## 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.

### Example for creating class Gef your own Python Server

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

class MyClass:  
 x = 5

### Example for creating object

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

p1 = MyClass()  
print(p1.x)

Array Implementation

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"

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

### Example

Get the value of the first array item:

x = cars[0]

### Example

Modify the value of the first array item:

cars[0] = "Toyota"

## The Length of an Array

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

### Example

x = len(cars)

## Looping Array Elements

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

### Example

Print each item in the cars array:

for x in cars:  
 print(x)

## Adding Array Elements

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

### Example

Add one more element to the cars array:

cars.append("Honda")

## Removing Array Elements

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

### Example

Delete the second element of the cars array:

cars.pop(1)

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

### Example

Delete the element that has the value "Volvo":

cars.remove("Volvo")

File Methods

## 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

## 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")

demofile.txt

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

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

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

### Example

f = open("demofile.txt")  
print(f.read())

If the file is located in a different location, you will have to specify the file path, like this:

### Example

Open a file on a different location:

f = open("D:\\myfiles\welcome.txt")  
print(f.read())

## Using the with statement

You can also use the with statement when opening a file:

### Example

Using the with keyword:

with open("demofile.txt") as f:  
 print(f.read())

## Close Files

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

If you are not using the with statement, you must write a close statement in order to close the file:

### Example

Close the file when you are finished with it:

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

## Read Only Parts of the File

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

### Example

Return the 5 first characters of the file:

with open("demofile.txt") as f:  
 print(f.read(**5**))

## Read Lines

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

### Example

Read one line of the file:

with open("demofile.txt") as f:  
 print(f.readline())

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

### Example

Read two lines of the file:

with open("demofile.txt") as f:  
 print(f.readline())  
 print(f.readline())

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

### Example

Loop through the file line by line:

with open("demofile.txt") as f:  
 for x in f:  
 print(x)

## Write to an Existing File

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

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

"w" - Write - will overwrite any existing content

### Example

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

with open("demofile.txt", "a") as f:  
 f.write("Now the file has more content!")  
  
#open and read the file after the appending:  
with open("demofile.txt") as f:  
 print(f.read())

## Overwrite Existing Content

To overwrite the existing content to the file, use the w parameter:

### Example

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

with open("demofile.txt", "w") as f:  
 f.write("Woops! I have deleted the content!")  
  
#open and read the file after the overwriting:  
with open("demofile.txt") as f:  
 print(f.read())

## 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

### Example

Create a new file called "myfile.txt":

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

## Delete a File

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

### Example

Remove the file "demofile.txt":

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

## Check if File exist:

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

### Example

Check if file exists, *then* delete it:

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

## Delete Folder

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

### Example

Remove the folder "myfolder":

import os  
os.rmdir("myfolder")

Keywords and Identifiers

#### Keywords

Keywords are reserved words in Python that have special meanings and cannot be used for variable names or other identifiers. They help define the structure and behavior of Python programs.

### Example

# Using 'if' and 'else' keywords in a conditional statement  
x = 10  
  
if x > 5:  
 print("x is greater than 5")  
else:  
 print("x is 5 or less")

Some commonly used keywords include: and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, if, import, in, is, lambda, None, not, or, pass, return, True, try, while, with, yield.

#### Identifiers

Identifiers are the names used to identify variables, functions, classes, objects, and other entities in Python. They must follow certain rules:

* Must start with a letter (A-Z or a-z) or an underscore (\_).
* Can contain letters, digits (0-9), and underscores (\_).
* Cannot be a keyword.
* Are case-sensitive (Var and var are different).

Example

# Correct usage of identifiers

student\_name = "Alice" # Variable  
def calculate\_average(score1, score2): # Function name  
 return (score1 + score2) / 2  
  
print(calculate\_average(85, 90))

Tuples

#### Tuple

A **tuple** is an ordered, immutable collection in Python. It allows storing multiple elements in a single variable and is written with round brackets ().

Example of Creating a Tuple:

# Creating a tuple  
my\_tuple = ("apple", "banana", "cherry")  
print(my\_tuple)

Tuples are useful when you need a collection of items that shouldn't be modified after creation.

