# Coding Convention

Note: Things which are not mentioned below are supposed to follow the same convention as in C++ or C#.

## Indentation and Code Block

Python **uses indentation for blocks** instead of curly braces like C/C++ or Java. Both tabs and spaces are supported, as long as it is consistent. But the standard convention of Python is to use **four spaces**.

The following code shows you how a code block in Python is created:

x = 1

if x == 1:

# indented four spaces

x = 2

print("x is changed from 1 to 2")

Note:

* Python 3 disallows mixing the use of tabs and spaces for indentation.

## Source File Encoding

* Code in the core Python distribution should always use UTF-8 (or ASCII in Python 2).
* Files using ASCII (in Python 2) or UTF-8 (in Python 3) should not have an encoding declaration.

## Naming Convention

**Packages and Modules:**

* Should be all lower case.
* Preferable to stick to 1 word names.

**Classes:**

* Should follow the CamelCase convention (however, Python’s built-in classes are typically lowercase words).

**Variables and Methods:**

* Should be all lower case.
* Words in a name should be separated by a single underscore (\_).
* Non-public variables/methods should BEGIN their name with a single underscore (\_).
* Private variables/methods MUST BEGIN their name with a double underscore (\_\_). This is the *name mangling* that the Python interpreter applies.
* If a variable/method name cannot be used because it conflicts with a keyword, it should END with a single underscore (\_) to avoid the conflict. For example, a list can be named as list\_.
* Variables/methods, which are reserved for special use in the language, should BEGIN and END their names with a double underscore (\_\_). For example, object constructor is named as \_\_init\_\_.
* Temporary or unused variables can be named as a single standalone underscore. For example:

for \_ in range(10):

print('Hello, World.')

**Method Arguments:**

* Instance methods should have their first argument named self.
* Class methods should have their first argument named cls.

**Constants:**

* Constant names must be FULLY CAPITALIZED
* Words in a constant name should be separated by a single underscore (\_).

**Global Variables:**

* Global variables should be all lowercase.
* Words in a global variable name should be separated by a single underscore (\_).

## Importing

Different imports should be on separate lines.

Imports are always put at the top of the file, just after any module comments and docstrings, and before module global variables and constants.

Imports should be grouped in the following order:

1. Standard library imports.
2. Related third-party imports.
3. Local application/library specific imports.

You should put a blank line between each group of imports.

Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path):

import mypkg.mymodule

from mypkg import mymodule

from mypkg.mymodule import Myclass

## Documentation Strings

Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.

class MenuConfig:

    """A configuration for the Menu.

    Attributes:

        title: The title of the Menu.

        body: The body of the Menu.

        button\_text: The text for the button label.

        cancellable: Can it be cancelled?

    """

    title: str

...

Note: The """ that ends a multiline docstring should be on a line by itself.

## Module Level Dunder Names

Module level "dunders" (i.e. names with two leading and two trailing underscores) should be placed after the module docstring, but before any import statements except from \_future\_ imports. Python mandates that future-imports must appear in the module before any other code except docstrings:

"""This is the example module.

This module does stuff.

"""

from \_future\_ import barry\_as\_FLUFL

\_\_all\_\_ = ['a', 'b', 'c']

\_\_version\_\_ = '0.1'

\_\_author\_\_ = 'Cardinal Biggles'

import os

import sys

## Others

<https://www.python.org/dev/peps/pep-0008/#programming-recommendations>

<https://realpython.com/python-pep8/>

# Python Itself vs. Python Implementation vs. Python Distribution

<https://stackoverflow.com/questions/27450172/python-implementation-vs-python-distribution-vs-python-itself>

**Tips**:

* To know which Python implementation you're using, type in the Command Prompt or Terminal:

py -c "import platform; print(platform.python\_implementation())"

# Variables and Data Types

Unlike C which is *statically typed*, Python is *dynamic typed*, so a **variable type is not required when declaration and it can be changed throughout the program** . Also, Python is completely object-oriented, so **every variable is an object**.

## Data Types

### Primitive

#### Number

Python supports three types of numbers - *integers*, *floating point* numbers and *complex* numbers.

Example:

# integer

myInt = 7

print(myInt) # Print out "7"

# floating point

myFloat = 7.0

print(myFloat) # Print out "7.0"

* **Number operations**

print(2 + 2)         # prints out 4

print(50 - 5\*6)  # prints out 20

print(17 / 3) # prints out 5.666666666666667

print(17 // 3) # prints out 5

print(17 % 3) # prints out 2

print(5 \*\* 2) # 5^2, prints out 25

#### String

Strings are defined either with a *single quote* or a *double quote*.

Example:

myString = 'hello'

# is same as

myString = "hello"

The difference between these two styles is that using double quotes makes it easy to include apostrophes "'". Whereas using single quotes makes it easy to include """.

myString = "Don't worry about apostrophes"

# is same as

myString = 'Don\'t worry about apostrophes' # must use escape character "\"

myString = '"Yes" they say'

# is same as

myString = "\"Yes\"" they say" # must use escape character "\"

* **String Manipulation**

Simple operators can be executed on strings using operator "+", just like numbers:

hello = "hello"

world = "world"

helloWorld = hello + " " + world

print(helloWorld) # Print out "Hello world"

* **String Comparison**

To compare strings in Python, we use the == and != operators, and NOT the is keyword.

For more details about comparison operators in Python, check [this section](#_Comparison_Operator).

#### Boolean

has\_passed = False

marks = 80

if (marks > 50):

has\_passed = True

print (has\_passed) # This prints out "True"

### Non-Primitive

Also known as [collections](#_Collections) in Python.

## Type Casting

There may be times when you want to specify a type on to a variable. This can be done with casting which is done using constructors:

* int() - Constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number).
* float() - Constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer).
* str() - Constructs a string from a wide variety of data types (strings, integer literals and float literals).

For example:

a = int(2.8) # a will be 2

b = int("3") # b will be 3

c = float(1) # c will be 1.0

d = float("4.2") # d will be 4.2

e = str(2) # e will be "2"

f = str(3.0) # f will be "3.0"

# str(object)

# str(list)

# ...

## The 'type' Function

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

x = 5

print(type(x)) # This prints out "<class 'int'>"

# Now change type of x drammatically

x = 'abc'

print(type(x)) # This prints out "<class 'str'>"

## Variable Scope

Just like C, a variable in Python is only available inside the block it is created.

### Local Scope

A variable created inside a function belongs to the local scope of that function and can only be used inside that function. For example:

def func():

x = 300

print(x)

func() # This prints out "300"

print(x) # Error (NameError: name 'x' is not defined)

**Note:**

**Unlike C/C++ or Java, scope of local variables in Python is not limited to condition or loop statement**. If we change the above func() to the below code, it still works correctly. Of course, we should not write code like that, but we must notice this behavior to understand when variables are actually cleaned up from the memory.

def func():

if True:

x = 300

print(x) # This prints out "300", not error

### Global Scope

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

x = 300

def func():

print(x)

func() # This prints out "300"

* **The 'global' Keyword**

It has two purposes.

**1**. Create variable belongs to the global scope but is declared inside a function:

def func():

global x

x = 300 # Warning: cannot "global x = 300"

print(x)

func() # This prints out "300"

print(x) # This prints out "300"

**2**. Make a change to global variables inside a function:

x = 300

def func():

global x

x = 400

print(x)

print(x) # This prints out "300"

func() # This prints out "400"

print(x) # This prints out "400"

* **Tips:**
* How to share global variables across modules: [Here](https://docs.python.org/3/faq/programming.html#how-do-i-share-global-variables-across-modules).
* Global variables are dangerous because they can be simultaneously accessed from multiple sections of a program. This frequently results in bugs. A better choice would be encapsulating the global variables into objects. More detail [here](https://docs.quantifiedcode.com/python-anti-patterns/maintainability/using_the_global_statement.html).

## The 'None' Keyword

The None keyword is used to define a null variable or object. In Python, None is an object and a data type of class NoneType.

We can assign None to any variable, but you cannot create other NoneType objects.

Note:

* None is not False.
* None is not 0.
* None is not '' or "".

So, comparing None to anything will always return False, except None itself.

Example:

# Example 1

var = None

if var is None:     # Checking if the variable is None

# if var == None:   # Same meaning

    print("None")

else:

  print("Not None")

# Example 2

typeOfNone = type(None)

print(typeOfNone) # This prints out "<class 'NoneType'>"

# Conditions

Python uses Boolean variables to evaluate conditions. Either the Boolean value True or False is returned when an expression is evaluated.

For example:

x = 2

print(x == 2) # prints out "True"

print(x == 3) # prints out "False"

print(x != 3) # prints out "True"

## Conditional Keywords

statement = True

another\_statement = True

if statement == True:

print("if")

elif another\_statement == True: # can use "else if"

print("elif")

else:

# do another thing

## Operators

### Logical Operators ('and' and 'or')

The and and or Boolean operators allow building complex Boolean expressions. They are alternatives for && and || in C.

