favorite product:

Code471:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "SELECT \
  person.name AS person, \
  product.name AS favorite \
  FROM person \
  LEFT JOIN product ON person.fav = product.id"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('John', 'Chocolate Heaven')
('Peter', 'Chocolate Heaven')
('Amy', 'Tasty Lemons')
('Hannah', None)
('Michael', None)
```

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:

Select all products, and the user(s) who have them as their favorite:

Code472:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="Aman@321",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "SELECT \
  person.name AS person, \
  product.name AS favorite \
  FROM person \
  RIGHT JOIN product ON person.fav = product.id"
```

```
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Output:

```
('Peter', 'Chocolate Heaven')
('John', 'Chocolate Heaven')
('Amy', 'Tasty Lemons')
(None, 'Vanilla Dreams')
```

Note: Hannah and Michael, who have no favorite product, are not included in the result.