#### Tuple Methods

count(value) – Returns the number of times a specified value appears in the tuple.

my\_tuple = (1, 2, 2, 3, 4, 2)  
print(my\_tuple.count(2)) # Output: 3

index(value) – Returns the index of the first occurrence of a specified value.

my\_tuple = ("apple", "banana", "cherry", "banana")  
print(my\_tuple.index("banana")) # Output: 1

#### Tuple Properties

* **Ordered** – The elements maintain their sequence.
* **Immutable** – Cannot be changed after creation.
* **Supports Duplicates** – Allows repeated elements.

Sets

#### Set

A **set** is an unordered collection of unique elements in Python. Unlike lists and tuples, sets do not allow duplicate values and are defined using curly brackets {}.

### Example

# Creating a set  
my\_set = {"apple", "banana", "cherry"}  
print(my\_set)

Sets are useful for removing duplicates from data and performing mathematical operations like union and intersection.

#### Common Set Methods

add(value) – Adds an element to the set.

my\_set = {1, 2, 3}  
my\_set.add(4)  
print(my\_set) # Output: {1, 2, 3, 4}

remove(value) – Removes the specified element. Raises an error if the element does not exist.

my\_set = {1, 2, 3}  
my\_set.remove(2)  
print(my\_set) # Output: {1, 3}

discard(value) – Removes the specified element without raising an error if it does not exist.  
  
my\_set = {1, 2, 3}  
my\_set.discard(5) # No error  
print(my\_set) # Output: {1, 2, 3}

pop() – Removes and returns a random element from the set.  
  
my\_set = {1, 2, 3, 4}  
removed\_element = my\_set.pop()  
print(removed\_element) # Output: Random element  
print(my\_set) # Output: Remaining elements

clear() – Removes all elements from the set.  
  
my\_set = {1, 2, 3}  
my\_set.clear()  
print(my\_set) # Output: set()

union(set2) – Returns a new set containing all unique elements from both sets.  
  
set1 = {1, 2, 3}  
set2 = {3, 4, 5}  
result = set1.union(set2)  
print(result) # Output: {1, 2, 3, 4, 5}

intersection(set2) – Returns a new set with common elements from both sets.  
  
set1 = {1, 2, 3}  
set2 = {2, 3, 4}  
result = set1.intersection(set2)  
print(result) # Output: {2, 3}

difference(set2) – Returns a new set with elements from the first set that are not in the second set.  
  
set1 = {1, 2, 3}  
set2 = {2, 3, 4}  
result = set1.difference(set2)  
print(result) # Output: {1}

#### Set Properties

* **Unordered** – No guaranteed order of elements.
* **Unique Elements** – No duplicates allowed.
* **Mutable** – Can add or remove elements.
* **Optimized for Membership Testing** – Fast operations for checking if an element exists.

Different Modules

### Python Modules

Modules in Python are files containing Python code—usually functions, classes, or variables—that help organize and reuse code across different programs.

## Create a Module

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

### Example:

Save this code in a file named mymodule.py

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

## Use a Module

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

### Example

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

import mymodule  
  
mymodule.greeting("Jonathan")

## Variables in Module

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

### Example

Save this code in the file mymodule.py

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

### Example

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

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

## Naming a Module

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

## Re-naming a Module

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

### Example

Create an alias for mymodule called mx:

import mymodule as mx  
  
a = mx.person1["age"]  
print(a)

## Built-in Modules

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

### Example

Import and use the platform module:

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

## Using the dir() Function

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

### Example

List all the defined names belonging to the platform module:

import platform  
  
x = dir(platform)  
print(x)

## Import From Module

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

### Example

The module named mymodule has one function and one dictionary:

def greeting(name):  
 print("Hello, " + name)  
  
person1 = {  
 "name": "John",  
 "age": 36,  
 "country": "Norway"  
}

### Example

Import only the person1 dictionary from the module:

from mymodule import person1  
  
print (person1["age"])

#### Types of Modules

Python has built-in modules as well as user-defined ones.

1. **Built-in Modules** – Provided by Python for common tasks.  
   1. math: Mathematical operations
   2. random: Generating random numbers
   3. datetime: Working with dates and times
   4. os: Interacting with the operating system
   5. sys: System-specific parameters and functions
2. **User-Defined Modules** – Created by programmers to organize their own code.
3. **Third-Party-Modules** – Modules provided by external libraries, installed via pip.

Examples: numpy, pandas, matplotlib

Directory and File Management System

#### Overview

Python provides several modules to handle directories and files efficiently. The os and shutil modules allow users to create, modify, and delete files and directories.