For example:

name = "John"

age = 23

if name == "John" and age == 23:

print("Your name is John, and you are also 23 years old.")

if name == "John" or name == "Rick":

print("Your name is either John or Rick.")

Output:

Your name is John, and you are also 23 years old.

Your name is either John or Rick.

### Comparison Operators ('==' and '!=')

The == operator is used to check if two variables (of any type) have the same value or not.

For the opposite meaning to the == operator, we use !=.

### Identity Operators ('is' and 'is not')

The is operator is used to check if two objects share the same memory location (or if two variables refer to the same object). For example:

x = [1, 2, 3]

y = [1, 2, 3]

print(x == y)   # Prints out True

print(x is y)   # Prints out False

print(id(x))    # Prints out 2402973475464

print(id(y))    # Prints out 2402973475528

In Python, the term 'identity' is sometimes interchangeable with memory location. That's because each object stored in the memory will have a unique identity. To know the identity of Python's objects, we use the id() method.

For the opposite meaning to the is operator, we use is not.

**You might not know**

There is an interesting fact that makes many programmers confuse. On some Python implementations (CPython for example), integers from -5 to 256 will work with the is operator (as the below example). That's because Python automatically creates those integers prior to runtime rather than constructing them on the fly in order to save time, and thus these particular integers have ids before being needed in the program.

There is also a difference in result when running Python on script and Shell. That's because when running in one script, the interpreter doesn't create another object if they have the same value. But in Shell, you're forcing the creating of another object.

Of course, **don't rely on this or use it in real programs**.

x = 1

y = 1

print(x == y)   # Prints out True

print(x is y)   # Prints out True whether running on script or Shell

print(id(x))    # Prints out 1669674545896

print(id(y)) # Prints out 1669674545896

a = 257

b = 257

print(a is b)   # Prints out True if running on script, but False if running on Shell

**A rule of thumb to follow is to use == or != when comparing numeric or string types and is or it not when comparing objects.**

### Membership Operator ('in' and 'not in')

The in operator is used to check if a specified object exists within an iterable object container, such as a list:

name = "John"

list = ["John", "Rick"]

if name in list:

print("Your name is either John or Rick.")

For the opposite meaning to the in operator, we use not in.

# String Formatting

Python uses C-style string formatting to format strings. Here are some basic argument specifiers that you find very familiar from C:

|  |  |
| --- | --- |
| **Specifier** | **Meaning** |
| %s | String (or any object with a string representation, like numbers) |
| %d | Integers |
| %f | Floating point numbers |
| %.<digit-num>f | Floating point numbers with a fixed number of digits to the right of the dot. |
| %x / %X | Integers in hex representation (lowercase / uppercase) |

For example:

# This prints out "Hello, John!"

name = "John"

print("Hello, %s!" % name) # note the "%" before name

# This prints out "John is 23 years old."

age = 23

print("%s is %d years old." % (name, age)) # note the parentheses after "%"

# This prints out "A list: [1, 2, 3]"

mylist = [1,2,3]

print("A list: %s" % mylist)

# IO Stream

## Output

We use print() to output message to the console. For example:

# This prints out "Hello, World" and a newline

print("Hello, World")

# This prints out "Hello, World" without a newline

print("Hello, World\") # note the \ at the end of the string

If you don’t want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote:

# This print out:

# C:\some

# ame

print('C:\some**\n**ame')   # here \n means a newline

# This print out:

# C:\some\name

print(r'C:\some\name')      # note the r before the quote

String literals can span multiple lines:

# This prints out:

# Usage: thingy [OPTIONS]

# -h Display this usage message

# -H hostname Hostname to connect to

print("""\

Usage: thingy [OPTIONS]

-h Display this usage message

-H hostname Hostname to connect to

""")

Two or more string literals next to each other are automatically concatenated. This feature is particularly useful when you want to break a long string:

# This prints out: Put several strings within parentheses to have them joined together.

print('Put several strings within parentheses '

'to have them joined together.')

Assignments can be done on more than one variable "simultaneously" on the same line like this:

# This prints out "3 4"

a = 3

b = 4

print(a, b)

But mixing operators between numbers and strings is not supported:

# This will NOT work!

one = 1

hello = "hello"

print(one + hello) # error here

You can print out the whole list without having to iterate each of its element:

# This prints out "[1, 2, 3]"

numbers = [1, 2, 3]

# This prints out "['John', 'Eric', 'Jessica']"

names = ["John", "Eric", "Jessica"]

You can put an expression in the print() to make it return True or False:

print(3 == 3) # Print out "True"

## Input

Python uses the input() method for user input. For example:

username = input("Enter username: ")

print("Your username is: " + username)

# Collections

There are four collection data types in Python:

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

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

## List

Lists in Python are very similar to std::vector in C++. They can contain any type of variable and as many variables as you wish. Lists can also be iterated over in a very simple manner.

Example 1: Initialize a list and access its elements by index

mylist = [1, 2, 3]

print(myList[0]) # prints 1

print(myList[1]) # prints 2

print(myList[2]) # prints 3

print(myList[3]) # error

Example 2: Append element to a list and iterate through all of its element

myList = []

myList.append(1)

myList.append(2)

myList.append(3)

# prints 1, 2, 3

for x in myList:

print(x)

More details: <https://www.w3schools.com/python/python_lists.asp>

## Tuple

<https://www.w3schools.com/python/python_tuples.asp>

## Set

## Dictionary

A dictionary is very similar to std::map in C++. Each *value* in a dictionary can be accessed using a *key*, which is any type of object (string, number, list, etc.) instead of using an index.

For example, a database of phone numbers could be stored using a dictionary like this:

phonebook = {

"John" : 938477566,

"Jack" : 938377264,

"Jill" : 947662781

}

# This is similar to:

phonebook = {}

phonebook["John"] = 938477566

phonebook["Jack"] = 938377264

phonebook["Jill"] = 947662781

# This prints out "{'Jill': 947662781, 'Jack': 938377264, 'John': 938477566}"

print(phonebook)

**Iterating Over Dictionaries**

Dictionaries can be iterated over, just like a list. However, unlike a list, a dictionary does not keep the order of the values stored in it.

To iterate over key value pairs:

phonebook = {"John" : 938477566, "Jack" : 938377264, "Jill" : 947662781}

for name, number in phonebook.items():

print("Phone number of %s is %d" % (name, number))

Output:

Phone number of Jill is 947662781

Phone number of Jack is 938377264

Phone number of John is 938477566

**Removing Elements**

To remove a specified key and its corresponding value:

phonebook = {

"John" : 938477566,

"Jack" : 938377264,

"Jill" : 947662781

}

phonebook.pop("John") # Or del phonebook["John"]

print(phonebook)

More details: <https://www.w3schools.com/python/python_dictionaries.asp>

# Loops

## The "for" Loop

Just like C, the for loop iterates over a given sequence. For example:

primes = [2, 3, 5, 7]

for prime in primes:

# Prints out 2, 3, 5, 7

print(prime)

* **Range functions**

For loops can iterate over a sequence of numbers using the range() and xrange() functions. The difference between range and xrange is that the range function returns a new list with numbers of that specified range, whereas xrange returns an iterator, which is more efficient.

Note: Python 3 uses the range() function, which acts like xrange().

# Prints out the numbers 0, 1, 2

for x in range(3): # zero-based

print(x)

# Prints out 3, 4, 5

for x in range(3, 6):

print(x)

# Prints out 3, 5, 7

for x in range(3, 8, 2):

print(x)

## The "while" loops

Just like C, the while loop repeats as long as a certain boolean condition is met. For example:

count = 0

while count < 3:

print(count)

count += 1 # Same as count = count + 1

print("Loop was already ended")

Output:

0

1

2

Loop was already ended

## The "break" and "continue" Statements

Just like C, break is used to exit a for or while loop, whereas continue is used to skip the current block, and return to the for or while statement. A few examples:

# Prints out 0, 1, 2, 3, 4

count = 0

while True:

print(count)

count += 1

if count >= 5:

break

# Prints out only odd numbers - 1, 3, 5, 7, 9

for x in range(10):

# Check if x is even

if x % 2 == 0:

continue

print(x)

## The "else" for Loops

Unlike C, we can use else for loops. When the loop condition of for or while statement fails then code part in else is executed.

If break statement is executed inside the loop then the else part is skipped.

Note that else part is executed even if there is a continue statement.

Here are a few examples:

# Prints out 0, 1, 2, 3, 4 and then it prints "count value reached 5"

Count = 0

While count < 5:

print(count)

count += 1

else:

print("count value reached %d" %(count))

# Prints out 1, 2, 3, 4

for i in range(1, 10):

if(i % 5 == 0):

break

print(i)

else:

print("this is not printed because of break")

# Functions

Just like C, functions are a convenient way to divide our code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time.

