Modules in Python

Aka libraries in C++.

The build-in module is “Python standard library”. Just like libraries in C++, there exists a LOT of modules in Python, much more than C++.

import math

import sys # https://docs.python.org/3/library/sys.html

and the code below is identical to the one above

import math, sys

Namespace ([Namespace - Wikipedia](https://en.wikipedia.org/wiki/Namespace)):

A namespace is a space (understood in a non-physical context) in which some names exist and the names don’t conflict with each other. Inside each namespace, each name must remain **unique**. If the module is valid (exists and accessible), Python imports its contents, all the names defined in the module become known, but they don’t enter the code’s namespace.

Photo: Geeks for geeks

This program shows that two namespaces (local and math module) can coexist.

import math

pi = 3.14

print(math.pi) # this is pi from the math module

print(pi) # this is local variable pi

# output:

# 3.141592653589793

# 3.14

This imports the name / list (if there are more than one argument) into the namespace. The names of the imported entities are accessible without qualification.

from math import pi

print(pi) # 3.141592653589793

Multiple names imported

from math import sin, pi

print(sin(pi / 2)) # 1.0

The imported names supersede the local ones.

pi = 3.14

def sin(x):

    if 2 \* x == pi:

        return 0.99999999

    else:

        return None

print(sin(pi / 2)) # local variables and functions, output: 0.99999999

from math import sin, pi

print(sin(pi / 2)) # imported names, output: 1.0

This imports all entities from the indicated module

from math import \*

This may not be able to avoid name conflicts

Import a module: the “as” keyword (it can be anything you’d like)

import math as sth

print(sth.pi) # 3.141592653589793

However, after successful execution of an aliased import, the original module name becomes inaccessible and must not be used.

from module import name as

from math import pi as PI, sin as sine

print(sine(PI/2)) # 1.0

WAIIIIIIIIIIIIIIT,

does this also mean that I can import from my own codes?

YES!!

from testing\_2 import hello\_world

hello\_world(2)

# Hello World

# Hello World

Where testing\_2.py is created within the same directory, its program is as followed.

def hello\_world (n: int):

    for i in range(n):

        print("Hello World")

**We will dig deeper into this in “Packages in Python”.**

The dir() function

This function returns an alphabetically sorted list containing all the entities’ names in the module.

import math

for name in dir(math):

    print(name, end = " ")

# \_\_doc\_\_ \_\_loader\_\_ \_\_name\_\_ \_\_package\_\_ \_\_spec\_\_ acos acosh

# asin asinh atan atan2 atanh ceil comb copysign cos cosh

# degrees dist e erf erfc exp expm1 fabs factorial floor

# fmod frexp fsum gamma gcd hypot inf isclose isfinite

# isinf isnan isqrt lcm ldexp lgamma log log10 log1p

# log2 modf nan nextafter perm pi pow prod radians

# remainder sin sinh sqrt tan tanh tau trunc ulp

Math module was briefly introduced before, another one worth mentioning is random

It delivers some mechanisms allowing people to operate with pseudorandom numbers, pseudo- means fake.

So, how are “random” numbers generated? They are all calculated using very refined algorithms, they are deterministic and predictable.

A random number generator tasks a value “seed”, calculates the next “random” value and replace it as the “seed” value.

So, this is a cycle after all? The answer is positive, but it may be very long.

But how is the initial “seed” value decided? It is augmented by setting the seed with a number taken from the current time.

#include <bits/stdc++.h>

using namespace std;

int main () {

    srand(time(NULL));

    cout << RAND\_MAX << "\n";

    cout << rand() << '\n';

}

Here is a simple C++ sample of random number generator, the 4th line resets the seed value (as C++ doesn’t reset it automatically).

The seed() function allows the programmer to reset the seed, either to an integer value, or to the current time.

from random import random, seed

seed(0) # sets seed to 0

seed() # sets seed to current time

for i in range(5):

    print(random())

Random integers (right-sided exclusion)

from random import randrange, randint

print(randrange(5), end=' ') # randrange(end)

print(randrange(0, 1), end=' ') # randrange(begin, end)

print(randrange(0, 5, 2), end=' ') # randrange(begin, end, step)

print(randint(0, 1)) # randint(left, right)

What about choosing from a list?

from random import choice, sample

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(choice(my\_list)) # choose a random element

# output: 7

print(sample(my\_list, 5)) # choose 5 elements randomly

# output: [5, 2, 3, 9, 10]

print(sample(my\_list, 10)) # choose 10, all

# output: [6, 9, 3, 4, 2, 10, 7, 1, 8, 5]

# samples may not be sorted

Obviously, as the sample() and choice() functions work on random algorithms, the printed result may not be the same.

Unrelated to Python:

Location of the programme.



Platform module

from platform import platform

Platform, it lets the user access the underlying platform’s data, as described above, hardware, operating system, and interpreter version information.

The **platform()** function within the platform module returns a string describing the environment, its output is addresses to humans rather than automated processing.

platform(aliased = False, terse = False)

**Aliased**, when set to True, it may cause the function to present the alternative underlying layer names instead of the common ones.

**Terse**, when set to True, it may convince the function to present a briefer from of the result, if possible (like in the case below)

from platform import platform

print(platform()) # Windows-10-10.0.19043-SP0

print(platform(False, True)) # Windows-10

print(platform(True, False)) # Windows-10-10.0.19043-SP0

print(platform(True, True)) # Windows-10

Sometimes Terse is not possible, like the case below

from platform import platform

print(platform()) # Linux-5.13.0-1017-aws-x86\_64-with-glibc2.29

print(platform(False, True)) # Linux-5.13.0-1017-aws-x86\_64-with-glibc2.29

print(platform(True, False)) # Linux-5.13.0-1017-aws-x86\_64-with-glibc2.29

print(platform(True, True)) # Linux-5.13.0-1017-aws-x86\_64-with-glibc2.29

The **machine()** function returns a string about the generic name of the processor which runs you OS together with Python.

from platform import machine

print(machine()) # AMD64

It differs by machines

from platform import machine

print(machine()) # x86\_64

The **processor()** function returns a string about the real processor name (if possible)

from platform import processor

print(processor()) # Intel64 Family 6 Model 158 Stepping 11, GenuineIntel

The **system()** function returns a string about the operating system

from platform import system

print(system()) # Linux

from platform import system

print(system()) # Windows

The **version()** function returns the OS version

from platform import version

print(version()) # #242-Ubuntu SMP Fri Apr 16 09:57:56 UTC 2021

from platform import version

print(version()) # 10.0.19043

The **python\_implementation()** and **python\_version\_tuple()** functions. The former one returns a string of the Python implementation (expect “CPython” here). The later one returns a three-element tuple, the **major** part of Python’s version, the **minor** part and the **patch** level number

from platform import python\_implementation, python\_version\_tuple

print(python\_implementation()) # Cpython

for atr in python\_version\_tuple():

    print(atr)

# 3

# 7

# 10

You can read about all the standard Python modules here:

[Python Module Index — Python 3.10.4 documentation](https://docs.python.org/3/py-modindex.html)