#### Working with Directories

**Get Current Directory** – Use os.getcwd() to retrieve the current working directory.

import os  
print(os.getcwd()) # Output: Current directory path

**Change Directory** – Use os.chdir() to switch to a different directory.

os.chdir("C:/Users/Example") # Change to a specific directory  
print(os.getcwd()) # Verify the change

**List Files and Directories** – Use os.listdir() to list all files and subdirectories.

print(os.listdir())  
# Output: List of files and folders

**Create a New Directory** – Use os.mkdir() to create a new folder.

os.mkdir("new\_folder")   
# Creates a folder named 'new\_folder'

**Rename a Directory or File** – Use os.rename() to rename a file or folder.

os.rename("old\_name", "new\_name")   
# Renames 'old\_name' to 'new\_name'

**Remove a Directory** – Use os.rmdir() to delete an empty directory.

os.rmdir("new\_folder")   
# Deletes 'new\_folder'

#### Working with Files

**Create and Write to a File** – Use open() with write mode.

with open("example.txt", "w") as file:  
 file.write("Hello, Python!")

**Read a File** – Use open() with read mode.

with open("example.txt", "r") as file:  
 content = file.read()  
 print(content)

**Append to a File** – Use open() with append mode.

os.remove("example.txt") # Deletes 'example.txt'

**Delete a File** – Use os.remove() to delete a file.

os.remove("example.txt") # Deletes 'example.txt'

Dictionary

A **dictionary** in Python is an unordered, mutable collection that stores data in **key-value pairs**. It allows fast lookups and modifications using keys.

Example

# Creating a dictionary  
student\_info = {  
 "name": "Alice",  
 "age": 25,  
 "course": "Computer Science"  
}  
print(student\_info)

#### Dictionary Methods

keys() – Returns all keys in the dictionary.

print(student\_info.keys())   
# Output: dict\_keys(['name', 'age', 'course'])

values() – Returns all values in the dictionary.

print(student\_info.values())

# Output: dict\_values(['Alice', 25, 'Computer Science'])

items() – Returns key-value pairs as tuples.

print(student\_info.items())

# Output: dict\_items([('name', 'Alice'), ('age', 25), ('course', 'Computer Science')])

get(key, default) – Retrieves a value for a given key, with an optional default.

print(student\_info.get("name"))

# Output: Alice  
print(student\_info.get("gender", "Not specified"))

# Output: Not specified

update(dictionary) – Updates the dictionary with another dictionary.

student\_info.update({"gender": "Female"})  
print(student\_info)

# Output: {'name': 'Alice', 'age': 25, 'course': 'Computer Science', 'gender': 'Female'}

pop(key) – Removes and returns the value of the specified key.

age = student\_info.pop("age")  
print(age) # Output: 25  
print(student\_info) # Output: {'name': 'Alice', 'course': 'Computer Science', 'gender': 'Female'}

popitem() – Removes and returns the last inserted key-value pair.

last\_item = student\_info.popitem()  
print(last\_item) # Output: ('gender', 'Female')

clear() – Removes all items from the dictionary.

student\_info.clear()  
print(student\_info) # Output: {}

Strings

A **string** in Python is a sequence of characters enclosed in either single (') or double (") quotes. Strings are **immutable**, meaning they cannot be changed after creation.

Example

# Using single and double quotes

string1 = "Hello, World!"  
string2 = 'Python is fun!'  
print(string1)  
print(string2)

#### String Methods

Python provides several built-in methods to manipulate strings:

upper() – Converts all characters to uppercase.

text = "hello"  
print(text.upper())

lower() – Converts all characters to lowercase.

text = "HELLO"  
print(text.lower())

strip() – Removes leading and trailing spaces.

text = " Python "  
print(text.strip())

replace(old, new) – Replaces occurrences of a substring.

text = "I love Java"  
print(text.replace("Java", "Python"))

split(separator) – Splits a string into a list based on a separator.

text = "apple,banana,cherry"  
print(text.split(","))

join(iterable) – Joins elements of an iterable into a string.

words = ["Hello", "World"]  
print(" ".join(words))

find(substring) – Returns the index of the first occurrence of a substring.

text = "Python programming"  
print(text.find("programming"))

count(substring) – Counts occurrences of a substring.

text = "banana banana banana"  
print(text.count("banana"))

#### String Indexing and Slicing

Strings can be accessed using indexing and slicing.