## Function Definition

Functions in Python are defined using the block keyword def, followed with the function's name as the block's name. For example:

def my\_function():

print("Hello!")

Functions may receive arguments. For example:

def my\_function\_with\_args(*username*, *greeting*):

print("Hello, %s , I wish you %s" %(*username*, *greeting*))

Functions may return a value. For example:

def sum\_two\_numbers(a, b):

return a + b

## Function Calling

my\_function()

my\_function\_with\_args("John Doe", "a great year!")

x = sum\_two\_numbers(1, 2)

# Lambda Expressions

A lambda (fully called *lambda expression* or *lambda function*) is a small **anonymous function which can be defined in another function**. It can take any number of arguments, but can only have one expression.

## Syntax

lambda argument(s) : expression

## Example

# lambda with one argument

x = lambda a : a + 10

print(x(5)) # This prints out "15"

# lambda with two arguments

y = lambda a, b : a \* b

print(y(5, 6)) # This prints out "30"

## Why Lambda?

The lambda really shines when you use them as an anonymous function inside another function. For example:

def func(n):

return lambda a : a \* n

doubler = func(2)

print(doubler(11)) # This prints out "22"

tripler = func(3)

print(tripler(11)) # This prints out "33"

# OOP

The rule of object-orient programming in Python is basically similar to C++. *Objects* are an encapsulation of variables and functions into a single entity. Objects get their member variables and functions from classes. *Classes* are essentially a template to create your objects.

## Class

### Example

The following shows you how to create a class and its members (variables and methods), as well as how to access its members.

class MyClass:

variable = "blah"

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

self.class\_name = "MyClass" # Note the keyword "self"

def function(self):

print("This is a message inside the class", self.class\_name)

myclass = MyClass()

myclass.function()

Output:

This is a message inside the class MyClass

### Constructor

The function \_\_init\_\_(self) has the same role as *constructor* in C++. It is called automatically every time the class is being used to create a new object. And it can have arguments just like C++ constructors. For example:

class MyClass:

variable = "blah"

def \_\_init\_\_(self, *class\_name*):

        print('MyClass created')

self.class\_name = *class\_name*

def function(self):

print("Method of MyClass ", self.class\_name)

myclass = MyClass("MyClass") # Argument of the \_init\_()

myclass.function()

Output:

MyClass created

Method of MyClass

### Destructor

The function \_\_del\_\_(self) has the same role as *destructor* in C++. It is called automatically **when the object's reference count becomes zero (not when object went out of scope) or when the program reaches to the end**.

However, in case of reference cycles, \_\_del\_\_ will never be called, except when the program ends.

Example 1:

class MyClass:

    def \_\_init\_\_(self):

        print('MyClass created')

    def \_\_del\_\_(self):

        print('MyClass deleted')

myclass = MyClass()

# del myclass # Destructor is always called regardless of the [del](#_The_'del'_Statement) statement

Output:

MyClass created

MyClass deleted

Example 2:

class MyClass:

    def \_\_init\_\_(self):

        print('MyClass created')

    def \_\_del\_\_(self):

        print("MyClass deleted")

def create\_obj():

    print('Make object')

    obj = MyClass()

    print('Function end')

    return obj

obj = create\_obj()

print('Program end')

Output:

Make object

MyClass created

Function end

Program end

MyClass deleted

Notice that the destructor is called after the '*program end*', not '*function end*'.

In Python, we rarely need to explicitly define \_\_del\_\_ (as we have in C++) because Python has a [reference counter](#_Reference_Counting) and a [garbage collector](#_Garbage_Collector).

**So, when to use \_\_del\_\_?**

The \_\_del\_\_ can be useful when we want to close unmanaged resources (file reading/writing, socket connections, caches flushed, etc.) before de-allocating them. Just make sure there is no reference cycles in your code so that \_\_del\_\_ can be called.

**Tip**: **There are better way** to de-allocate unmanaged resources. Check [this section](#_How_to_De-Allocate) for more details.

### Member Access

To access variables or functions of classes from outside, you would do the following:

myclass.variable # first create an instance of the class object

myclass.function() # now you can access its members

Note: To access member variables and functions inside classes, you must put a self. before their names.

### Attributes

|  |  |
| --- | --- |
| Attribute | Role |
| \_doc\_ | Class documentation string (if undefined). |
| \_name\_ | Class name. |
| \_module\_ | Module name in which the class is defined. This attribute is "\_\_main\_\_" in interactive mode. |
| \_bases\_ | Base class names in the order of their occurrence in the base class list. |

For example:

class Employee:

"Common base class for all employees"

empCount = 0

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

self.name = name

self.empCount += 1

def displayCount(self):

print ("Total Employee: ", Employee.empCount)

def displayEmployee(self):

print ("Name : ", self.name)

print("Employee.\_\_doc\_\_:", Employee.\_\_doc\_\_)

print("Employee.\_\_name\_\_:", Employee.\_\_name\_\_)

print("Employee.\_\_module\_\_:", Employee.\_\_module\_\_)

print("Employee.\_\_bases\_\_:", Employee.\_\_bases\_\_)

print("Employee.\_\_dict\_\_:", Employee.\_\_dict\_\_)

Output:

Employee.\_\_doc\_\_: Common base class for all employees

Employee.\_\_name\_\_: Employee

Employee.\_module\_\_: \_\_main\_

Employee.\_\_bases\_\_: (<class 'object'>,)

## Properties

### Inheritance

Just like C++, inheritance in Python 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.

For example:

# Parent class

class Person:

def \_\_init\_\_(self, fName, lName):

self.firstName = fName

self.lastName = lName

def printName(self):

print(self.firstName, self.lastName)

# Child class

class Student(Person): # *Student* now inherits *Person*

pass

student = Student("Tri", "Ho")

student.printName() # This prints out "Tri Ho"

Note that in the above example, we use the pass keyword when we don’t want to add any other properties or methods to the class.

### Polymorphism

Just like C++, polymorphism in Python allows us to override methods of parent classes to have a different functionality in your child classes.

class Parent:

   def my\_method(self):

      print('Calling parent method')

class Child(Parent):

   def my\_method(self):

      print('Calling child method')

child = Child()

child.my\_method()       # This prints out " Calling child method"

### Encapsulation

Just like C++, encapsulation in Python allows us to make object's attributes and methods 'invisible' outside the class definition. To do that, you need to name attributes or methods with a double underscore prefix (\_\_).

In the below example, we can access the public() method outside the MyClass, but we cannot do the same thing with \_\_private() method or \_\_msg attribute:

class MyClass:

    \_\_msg = "This is a message inside the class"

    def \_\_init\_\_(self):

        self.class\_name = "MyClass"

    def \_\_private(self):

        print(self.\_\_msg, self.class\_name)

    def public(self):

        self.\_\_private()

myclass = MyClass()

myclass.public() # Okay

myclass.\_\_private() # Error

Output:

This is a message inside the class MyClass

Traceback (most recent call last):

File "c:/Users/ADMIN/Desktop/test.py", line 15, in <module>

myclass.\_\_private()

AttributeError: 'MyClass' object has no attribute '\_\_private'

### Abstraction

## The "super()" Function

Python has a super() function that allows a child class to call methods and properties from its parent class without having to specify the parent name.

For example:

class Mammal:

def \_\_init\_\_(self, *mammalName*):

print(*mammalName*, 'is a warm-blooded animal.')

class Dog(Mammal):

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

print('Dog has four legs.')

super().\_\_init\_\_('Dog') # Same as "Mammal. \_\_init\_\_('Dog')"

dog = Dog()

Output:

Dog has four legs.

Dog is a warm-blooded animal.

## Operator Overload

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could define the \_\_add\_\_ method in your class to perform vector addition and then the plus operator would behave as per expectation:

class Vector:

   def \_\_init\_\_(self, a, b):

      self.a = a

      self.b = b

    # Printable string representation

   def \_\_str\_\_(self):

      return 'Vector (%d, %d)' % (self.a, self.b)

   def \_\_add\_\_(self, other):

      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2, 10)

v2 = Vector(5, -2)

print(v1 + v2)  # This prints out "Vector (7, 8)"

# The 'yield' Keyword

The yield statement suspends function’s execution and sends a value back to the caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows the code to produce a series of values over time, rather than computing them at once and sending them back like a list.

For example:

def simple\_generator():

    yield 1

    yield 2

    yield 3

# This prints out 1, 2, 3

for value in simple\_generator():

print(value)

Comparing to the keyword return which sends a specified value back to its caller, whereas yield produces a sequence of values. We use it when we want to iterate over a sequence, but don’t want to store the entire sequence in memory (inefficient in case of having lots of data).

The yield is most commonly used for implementing [generators](https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained) in Python. A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a function contains yield, the function automatically becomes a generator function.