Indexing

Example

text = "Python"  
print(text[0]) # Output: P  
print(text[-1]) # Output: n

Slicing

Example

text = "Python"  
print(text[0:3]) # Output: Pyt  
print(text[:4]) # Output: Pyth  
print(text[2:]) # Output: thon

#### String Formatting

Python provides multiple ways to format strings:

**Using** f-string

name = "Alice"  
age = 25  
print(f"My name is {name} and I am {age} years old.")

**Using** .format()

print("My name is {} and I am {} years old.".format(name, age))

**Using** % **formatting**

print("My name is %s and I am %d years old." % (name, age))

Data Type Conversion

Data type conversion in Python refers to changing the type of a variable from one data type to another. Python supports **implicit** and **explicit** type conversion.

#### Implicit Type Conversion

Python automatically converts one data type to another when necessary, without requiring explicit instructions.

Example

**Converting** int **to** float

integer\_number = 10  
float\_number = 2.5  
  
result = integer\_number + float\_number

# Python converts int to float automatically

print(result) # Output: 12.5  
print(type(result)) # Output: <class 'float'>

#### Explicit Type Conversion (Type Casting)

Explicit conversion requires manually changing the data type using built-in functions like int(), float(), str(), etc.

Example**:**

**Converting** str **to** int

num\_string = "15"

num\_integer = int(num\_string)

# Explicit conversion using int()

print(num\_integer) # Output: 15  
print(type(num\_integer)) # Output: <class 'int'>

Numbers

There are three numeric types in Python:

* int
* float
* complex

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

### Example[Get](https://www.w3schools.com/python/python_server.asp)

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

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

### Example

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

## Int

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

### Example

Integers:

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

## Float

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

### Example

Floats:

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

## Complex

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

### Example

Complex:

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

Namespace and scope

#### Namespace

A **namespace** in Python is a mapping between names and objects. It acts like a dictionary where variable names (keys) are associated with their respective objects (values). Python maintains multiple namespaces to prevent naming conflicts.

#### Types of Namespaces

1. **Built-in Namespace** – Contains built-in functions and exceptions (print(), len(), Exception).
2. **Global Namespace** – Stores variables and functions defined at the top level of a script or module.
3. **Local Namespace** – Created inside functions and stores local variables.
4. **Enclosing Namespace** – Exists in nested functions, allowing inner functions to access outer function variables.

#### Scope in Python

Scope defines the accessibility of variables in different parts of a program.

### Example

# Global variable  
global\_var = 10

def outer\_function():  
 # Enclosing variable  
 outer\_var = 20

def inner\_function():  
 # Local variable  
 inner\_var = 30  
 print(inner\_var) # Accessible within inner\_function  
 print(outer\_var) # Accessible from enclosing function  
 inner\_function()  
 print(global\_var) # Accessible globally  
  
outer\_function()

Global, Local and nonlocal variables

#### Local Variables

Local variables are declared inside a function and can only be accessed within that function.

Example

def greet():  
 message = "Hello, World!" # Local variable  
 print(message)  
  
greet()  
print(message) # Error: message is not defined outside the function

#### Global Variables

Global variables are declared outside any function and can be accessed anywhere in the program.

Example

message = "Hello, World!" # Global variable  
  
def greet():  
 print(message) # Accessible inside the function  
  
greet()  
print(message) # Accessible outside the function

#### Using the global Keyword

The global keyword allows modifying a global variable inside a function.

Example

counter = 0 # Global variable  
  
def increment():  
 global counter  
 counter += 1 # Modifying global variable  
  
increment()  
print(counter) # Output: 1

#### Nonlocal Variables

Nonlocal variables are used inside nested functions to modify variables from an enclosing function.

Example

def outer():  
 count = 0 # Enclosing variable  
  
 def inner():  
 nonlocal count  
 count += 1 # Modifying enclosing variable  
  
 inner()  
 print(count) # Output: 1  
  
outer()

Global Keyword

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

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

### Example

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

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

Iterators

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

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

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods \_\_iter\_\_() and \_\_next\_\_().

## Iterator vs Iterable

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

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

### Example

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

mytuple = ("apple", "banana", "cherry")  
myit = iter(mytuple)  
  
print(next(myit))  
print(next(myit))  
print(next(myit))

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

### Example

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

mystr = "banana"  
myit = iter(mystr)  
  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))

Iterators Using for

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

### Example

Iterate the values of a tuple:

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

### Example

Iterate the characters of a string:

mystr = "banana"  
  
for x in mystr:  
 print(x)

## Create an Iterator

To create an object/class as an iterator you have to implement the methods \_\_iter\_\_() and \_\_next\_\_() to your object.

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.

### Example

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

class MyNumbers:  
 def \_\_iter\_\_(self):  
 self.a = 1  
 return self  
  
 def \_\_next\_\_(self):  
 x = self.a  
 self.a += 1  
 return x  
  
myclass = MyNumbers()  
myiter = iter(myclass)  
  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))

## StopIteration

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

To prevent the iteration 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:

### Example

Stop after 20 iterations:

class MyNumbers:  
 def \_\_iter\_\_(self):  
 self.a = 1  
 return self  
  
 def \_\_next\_\_(self):  
 if self.a <= 20:  
 x = self.a  
 self.a += 1  
 return x  
 else:  
 raise StopIteration  
  
myclass = MyNumbers()  
myiter = iter(myclass)  
  
for x in myiter:  
 print(x)

Inheritance

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

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

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

## Create a Parent Class

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

### Example

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

class Person:  
 def \_\_init\_\_(self, fname, lname):  
 self.firstname = fname  
 self.lastname = lname  
  
 def printname(self):  
 print(self.firstname, self.lastname)  
  
#Use the Person class to create an object, and then execute the printname method:  
  
x = Person("John", "Doe")  
x.printname()

## Create a Child Class

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

### Example

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

class Student(Person):  
 pass

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

### Example

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

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

## Types of Python Inheritance

**Single Inheritance**: A child class inherits from one parent class.

**Multiple Inheritance**: A child class inherits from more than one parent class.

**Multilevel Inheritance**: A class is derived from a class which is also derived from another class.

**Hierarchical Inheritance**: Multiple classes inherit from a single parent class.

**Hybrid Inheritance**: A combination of more than one type of inheritance.

Multiple Inheritance

Multiple inheritance in Python allows you to construct a class based on more than one parent classes. The Child class thus inherits the attributes and method from all parents. The child can override methods inherited from any parent.

### Syntax

class parent1:

#statements

class parent2:

#statements

class child(parent1, parent2):

#statements

### Example

class Mammal:

def mammal\_info(self):

print("Mammals can give direct birth.")

class WingedAnimal:

def winged\_animal\_info(self):

print("Winged animals can flap.")

class Bat(Mammal, WingedAnimal):

pass

# create an object of Bat class

b1 = Bat()

b1.mammal\_info()

b1.winged\_animal\_info()

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:

### Example

def my\_function(**fname**):  
 print(fname + " Refsnes")  
  
my\_function(**"Emil"**)  
my\_function(**"Tobias"**)  
my\_function(**"Linus"**)

## 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.

### Example

This function expects 2 arguments, and gets 2 arguments:

def my\_function(fname, lname):  
 print(fname + " " + lname)  
  
my\_function("Emil", "Refsnes")

## 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:

### Example

If the number of arguments is unknown, add a \* before the parameter name:

def my\_function(\*kids):  
 print("The youngest child is " + kids[2])  
  
my\_function("Emil", "Tobias", "Linus")

## Keyword Arguments

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

This way the order of the arguments does not matter.

### Example

def my\_function(child3, child2, child1):  
 print("The youngest child is " + child3)  
  
my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

Python Functions

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

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

A function can return data as a result.

## Creating a Function

In Python a function is defined using the def keyword:

### Example

def my\_function():  
 print("Hello from a function")

## Calling a Function

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

### Example

def my\_function():  
 print("Hello from a function")  
  
**my\_function()**

Break statements

## The break Statement

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

### Example

Exit the loop when x is "banana":

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

Continue statement

## The continue Statement

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

### Example

Do not print banana:

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

Errors and Exception

Errors are problems in a program that causes the program to stop its execution. On the other hand, exceptions are raised when some internal events change the program’s normal flow.

Exception Try,except and finally

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:

### Example

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

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

## Many Exceptions

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

### Example

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

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

## Else

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

### Example

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

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

## Finally

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

### Example

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

## 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.

### Example

Raise an error and stop the program if x is lower than 0:

x = -1  
  
if x < 0:  
 raise Exception("Sorry, no numbers below zero")

User Defined Exception

User-defined exceptions in Python are custom error classes that you create to handle specific error conditions in your code. They are derived from the built-in Exception class or any of its sub classes.