For example:

def next\_square():

    i = 1;

    # An Infinite loop to generate squares

    while True:

        yield i\*i

        i += 1   # Next execution resumes from this point

# This prints out 1, 4, 9, 16, 25, 36, 49, 64, 81, 100

for num in next\_square():

    if num > 100:

         break

print(num)

More details: <https://pythontips.com/2013/09/29/the-python-yield-keyword-explained/>

# Enum

<https://docs.python.org/3/library/enum.html>

# Modules

In programming, a module is a piece of software that has a specific functionality. For example, when building a game, one module would be responsible for the game logic, and another module would be responsible for the game UI. Each module is a different file, which can be edited separately.

## Writing Modules

**Modules in Python are simply Python files with a .py extension**. The name of the module will be the name of the file. A Python module can have a set of functions, classes or variables defined and implemented.

In the example above, we will have two files:

mygame/game.py

mygame/draw.py

The game.py will implement the game. It will use the draw\_game() which is a member function of the Drawer class from draw.py for drawing the game on the screen.

## Importing Modules

Modules are imported to other modules using the import command. In this example, the game.py may look something like this:

# In game.py

# Import the draw module

import draw

def main():

drawer = draw.Drawer() # Suppose Drawer's constructor has no parameter

drawer.draw\_game()

# This means if this script is executed, main() will be executed

if \_\_name\_\_ == '\_\_main\_\_':

main()

The draw module may look something like this:

# In draw.py

class Drawer:

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

pass

def draw\_game():

...

In this example, the game module imports the draw module, which enables it to use objects (and members) implemented in that module, such as Drawer class and draw\_game() function. But first we have to specify in which module the object is implemented, using the dot operator, such as draw.Drawer().

If you lack of "draw." (only drawer = Drawer()), you will get error "*Module object is not callable*" because the compiler cannot know which module the Drawer object belongs to.

*You might not know!*

When the import draw directive runs, the Python *interpreter* will look for a file in the directory which the script was executed from, by the name of the module with a .py prefix. In our case it will try to look for draw.py. If it find one, it will import it. **If not, it will continue to look for built-in modules**.

You may have noticed that when importing a module, a .pyc file (in a folder named \_\_pycache\_\_) appears. It is the compiled Python file. Python compiles files into Python bytecode so that it won't have to parse the files each time a module is loaded. If a .pyc file exists, it gets loaded instead of the .py file, but this process is transparent to the user.

* **Importing a specific object from a module to the current namespace**

A module can have multiple objects. We can import only the Drawer object into the main script's namespace using the from command.

# In game.py

from draw import Drawer

def main():

drawer = Drawer() # No "draw." required

drawer.draw\_game()

The advantages of using this notation is that it is easier to use the functions inside the current module because you don't need to specify which module the function comes from. However, **a namespace cannot have two objects with the exact same name**, so the import command may replace an existing object in the namespace.

* **Importing all objects from a module**

We may use the import \* command to import all objects from a specific module, like this:

# In game.py

# Import the draw module

from draw import \*

def main():

drawer = Drawer() # No "draw." required too

drawer.draw\_game()

This might be a bit risky as changes in the module might affect the module which imports it, but it is shorter and also does not require you to specify which objects you wish to import from the module. However, this is considered a **bad habit** for [various reasons](https://www.geeksforgeeks.org/why-import-star-in-python-is-a-bad-idea/).

***Warning***

The more modules are imported, the more memory is consumed when the script is executed. So as a good practice, **never import modules/packages you do not use** in your program.

## Main Guard

In the first example, you saw if \_\_name\_\_ == '\_\_main\_\_'. What does this line mean? Consider the following example:

print("Before import")

import math

print("Before functionA")

def functionA():

print("Function A")

print("Before functionB")

def functionB():

print("Function B")

def main():

print("main()")

functionA()

functionB()

print("Before \_name\_ guard")

if \_\_name\_\_ == '\_\_main\_\_':

print("Before calling main()")

main()

print("After \_name\_ guard")

Output:

Before import

Before functionA

Before functionB

Before \_name\_ guard

Before calling main()

main()

Function A

Function B

After \_name\_ guard

As you see, in a script language like Python, statements are executed from top to bottom. Unlike C, it doesn’t follow the rule "The main() function must be always called first wherever it is placed in the project.".

In addition, code of **function definitions must be placed above function callings**. In the above example, placing the functionA() before def functionA(): ... will cause syntax error.

* **How Python knows which function is the main?**

Whenever the Python interpreter reads a source file, it does two things:

1. It sets a few special variables. In this case, we care about the \_\_name\_\_ variable.
2. It then executes all of the code found in the file.

**When your module is the main program:**

If you are running your module as the main program, e.g. python foo.py, the interpreter will assign string \_\_main\_\_ to the \_\_name\_\_ variable.

**When your module is imported by another:**

On the other hand, suppose another module is the main program and it imports your module. This means there is at least a statement import foo in the main program or in some other modules.

In this case, the interpreter will look at the filename of your module, foo.py, strip off the .py, and assign that string to your module's \_\_name\_\_ variable (\_\_name\_\_ = "foo").

**Executing the module's code:**

After special variables are set up, the interpreter executes all the code in the module, one statement at a time. If your module is the main program, the if \_\_name\_\_ == '\_\_main\_\_' is True. The main() and all functions it call will be executed. By contrast, if your module is not the main program, then guard is False. So, no statements in the if is executed.

**Why does it work this way?**

You might naturally wonder why anybody would want this. Well, sometimes you want to write a .py file that can be both used by other programs and/or modules as a module, and can also be run as the main program itself. Examples:

* Your module is a library, but you want to have a script mode where it runs some unit tests or a demo.
* Your module is only used as a main program, but it has some unit tests, and the testing framework works by importing .py files like your script and running special test functions. You don't want it to try running the script just because it's importing the module.
* Your module is mostly used as a main program, but it also provides a programmer-friendly API for advanced users.

## Customizing Import Name

We may load modules under any name we want. This is useful when we want to import a module conditionally to use the same name in the rest of the code.

For example, if you have two draw modules with slighty different names - you may do the following:

# In game.py

# Import the draw module

if visual\_mode:

# In visual mode, we draw using graphics

import draw\_visual as draw

else:

# In textual mode, we print out text

import draw\_textual as draw

def main():

result = play\_game()

# This can either be visual or textual depending on visual\_mode

draw.draw\_game(result)

## Module Initialization

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. **If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton"** - they are initialized only once.

This is useful to know, because this means that you can rely on this behavior for initializing objects. For example:

# In draw.py

def draw\_game():

# When clearing the screen we can use the main screen object initialized in this module

clear\_screen(main\_screen)

...

def clear\_screen(screen):

...

class Screen():

...

# Initialize main\_screen as a singleton

main\_screen = Screen()

## Extending Module Load Path

There are a couple of ways we could tell the Python interpreter where to look for modules, aside from the default, which is the local directory and the built-in modules.

You could either use the environment variable PYTHONPATH to specify additional directories to look for modules in, like this:

PYTHONPATH=/foo python game.py

This will execute game.py and will enable the script to load modules from the foo directory as well as the local directory.

Another way is using the sys.path.append function. You may execute it before running an import command:

sys.path.append("/foo")

This will add the foo directory to the list of paths to look for modules in as well.

## Exploring Built-In Modules

Check out the full list of built-in modules in Python standard library here.

Two very important functions come in handy when exploring modules in Python - the dir and help functions.

If we want to import the module urllib, which enables us to read data from URLs, but we don’t know which function you want to use, we can look for which functions are implemented in this module by using the dir function:

>>> import urllib

>>> dir(urllib)

['ContentTooShortError', 'FancyURLopener', 'MAXFTPCACHE', 'URLopener', '\_\_all\_\_', '\_\_builtins\_\_',

'\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_version\_\_', '\_ftperrors', '\_get\_proxies',

'\_get\_proxy\_settings', '\_have\_ssl', '\_hexdig', '\_hextochr', '\_hostprog', '\_is\_unicode', '\_localhost', '\_noheaders', '\_nportprog', '\_passwdprog', '\_portprog', '\_queryprog', '\_safe\_map', '\_safe\_quoters', '\_tagprog', '\_thishost', '\_typeprog', '\_urlopener', '\_userprog', '\_valueprog', 'addbase', 'addclosehook', 'addinfo', 'addinfourl', 'always\_safe', 'basejoin', 'c', 'ftpcache', 'ftperrors', 'ftpwrapper', 'getproxies', 'getproxies\_environment', 'getproxies\_macosx\_sysconf', 'i', 'localhost', 'main', 'noheaders', 'os', 'pathname2url', 'proxy\_bypass', 'proxy\_bypass\_environment', 'proxy\_bypass\_macosx\_sysconf', 'quote', 'quote\_plus', 'reporthook', 'socket', 'splitattr', 'splithost', 'splitnport', 'splitpasswd', 'splitport', 'splitquery', 'splittag', 'splittype', 'splituser', 'splitvalue', 'ssl', 'string', 'sys', 'test', 'test1', 'thishost', 'time', 'toBytes', 'unquote', 'unquote\_plus', 'unwrap', 'url2pathname', 'urlcleanup', 'urlencode', 'urlopen', 'urlretrieve']

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter:

help(urllib.urlopen)

# Packages

Packages are *namespaces* which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which MUST contain a special file called \_\_init\_\_.py. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the \_\_init\_\_.py file inside the foo directory.