OOP Approach

[Object-Oriented Programming (OOP)](https://www.geeksforgeeks.org/python-oops-concepts/) is a fundamental paradigm in Python which is designed to model real-world entities through classes and objects. It offers features like **encapsulation, inheritance, polymorphism, and abstraction** enabling developers to write reusable, modular, and maintainable code.

Nested Dictionary Implementation

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

### Example[Get](https://www.w3schools.com/python/python_server.asp)

Create a dictionary that contain three dictionaries:

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

## Access Items in Nested Dictionaries

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

### Example

Print the name of child 2:

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

## Loop Through Nested Dictionaries

You can loop through a dictionary by using the items() method like this:

### Example

Loop through the keys and values of all nested dictionaries:

for x, obj in myfamily.items():  
 print(x)  
  
 for y in obj:  
 print(y + ':', obj[y])

Python Operator Overloading

**Operator Overloading** means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

# # Python program to show use of

# # + operator for different purposes.

print(1 + 2)

# # concatenate two strings

print("Geeks"+"For")

# # Product two numbers

print(3 \* 4)

# Repeat the String

print("Geeks"\*4)

Statements and Comments

#### Statements in Python

A **statement** in Python is an instruction that the interpreter can execute. Python statements include assignments, function calls, loops, and conditionals.

### Example

x = 10 # Assignment statement  
print(x) # Function call statement  
  
if x > 5: # Conditional statement  
 print("x is greater than 5")  
  
for i in range(3): # Loop statement  
 print(i)

#### Comments in Python

Comments are used to explain code, improve readability, and prevent execution when testing.

1. **Single-Line Comments** – Use # to write a comment.

# This is a single-line comment  
print("Hello, World!") # This prints a message

1. **Multi-Line Comments** – Use multiple # or triple quotes (""" or ''').

# This is a  
# multi-line comment  
print("Python is fun!")  
  
"""This is another way  
to write multi-line comments."""  
print("Learning Python!")

Pass Statement

## The pass Statement

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

### Example

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

Generators

A generator [function](https://www.geeksforgeeks.org/python-functions/) is a special type of function that returns an iterator object. Instead of using [return](https://www.geeksforgeeks.org/python-return-statement/) to send back a single value, generator functions use [yield](https://www.geeksforgeeks.org/python-yield-keyword/) to produce a series of results over time. This allows the function to generate values and pause its execution after each yield, maintaining its state between iterations.

def fun(max):

cnt = 1

while cnt <= max:

yield cnt

cnt += 1

ctr = fun(5)

for n in ctr:

print(n)

Decorators

In [Python](https://www.geeksforgeeks.org/python-programming-language-tutorial/), decorators are a powerful and flexible way to modify or extend the behavior of functions or methods, without changing their actual code. A decorator is essentially a [function](https://www.geeksforgeeks.org/python-functions/) that takes another function as an argument and returns a new function with enhanced functionality.

**# A simple decorator function**

def decorator(func):

def wrapper():  
 print("Before calling the function.")  
 func()  
 print("After calling the function.")  
return wrapper

# **# Applying the decorator to a function**

@decorator

def greet():

print("Hello, World!")

greet()

While loop

## The while Loop

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

### Example

Print i as long as i is less than 6:

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

Use of if, elif and else

Conditional statements in Python allow decision-making in programs. The if, elif, and else statements help execute different blocks of code based on conditions.

#### Using if Statement

The if statement executes a block of code only if the condition evaluates to True.

Example

x = 10  
  
if x > 5:  
 print("x is greater than 5")

#### Using if...else Statement

The else statement executes when the if condition evaluates to False.

Example

x = 3  
  
if x > 5:  
 print("x is greater than 5")  
else:  
 print("x is 5 or less")

#### Using if...elif...else Statement

The elif statement allows checking multiple conditions.

Example

score = 85  
  
if score >= 90:  
 print("Grade: A")  
elif score >= 75:  
 print("Grade: B")  
elif score >= 60:  
 print("Grade: C")  
else:  
 print("Grade: D")

Matrix Implementation

A **matrix** is a two-dimensional data structure consisting of rows and columns. Python does not have a built-in matrix type, but matrices can be implemented using **nested lists** or the **NumPy library**.

#### Matrix Using Nested Lists

Python allows matrix representation using lists within lists.

Example

**Creating a Matrix**

matrix = [  
 [1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]  
]  
  
print(matrix) # Output: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

**Accessing Elements in a Matrix**

print(matrix[0][1]) # Output: 2 (Element at row 0, column 1)

#### Matrix Operations

**Matrix Addition**

A = [[1, 2], [3, 4]]  
B = [[5, 6], [7, 8]]  
  
result = [[A[i][j] + B[i][j] for j in range(len(A[0]))] for i in range(len(A))]  
print(result) # Output: [[6, 8], [10, 12]]

**Matrix Multiplication**

A = [[1, 2], [3, 4]]  
B = [[5, 6], [7, 8]]  
  
result = [[sum(A[i][k] \* B[k][j] for k in range(len(B))) for j in range(len(B[0]))] for i in range(len(A))]  
print(result) # Output: [[19, 22], [43, 50]]

#### Matrix Using NumPy

NumPy provides efficient matrix operations.

Example

**Creating a Matrix Using NumPy**

import numpy as np  
  
A = np.array([[1, 2], [3, 4]])  
B = np.array([[5, 6], [7, 8]])  
  
print(A + B) # Matrix Addition  
print(A.dot(B)) # Matrix Multiplication

Regular Expressions

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

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

## RegEx Module

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

Import the re module:

import re

## RegEx in Python

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

### Example

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

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

## The findall() Function

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

### Example

Print a list of all matches:

import re  
  
txt = "The rain in Spain"  
x = re.findall("ai", txt)  
print(x)

## The search() Function

The search() function searches the string for a match, and returns a [Match object](https://www.w3schools.com/python/python_regex.asp#matchobject) if there is a match.

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

### Example

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

import re  
  
txt = "The rain in Spain"  
x = re.search("\s", txt)  
  
print("The first white-space character is located in position:", x.start())

## The split() Function

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

### Example

Split at each white-space character:

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

## The sub() Function

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

### Example

Replace every white-space character with the number 9:

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

List Comprehension

**List comprehension** is a way to create lists using a concise syntax. It allows us to generate a new list by applying an **expression** to each item in an existing **iterable** (such as a **list** or **range**). This helps us to write cleaner, more readable code compared to traditional looping techniques.

### Example

fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
newlist = []  
  
for x in fruits:  
 if "a" in x:  
 newlist.append(x)  
  
print(newlist)

Recursion

Recursion involves a function calling itself directly or indirectly to solve a problem by breaking it down into simpler and more manageable parts. In [Python](https://www.geeksforgeeks.org/python-programming-language-tutorial/), recursion is widely used for tasks that can be divided into identical subtasks.

In Python, a [recursive function](https://www.geeksforgeeks.org/recursive-functions/) is defined like any other [function](https://www.geeksforgeeks.org/python-functions/), but it includes a call to itself. The syntax and structure of a recursive function follow the typical function definition in Python, with the addition of one or more conditions that lead to the function calling itself.

### Example [Get](https://www.w3schools.com/python/python_server.asp)

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n-1)

print(factorial(5))

Input, Output and Import

#### **Input in Python**

Python uses the input() function to take user input from the keyboard.

Example

name = input("Enter your name: ")  
print("Hello, " + name + "!")

#### **Output in Python**

Python uses the print() function to display output.

Example

print("Python is fun!")

#### **Importing Modules in Python**

Python allows importing built-in and user-defined modules using the import statement.

Example

import math  
print(math.sqrt(16))

Read and Write Operations

#### **Reading from a File**

Python provides the open() function to read files. You can read the entire file or line by line.

### Example[Get your own Python Server](https://www.w3schools.com/python/python_server.asp)

### **Reading a File**

with open("example.txt", "r") as file:  
 content = file.read()  
 print(content)

**Reading Line by Line**

with open("example.txt", "r") as file:  
 for line in file:  
 print(line.strip())

#### **Writing to a File**

You can write to a file using w mode (overwrite) or a mode (append).

### Example[Get your own Python Server](https://www.w3schools.com/python/python_server.asp)

**Writing to a File**

with open("example.txt", "w") as file:  
 file.write("Hello, Python!")

**Appending to a File**

with open("example.txt", "a") as file:  
 file.write("\nAdding more content.")