To use the module bar from package foo, we can import it in two ways:

# way 1

import foo.bar

# way 2

from foo import bar

In the first method, we must use the foo. prefix whenever we access the module bar (foo.bar.function()). In the second method, we don't (bar.function()), because we import the module to our module's namespace.

The \_init\_\_.py file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the \_\_all\_ variable, like so:

\_\_init\_\_.py:

\_all\_ = ["bar"]

# Memory Management

Standard CPython's garbage collector has two components: the **reference counting collector** and the generational **garbage collector** (known as gc module).

Why two? The reference counting algorithm is incredibly efficient and straightforward, but it cannot detect reference cycles. So Python has a GC which specifically deals with reference cycles. The reference counting module is fundamental and can't be disabled, whereas the **GC is optional and can be disabled manually (by default, it's enabled)**.

## First Thing To Know

Unlike many other languages, Python does not necessarily release the memory back to the OS. Instead, it creates memory pools for allocating to objects, and cannot return a pool of memory to the OS unless the entire pool is empty and unfragmented. **In many cases, all allocated memory could be released only when Python process terminates**. So it may look as the Python process uses a lot more virtual memory than it actually uses.

**If a long-running Python process takes more memory over time, it does not necessarily mean that you have memory leaks.**

The only really reliable way to ensure that a large use of memory DOES return all resources to the system is to have that use happen in a sub-process, which does the memory-hungry work then terminates. You can use the multiprocessing module and kill the process after you are done and create another one. This way you free the memory by force and OS must free up the memory used by that child process.

## Reference Counting

Every variable in Python is a reference (a pointer) to an object and not the actual value itself. For example, the assignment statement just adds a new reference to the right-hand side.

To keep track of references, every object (even integer) has an extra field called reference count that is increased or decreased when a pointer to the object is created or deleted. **When this count reaches zero, the memory occupied by the object is released**.

If an object contains references to other objects, then their reference count is automatically decremented too. Thus other objects may be de-allocated in turn. For example, when a list is deleted, the reference count for all its items is decreased. If another variable references an item in a list, the item won't be de-allocated.

Variables, which are global (declared outside of functions and classes), live until the end of the process. Thus, the reference count of objects referred by global variables never drops to zero.

Variables, which are local (defined inside functions or classes), are destroyed (along with their references) when exiting from their block.

Let’s take a look at a brief code example to demonstrate how reference counting works:

import sys

foo = []

# 2 references to the list object []: 1 from the foo var and 1 from getrefcount

print(sys.getrefcount(foo))

def bar(a):

    # 4 references to the list object []: from the foo var, function argument, getrefcount and function stack

    print(sys.getrefcount(a))

bar(foo)

# Back to 2 references to the list object [] because the function scope was destroyed

print(sys.getrefcount(foo))

### The 'del' Statement

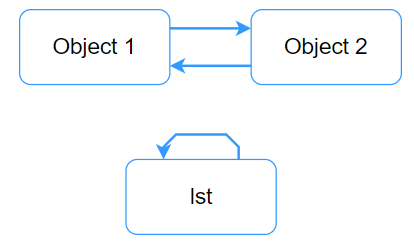
The del statement is used to remove a variable (not the object itself) and decrease its reference by 1.

Note:

* Because of being removed, after the del statement, any attempt to access the variable causes error "*… is not defined*".
* The del statement only removes the binding between that identifier and the object. It **does not actually result in the object being deleted from memory, and it doesn't call the** [\_\_del\_\_](#_Destructor) **destructor either**. It merely dereferences that local binding - this results in the ref count being decremented by 1. Of course, if the ref count reached zero due to the del, then it'll be freed (and \_\_del\_\_ will be called).
* In fact, doing 'del <variable>' is not only pointless, it's considered BAD style.

### Reference Cycles

Reference counting has a fundamental *problem* — it cannot detect reference cycles. A reference cycle occurs when one or more objects are referencing each other. Here are two examples:



1. Object 1 and object 2 are pointing to each other.

2. The 'lst' object is pointing to itself.

🡪 The reference count for such objects is always at least 1.

**How to find reference cycles?**

Reference cycles can only occur in container objects (i.e., in objects that can contain other objects), such as lists, dictionaries, classes, tuples. Garbage collector does not track all immutable types except for tuples. Tuples and dictionaries containing only immutable objects can also be untracked depending on certain conditions. Thus, the reference counting technique handles all non-circular references.

**How to avoid reference cycles?**

To avoid circular references, you can use weak references which are implemented in the weakref module. Unlike the usual references, the weakref.ref doesn't increase the reference count and returns None if an object was destroyed.

## Garbage Collector

import sys

import gc

import ctypes

# We use ctypes module to access our unreachable objects by memory address

# This trick helps us check ref count of ojects after being removed by the 'del' statment

class PyObject(ctypes.Structure):

    \_fields\_ = [("refcnt", ctypes.c\_long)]

# Ref cycle because the 'lst' object is pointing to itself

lst = []

lst.append(lst)

lst\_address = id(lst)

print('ref count of lst is', PyObject.from\_address(lst\_address).refcnt)

del lst

print('ref count of lst is', PyObject.from\_address(lst\_address).refcnt)

# Ref cycle because object 1 and object 2 are pointing to each other

object\_1 = {}

object\_2 = {}

object\_1['obj2'] = object\_2

object\_2['obj1'] = object\_1

obj\_address = id(object\_1)

print('ref count of obj is', PyObject.from\_address(obj\_address).refcnt)

del object\_1, object\_2

print('ref count of obj is', PyObject.from\_address(obj\_address).refcnt)

# Thanks to GC, ref cycles are resolved

gc.collect()

print('After gc collected')

print('ref count of lst is', PyObject.from\_address(lst\_address).refcnt)

print('ref count of obj is', PyObject.from\_address(obj\_address).refcnt)

Output:

ref count of lst is 2

ref count of lst is 1

ref count of obj is 2

ref count of obj is 1

After gc collected

ref count of lst is 0

ref count of obj is 0

As you see, the GC helped resolve the problem of reference counting. Of course, you can use weakref module to do prevent reference cycles right from the beginning, but GC seems to be an easier solution.

**When does the generational GC trigger?**

Unlike reference counting, GC does not work in real-time but runs periodically. To reduce the frequency of GC calls and pauses, the GC is threshold-based. This means the **GC will happen only when some object thresholds are met. An explicit call to gc.collect() is just an immediate way to release the memory**. However, note that this function doesn't always return the memory to the OS (as mentioned in [this section](#_First_Thing_To)).

The standard threshold values are set to (700, 10, 10) respectively, but you can adjust them using the gc.get\_threshold() function.

**How to use GC?**

The GC can be disabled by calling gc.disable(). To manually run the collection process, you need to use gc.collect().

Also, the gc module provides a lot of useful helpers that can help in debugging. If you set debugging flags to DEBUG\_SAVEALL, all unreachable objects found will be appended to gc.garbage list.

## How to De-Allocate Unmanaged Resources?

By unmanaged resources, we mean file reading/writing, socket connections, caches flushing, etc.

### Using the '\_\_del\_\_' destructor

The [\_\_del\_\_](#_Destructor) can be useful. For how to use it, check [here](https://eli.thegreenplace.net/2009/06/12/safely-using-destructors-in-python/).

### Using an explicit 'close()' method

This is generally a better approach than using \_\_del\_\_. We will define an explicit close() method which then calls the \_\_del\_\_. You call the close() whenever you're done with the resources. It can then remove attributes that refer to sub-objects.

### Using the 'with' statement

The problem with using an explicit close() statement is that you have to worry about people forgetting to call it or forgetting to place it in a try/finally block to prevent a resource leak when an exception occurs.

The with statement (similar to the using statement in C#) ensures that a resource is cleaned up when the code using it finishes running, even if exceptions are thrown. It provides 'syntactic sugar' for try/finally blocks.

**For built-in libraries**

Python provides some built-in libraries which already implemented to be able to be used with the with statement.

For example, to open a file, process its contents, then make sure to close it, you should do:

# Way 1: recommended

with open("file-name.txt") as file:

    data = file.read()

    # Do something with data

    # Note: no need to close the file

# because "with" ensures the file will be closed when the block is exited.

# Way 2: avoid

file = open("file-name.txt", "wb")

# Do something with data

file.close()   # This is not exception safe

**For your own code**

To implement your own with statement, create a class with the following methods:

class Package:

    def \_\_init\_\_(self):

        self.files = []

    def \_\_enter\_\_(self):

        return self

    def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

        for file in self.files:

            os.unlink(file)

Then, when someone wanted to use your class, they'd do the following:

with Package() as package:

    # use package

The package is an instance of type Package (it's the value returned by the \_\_enter\_\_ method). The \_\_exit\_\_ method will automatically be called, even when an exception occurs.

You could even take this approach a step further. In the example above, someone could still instantiate Package using its constructor without using the with clause. You don't want that to happen. You can fix this by creating a PackageResource class that defines the \_\_enter\_\_ and \_\_exit\_\_ methods. Then, the Package class would be defined strictly inside the \_\_enter\_\_method and returned. That way, the caller never could instantiate the Package class without using a with statement:

class PackageResource:

    def \_\_enter\_\_(self):

        class Package:

            ...

        self.package = Package()

        return self.package

    def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

        self.package.cleanup()

You'd use this as follows:

with PackageResource() as package:

    # use package

## How to Reduce Object Size?

<https://habr.com/en/post/458518/>

## Memory Profiling Tools

<https://pypi.org/project/memory-profiler/>

# Multi-Threading

## Global Interpreter Lock (GIL)

### What Is GIL?

When two threads try to modify the same resource at the same time, they can step on each other’s toes. The end result can be a garbled mess where neither of the threads ends up with what they wanted.

For example, we know that Python uses reference counting to manage memory de-allocation of variables. These variables need protection from race conditions where two threads increase or decrease its value simultaneously. If this happens, it can cause either leaked memory which is never released or, even worse, incorrectly release the memory while a reference to that object still exists. The consequence is "weird" bugs in your programs.

The GIL is a solution to the common problem of dealing with shared resources in multi-threaded programs. It accomplishes this by **locking the entire interpreter, meaning that it’s not possible for another thread to step on the current one**. When CPython handles memory, it uses the GIL to ensure that it does so safely.

So in simple words, GIL is a mutex.

*You might not know!*

- There are pros and cons to this approach, and the GIL is heavily debated in the Python community (because CPU nowadays has more than one core).

- **GIL is available in CPython only**. So if you're using other Python interpreter implementations, you won't have it.

### Pros of GIL

Shared resource can be kept safe by added locks across threads so that they are not modified inconsistently. But adding a lock to each object or groups of objects means multiple locks will exist – which can cause another problem—deadlocks (deadlocks can only happen if there is more than one lock). Another side effect would be decreased performance caused by the repeated acquisition and release of locks.

The GIL is a single lock on the interpreter itself which adds a rule that execution of any Python bytecode requires acquiring the interpreter lock. This **prevents deadlocks** (as there is only one lock) and doesn’t introduce much performance overhead. To simply put, the GIL **helps single-threaded program run faster** as only one lock needs to be managed.

### Cons of GIL

When you look at a typical Python program – or any computer program – there’s a difference between those that are CPU-bound in their performance and those that are I/O-bound:

* CPU-bound programs push the CPU to its limit. They usually do mathematical computations like matrix multiplications, searching, image processing, etc.
* I/O-bound programs spend time waiting for input/output which can come from a user, file, database, network, etc. They sometimes have to wait for a significant amount of time till they get what they need from the source because the source may need to do its own processing before the input/output is ready. For example, a user thinking about what to enter into an input prompt or a database query running in its own process.

Let’s have a look at a simple CPU-bound program that performs a countdown:

# single\_threaded.py

import time

from threading import Thread

COUNT = 50000000

def countdown(n):

    while n>0:

        n -= 1

start = time.time()

countdown(COUNT)

end = time.time()

print('Time taken in seconds -', end - start)

Running this code on my system with 4 cores gave the following output:

Time taken in seconds - 6.20024037361145

Now modify the code a bit to do to the same countdown using **two threads in parallel**:

# multi\_threaded.py

import time

from threading import Thread

COUNT = 50000000

def countdown(n):

    while n>0:

        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))

t2 = Thread(target=countdown, args=(COUNT//2,))

start = time.time()

t1.start()

t2.start()

t1.join()

t2.join()

end = time.time()

print('Time taken in seconds -', end - start)

And when I ran it again:

Time taken in seconds - 6.924342632293701

As you can see, **both versions take almost same amount of time to finish** (in fact, the multi-thread version runs even slower than the single-threaded one).

The GIL does not have much impact on the performance of I/O-bound multi-threaded programs as the lock is shared between threads while they are waiting for I/O.

**But a program whose threads are entirely CPU-bound would not only become single threaded due to the lock but will also see an increase in execution time compared to where it was written to be entirely single-threaded**. This increase is the result of acquire and release overheads added by the lock.

As seen in the above example, in the multi-threaded version, the GIL prevented the CPU-bound threads from executing in parallel.

### How to Deal with GIL?

If the GIL is causing you problems, here a few approaches you can try:

**Multi-processing instead of multi-threading**

The most popular way is to use a multi-processing approach where you use multiple processes instead of threads. Each Python process gets its own Python interpreter and memory space so the GIL won’t be a problem. Python has a multiprocessing module which lets us create processes easily like this:

from multiprocessing import Pool

import time

COUNT = 50000000

def countdown(n):

    while n>0:

        n -= 1

if \_\_name\_\_ == '\_\_main\_\_':

    pool = Pool(processes=2)

    start = time.time()

    r1 = pool.apply\_async(countdown, [COUNT//2])

    r2 = pool.apply\_async(countdown, [COUNT//2])

    pool.close()

    pool.join()

    end = time.time()

    print('Time taken in seconds -', end - start)

Running this on my system gave this output:

Time taken in seconds - 4.060242414474487

Now the execution time is significantly decreased, but it didn’t drop to half. That's because process management has its own overheads. Multiple processes are heavier than multiple threads, so this could become a scaling bottleneck.

**Alternative Python interpreters**

Python has multiple interpreter implementations: CPython, Jython, IronPython and PyPy. GIL exists only in the default Python implementation – CPython. If your program, with its libraries, is available for one of the other implementations then you can try them out as well.

**Just wait it out**

While many Python users take advantage of the single-threaded performance benefits of GIL. The multi-threading programmers don’t have to fret as some of the brightest minds in the Python community are working to remove the GIL from CPython. One such attempt is known as the Gilectomy.

## The 'threading' Package

### Terminate Thread

The rule of thumb is: don't kill threads (note that in some environments this may not even be possible, e.g. standard C++11 threads). Let the thread fetch the information (listen to signals you send) and terminate itself. Controlling threads from other threads makes code vulnerable to bugs and hard to maintain.

**The 'join()' method**

Thread cleans up the underlying resources by itself. Thread.join() merely waits for the thread to end and does not perform cleanup. Basically, each Thread has a lock that is released when the thread is done and subsequently cleaned up. Thread.join() just waits for the lock to be released, then it sets a flag to mark the thread as dead. This is an optimization to avoid needlessly waiting for the lock. These are internal, however, and also performed by all other public methods relying on the lock and flag.

For example, the below code help terminates the worker thread. Without calling join() after setting the thread to terminated state, Thread.is\_alive() will not return to False.

import threading

import time

IS\_FINISH = False

def run():

    while True:

        print("worker thread is running")

        if IS\_FINISH:

            break

        time.sleep(5)

def main():

    print("main thread starts")

    t = threading.Thread(target=run)

    t.start()

    time.sleep(5)

    print("ready to terminate worker thread")

    global IS\_FINISH

    IS\_FINISH = True

    t.join() # If we replace this line by 'time.sleep(1)', we'll get the same output

    print("is worker thread alive? " + str(t.is\_alive()))

    # t.start() # But re-start the thread causes error "threads can only be started once"

# We have to create a new instance for the thread to start it again.

if \_\_name\_\_ == '\_\_main\_\_':

    main()

Output:

main thread starts

worker thread is running

ready to terminate worker thread

worker thread is running

is worker thread alive? False

**Why threading.Thread object has 'start', but no 'stop' method?**

<https://stackoverflow.com/a/14482304>

**More ways to terminate threads**

<https://www.geeksforgeeks.org/python-different-ways-to-kill-a-thread/>

<https://cuyu.github.io/python/2016/08/15/Terminate-multiprocess-in-Python-correctly-and-gracefully>

# Exception Handling

Exceptions are around programmers. But sometimes you don't want exceptions to completely stop the program. You might want to do something special when an exception is raised. This is done in a try / except block.

For example:

def do\_stuff\_with\_number(n):

print(n)

def catch\_this():

list = (1, 2, 3)

for i in range(5):

try:

do\_stuff\_with\_number(the\_list[i])

# Raised when accessing a non-existing index of a list

except IndexError:

do\_stuff\_with\_number(0)

catch\_this()

Output:

1

2

3

0

0

Tip: For how to get exception message, check [here](https://stackoverflow.com/a/33239954).