Different Looping Techniques

## Python Loops

Python has two primitive loop commands:

* while loops
* for loops

## The while Loop

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

### Example

Print i as long as i is less than 6:

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

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

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

### Example

### Print each fruit in a fruit list:

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

## Nested Loops

A nested loop is a loop inside a loop.

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

### Example

Print each adjective for every fruit:

adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]  
  
for x in adj:  
 for y in fruits:  
 print(x, y)

Shallow Copy and Deep Copy

Copying in Python refers to creating a duplicate of an object. There are two types of copies:

* **Shallow Copy**: Creates a new object but references the original elements.
* **Deep Copy**: Creates a completely independent copy, including all nested objects.

#### Shallow Copy

A shallow copy creates a new object but does not create copies of nested objects. Instead, it stores references to the original elements.

### Example[Get your own Python Server](https://www.w3schools.com/python/python_server.asp)

import copy  
  
list1 = [1, 2, [3, 4]]  
list2 = copy.copy(list1) # Shallow copy  
  
list1[2][0] = 100 # Modifying nested list  
print("Original List:", list1) # Output: [1, 2, [100, 4]]  
print("Shallow Copy:", list2) # Output: [1, 2, [100, 4]] (Nested list is affected)

Since the nested list is referenced, changes in list1 affect list2.

#### Deep Copy

A deep copy creates a completely independent copy, including all nested objects.

### Example[Get your own Python Server](https://www.w3schools.com/python/python_server.asp)

import copy  
  
list1 = [1, 2, [3, 4]]  
list2 = copy.deepcopy(list1) # Deep copy  
  
list1[2][0] = 100 # Modifying nested list  
print("Original List:", list1) # Output: [1, 2, [100, 4]]  
print("Deep Copy:", list2) # Output: [1, 2, [3, 4]] (Nested list remains unchanged)

Since deepcopy() creates independent copies of nested objects, changes in list1 do not affect list2.

Assert

The assert statement in Python is used for debugging and testing conditions. It helps verify assumptions in the code and raises an AssertionError if the condition evaluates to False.

x = 10  
assert x > 5 # No error, since condition is True  
  
assert x < 5, "x should be greater than 5" # Raises AssertionError

# Output

Traceback (most recent call last):  
 File "<stdin>", line 4, in <module>  
AssertionError: x should be greater than 5

### Example

def divide(a, b):  
 assert b != 0, "Denominator cannot be zero"  
 return a / b  
  
print(divide(10, 2)) # Output: 5.0  
print(divide(10, 0)) # Raises AssertionError

@Property

The @property decorator in Python is used to define **getter methods** in a class, allowing controlled access to attributes. It helps implement **encapsulation** by restricting direct modification of attributes while providing a clean interface.

### Example

class Celsius:  
 def \_\_init\_\_(self, temperature=0):  
 self.\_temperature = temperature # Private attribute  
  
 @property  
 def temperature(self):  
 """Getter method"""  
 return self.\_temperature  
  
 @temperature.setter  
 def temperature(self, value):  
 """Setter method with validation"""  
 if value < -273.15:  
 raise ValueError("Temperature below -273.15 is not possible.")  
 self.\_temperature = value  
  
# Using the class  
c = Celsius()  
c.temperature = 25 # Calls setter method  
print(c.temperature) # Calls getter method

#### **Why Use @property?**

* **Encapsulation** – Prevents direct modification of attributes.
* **Validation** – Ensures data integrity before assignment.
* **Improved Readability** – Allows attribute-like access instead of method calls.

Closure

A **closure** in Python is a nested function that retains access to variables from its enclosing function, even after the outer function has finished executing. Closures are useful for maintaining state and avoiding global variables.

### Example

def outer\_function(message):  
 def inner\_function():  
 return f"Hello, {message}!"  
 return inner\_function # Returning the inner function  
  
greet = outer\_function("Alice") # Calling outer function  
print(greet())

# Output: Hello, Alice!

Here, inner\_function() retains access to message even after outer\_function() has completed execution.

#### **Using Closures for Stateful Functions**

Closures can be used to maintain state across multiple function calls.

def counter():  
 count = 0  
 def increment():  
 nonlocal count # Allows modification of enclosing variable  
 count += 1  
 return count  
 return increment  
  
counter\_instance = counter()  
print(counter\_instance()) # Output: 1  
print(counter\_instance()) # Output: 2  
print(counter\_instance()) # Output: 3