# Logging

# Command Line Arguments

<https://stackabuse.com/command-line-arguments-in-python/>

# Virtual Environment ('virtualenv' Package)

## Steps to Set Up a Virtual Environment

1. Create a new folder which stores the virtual environment (VE) for your project (for example: vir\_env)

2. Activate that VE by running following commands:

cd vir\_env

virtualenv .

Scripts\activate

3. Install packages for that VE using pip (just like normal way)

More details:

<https://packaging.python.org/tutorials/installing-packages/#creating-virtual-environments>

<https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/>

## How to Use VE with VSCode

<https://code.visualstudio.com/docs/python/environments#_select-and-activate-an-environment>

# Run Python Script in Background

No console window appears: <https://stackoverflow.com/a/9706045>

# Create Executable From Python Script

## pyinstaller

<https://pyinstaller.readthedocs.io/en/stable/>

<http://htmlpreview.github.io/?https://github.com/pyinstaller/pyinstaller/blob/v2.0/doc/Manual.html>

The pyinstaller module allows us to **pack the Python script, along with the Python interpreter and imported packages/modules** used by the script, into a single executable file (.exe in Windows). When the executable is run, it is first "unfolded" into a temporary site and then runs things 'normally'.

The ultimate goal is to help the Python code run on any PC without Python installed and packages downloaded.

**Tips**:

* Add option -F or --onefile to put all the needed files and libraries into a **single executable file**.
* Add option --noconsole to **hide the console** when the executable is launched.
* To **set icon** for the executable, add option '--icon=path/to/.ico/file'. (Both / and \ are acceptable).
* **The Python interpreter and all imported packages/modules (used by your script) are included in the executable file**. By default, these are gotten from the system directories where Python and its packages are installed (e.g., C:\Users\<user-name>\AppData\Local\Programs\Python in Windows).

For some reasons (I don't know why), the executable file created with this default setting has bigger size than what it's supposed to have (only imported packages/modules, which your script uses, are included).

A simple way to significantly **reduce the executable file size is using** [**virtualenv**](#_Install_Packages_and) with PyInstaller installed and used.

For even a smaller size, we can:

* [Exclude certain packages/modules](https://stackoverflow.com/questions/4890159/python-excluding-modules-pyinstaller) (e.g. pyinstaller, pip, wheel, etc.).
* Compress the executable with [UPX](https://upx.github.io/) by adding option '--upx-dir <path/to/upx-dir/not/upx.exe-dir' option.

**Anyway, we should note that executable file created by PyInstaller is generally large in size** (a simple 'Hello, World' can take up ~6MB). That's the weakness of PyInstaller.

* By default, PyInstaller **sets its log level** as INFO. To change log level (DEBUG, INFO, WARN, ERROR, CRITICAL, etc.), specify option --log-level=<LOGLEVEL>. The more details, the more info is shown when running PyInstaller.

**Notes**:

* For resource files (configurations, images, etc.) or runtime libraries (.dll, .so, etc.), we usually hard-code their relative paths. Although the scripts work fine this way, the generated exe file (generated by PyInstaller) might not. That's because the current directory might be changed (the script and the exe file don't share the same directory). In this case, we **have to change paths of resource files or runtime libraries** (in code) corresponding to the exe file location to make thing work fine again.

Another way is to include all resource files and libraries into the executable (when it’s created) using option. On Windows, it is:

'--add-data <path/to/lib/file/or/lib/dir>;<path/to/lib/dir/in/exe/file>'. (note: the first path can be either a file path or a dir path; the second one must be a dir path). More details [here](https://pyinstaller.readthedocs.io/en/stable/spec-files.html#adding-data-files).

## nuitka

<http://nuitka.net/doc/user-manual.html>

<https://dev.to/k4ml/python-compile-standalone-executable-with-nuitka-1ml1>

The nuitka module is like a Python compiler and a replacement to the Python interpreter. It **translates the Python script into C language**, then compiles the C code into binary (gcc compiler is required), then uses libpythonXX (pythonXX.dll) to execute in the same way as CPython does (meaning executing uncompiled code and compiled code together in a compatible way).

The ultimate goal is to help the Python code run on any PC without Python installed and packages downloaded, while at the same time, improve program performance by taking advantages of compiled code.

**Tips:**

* Add option --standalone to create **executable file** Which can run on any PC without Python installed and packages downloaded.
* Add option --windows-disable-console to **hide the console** when the executable is launched.
* To use with nuitka with virtualenv, have to install nuitka on that VE. And when running the nuitka command, have to specify the path to python.exe of that VE. Else the system python.exe's path will be used.

## pyinstaller vs nuitka

|  |  |  |
| --- | --- | --- |
|  | **pyinstaller** | **nuitka** |
| Principle | Python script + Python interpreter + imported packages/modules = executable | Binary generated from Python script and imported packages/modules + libpythonXX = executable |
| Ease of use | Easy to use.  Might face some errors but quite easy to resolve. | Harder to use.  Have chance of getting errors which are difficult to resolve. |
| Executable file size | Smaller | Larger |
| Executable generation speed | Quick | Long |

Comparing some ways to generating executable file from Python script: <https://tryexceptpass.org/article/package-python-as-executable/>

# Cython

## What Is Cython?

<https://pythonprogramming.net/introduction-and-basics-cython-tutorial/>

<https://towardsdatascience.com/use-cython-to-get-more-than-30x-speedup-on-your-python-code-f6cb337919b6>

<https://en.wikipedia.org/wiki/Cython>

## Notes When Using Cython

***What is Cython best used for?***

It typically used to generate CPython extension modules which can be loaded and used by regular Python code using the import statement, but with significantly less computational overhead at run time. It also facilitates wrapping independent C or C++ code into Python-importable modules.

For how to do that, check <https://cython.readthedocs.io/en/latest/index.html>.

***Can we create a Cython program which is fully standalone without the Python interpreter?***

We can, but … A Cython program will depend on the Python interpreter and standard libraries. We cannot avoid this. **There are ways we can bundle these with our Cython program; however, this is doing exactly the same thing as PyInstaller** and will give us a file which is just as big. At the end, we could should just use PyInstaller (much easier than doing it with Cython) and accept the large file.

***Can we compile multiple source files into a single executable file?***

Cython is mostly designed to make compiled Python-importable modules instead of executable files (though it's possible), and it isn't really designed to compile multiple source files to one object (though it is possible with a lot of work). You can do what you want but it isn't well supported. More details [here](https://stackoverflow.com/a/59389683).

A trick to compile multiple source file into a single executable file is:

1. Merge all of the source files into one .py file. Then edit the merged code a bit (removing duplicate import statement, renaming duplicate constants/variables, etc.) and make sure it runs correctly.
2. Use Cython to convert the .py or .pyx files into C files. If the converting process fails, check [this guide](https://stackoverflow.com/a/56032204).
3. Use gcc (in Windows, install Mingw-w64), or MSVC, to compile C files and build the executable. Check [this guide](http://masnun.rocks/2016/10/01/creating-an-executable-file-using-cython/) and [this guide](https://stackoverflow.com/a/22513682) for more details.

# Web Scrapping

**Available tools:**

<https://www.scrapestorm.com/>

<http://www.botchief.com/>

## Downloading Web Page ('request' Package)

General guide: <https://docs.python-guide.org/scenarios/scrape/>

Get Unicode text: <https://stackoverflow.com/a/12843406>

## Parsing HTML

### 'BeautifulSoup' Package

Basic Xpath guide: <https://www.guru99.com/xpath-selenium.html>

### 'html' Module of 'lxml' Package

## Converting RST to HTML ('docutils.core' Module)

<https://kite.com/python/examples/3734/docutils-convert-a-restructuredtext-file-to-html>

## Controlling Browsers ('selenium' Package)

<https://selenium-python.readthedocs.io/>

### Tips

* Run selenium in backgroud: <https://stackoverflow.com/a/43543955>

### Experiences

**With selenium, no more need of 'pyautogui' for UI automation of websites**

Selenium can do almost things that pyautogui can. Most importantly, it allows running in background; while with pyautogui, we have the left our computer alone.

To perform a keystroke, we can use driver.find\_element\_by\*\*\*.send\_keys(). If we don’t have any element which we'll send the key to, use ActionChains class which not only allows keystrokes but also mouse movement and clicking actions.

For example:

# Cick on the page body, then blackout all text and copy it

driver.find\_element\_by\_tag\_name('body').send\_keys(Keys.CONTROL + "a" + "c")

# Paste text (use ActionChains because we don't find any element which we'll send the key to)

from selenium.webdriver.common.action\_chains import ActionChains

ActionChains(driver).key\_down(Keys.CONTROL).send\_keys("v").key\_up(Keys.CONTROL).perform()

# Note: Rember to key\_up after each key\_down. Else, later keystrokes will be affected.

**Open in new tab**

Avoid using commands which simulate "ctrl+tab" to open a URL in a new tab in the browser (either Chrome and Firefox). That’s because this command is completely useless (that seem a limitation of Selenium webdriver when handling multiple tab handlers).

To open a new tab in Chrome or Firefox, add:

driver.execute\_script("window.open('');") # Open in new tab

driver.switch\_to.window(driver.window\_handles[-1]) # Switch to the last opened tab handle

driver.get('your-url') # Now you can go to the URL you want

**Use cookies of browses for webdriver**

There will be many cases when you want to apply cookies for webdrivers (either Chrome or Firefox). For example, log in Facebook with saved profile info (including username and password).

To use cookies in Chrome driver, use:

profile\_path = os.getenv('LOCALAPPDATA') + "\\Google\\Chrome\\User Data"

options.add\_argument("user-data-dir=" + profile\_path")

driver = webdriver.Chrome(executable\_path="path-to-chromedriver.exe", options=options)

Note: The steps to use cookies for Firefox driver is much more complicated than Chrome driver.

**Start webdriver with full window size**

Extremely easy, just need to add:

options.add\_argument("start-maximized")

driver = webdriver.Chrome(executable\_path=<path-to-chromedriver.exe>, options=options)

**Keep Chrome driver opening after the script is completed**

The Chrome driver will be automatically terminated after the scripts run to the end. So, to keep it (also its window) opening until you want to close, add:

options.add\_experimental\_option("detach", True)

driver = webdriver.Chrome(executable\_path=<path-to-chromedriver.exe>, options=options)

Note: Firefox driver already keeps opening its process and window, so no need to add anything.

# UI Automation

## Full UI Automation

### 'pyautogui' Package

<https://pyautogui.readthedocs.io/en/latest/>

Some personal experiences when using pyautogui are as below:

**#1. Shortcut keystrokes might not work**

# Black out all text then copy

pyautogui.hotkey('ctrl', 'a', 'c', interval=0.2)

# Note: Following codes do NOT work

# pyautogui.hotkey('ctrl', 'a', interval=0.2)

# pyautogui.hotkey('ctrl', 'c', interval=0.2)

**#2. Thinking of keystrokes before any mouse action**

In many cases, all we need is just a shortcut keystroke. For example, to input text on the Status box of Facebook, we don't have to find the element then click on it then input the text. We just need to press "p" and that’s all.

## Keyboard and Mouse Controlling and Monitoring

### 'keyboard' and 'mouse' Packages

<https://pypi.org/project/keyboard/>

<https://pypi.org/project/mouse/>

### 'pynput' Package

<https://pynput.readthedocs.io/en/latest/>

## Getting Control Attributes from Current Process or Other Processes in Windows

|  |  |  |
| --- | --- | --- |
| **Libs** | **Pros** | **Cons** |
| Win32 | Use internal Win32 dlls available in the Window. | Many controls (menu, menu item, etc.) cannot be gotten. |
| For how to get control handler from other processes in Window, check [here](https://www.codeproject.com/Articles/34752/Control-in-Focus-in-Other-Processes).  For how to get text from controls, check [here](https://stackoverflow.com/a/18698). | |
| UIAutomation | Can get attributes from most of the standard controls. Also from UI using non-system controls - including WPF, text in IE and Firefox, etc. | Use .NET dlls. So having to import them to Python. |
|  | For how to use, check [here](https://stackoverflow.com/a/11042272).  For full [docs](http://msdn.microsoft.com/en-us/library/ms747327.aspx), check here.  For how to use .NET dlls in Python, check [this section](#_.NET_Wrapper_for). | |

# UI Applications

<https://stackoverflow.com/questions/19584076/wxpython-vs-pyqt-vs-pygtk-when-and-what-to-use>

## Choices

### 'Tkinter' Built-In Package

<https://tkdocs.com/tutorial/index.html>

<https://www.python-course.eu/python_tkinter.php>

<http://zetcode.com/ebooks/tkinter/>

Others:

* Color chart: [here](http://www.science.smith.edu/dftwiki/images/3/3d/TkInterColorCharts.png).
* **Ttkinter** vs. Tkinter: [here](https://stackoverflow.com/a/19564114) (for a native, modern UI looking by default, use Ttkinter).
* Widget grouping: [here](https://runestone.academy/runestone/books/published/thinkcspy/GUIandEventDrivenProgramming/05_widget_grouping.html).

### 'PyQt' Package

<https://pypi.org/project/PyQt5/>

### 'PySide' Package (or 'Qt for Python')

<https://doc.qt.io/qtforpython/api.html>

### 'wxPython' Package

<https://docs.wxpython.org/>

## More Features

### Create System Tray Icon for App

|  |  |  |
| --- | --- | --- |
| **Package** | **Pros** | **Cons** |
| [pystray](https://pystray.readthedocs.io/en/latest/index.html) | Support menu updating (update icon for checkmark, enable/disable menu items, change menu item label dramatically, etc.). | - Have to configure multithreading manually (when started, pystray runs in a separate un-parallel worker thread, so we have to configure a separate parallel thread for it so that it can run concurrently with the main thread or other threads). Check [here](https://www.tutorialspoint.com/python/python_multithreading.htm).  - Cannot know how to add separator for item groups. |
| [infi.systray](https://github.com/Infinidat/infi.systray) | - Easier to use.  - Support multithreading by default. | - Doesn't support menu icon updating (checkmark, etc.)  - Doesn't support [pyinstaller](#_Create_Executable_Python). Fix [here](https://www.reddit.com/r/learnpython/comments/ex44mj/how_to_create_an_executable_python_file_in_windows/fg6n7b9/). |

# Win32 Wrapper for Python (win32api)

For how to install: <https://stackoverflow.com/a/43880051>

For how to import: <https://stackoverflow.com/a/46553507>

# .NET Wrapper for Python (pythonnet)

Docs: <https://pypi.org/project/pythonnet/>

**Note**: For some libs, have to copy the DLLs (from C:\Windows\Microsoft.NET\assemply) to the project folder and specify their path using sys.path.append() method. For how, check [here](https://stackoverflow.com/questions/13259617/python-for-net-unable-to-find-assembly-error).

# Language Translation

|  |  |  |
| --- | --- | --- |
| **Package** | **Pros** | **Cons** |
| [translate](https://pypi.org/project/translate/) | Closest result (some are the same, some are similar) as Google Translate website (although it uses [MyMemory](https://mymemory.translated.net/) provider, not Google). | - New line feeds are ignored, causing merging paragraphs.  - Error when parsing special characters (', ", <, >, etc.).  - Poor feature supports. |
| [googletrans](https://pypi.org/project/googletrans/) | - No error when parsing special characters (', ", etc.).  - Rich feature support (pronunciation, customized service URL, concurrent translation querying). | - Different result from Google Translate website.  - New line feeds are ignored, causing merging paragraphs. |
| [goslate](https://pypi.org/project/goslate/) | - No error when parsing special characters (', ", etc.).  - Richest feature support (pronunciation, customized service URL, concurrent translation querying, bypassing text length limitation by Google, fast translation speed, proxy support). | - Different result from Google Translate website.  - New line feeds are ignored, causing merging paragraphs. |

**Test code** (note: only run one package each time to prevent "429 – Too many requests" error):

text = "改行コードを終端文字列に変更"

dest = 'en'

# Result from Google Translat website: Change line feed code to end character string

import translate

trans = translate.Translator(to\_lang=dest, from\_lang='autodetect')

print(trans.translate(text))

# Change line feed code to end character string

# Note: Same as Google Translate websie, but translation speed is pretty slow

import googletrans

gt = googletrans.Translator()

print(gt.translate(text, dest=dest, src='auto').text)

# Change the new line code to-terminated string

# Note: Translation speed is pretty slow

import goslate

gs = goslate.Goslate()

print(gs.translate(text, dest, source\_language='auto'))

# Change the new line code to-terminated string

# Note: Translation speed is very fast

Other text to test:

|  |
| --- |
| Japanese version |
| 名 称 ： SChk\_JianKeyKihonOK  概 要 ： 事案キーチェック処理(事案基本情報更新ＯＫ通知)  引 数 ： (i)pJusinDat 受信データ  (i)pJiannKey 事案キー  関数値 ： 0 送信時の事案キーと同じ  -1 送信時の事案キーと違う |
| 中国政府は外国から新型ウイルスが運び込まれるのを防ぐため、全ての外国人の入国を一時的に禁止した。ビザ（査証）や居留許可を持っている人も入国を認めていない。  また、外国の航空会社による中国への運航を週1便に制限するとともに、中国の航空会社による外国への運航も週1便に制限している。搭乗率は75％を超えてはならない。 |