**Python Essentials 2:  
Module 1**

**Modules, Packages and PIP**

**What is a module?**

Computer code has a tendency to grow. We can say that code that doesn't grow is probably completely unusable or abandoned. A real, wanted, and widely used code develops continuously, as both users' demands and users' expectations develop in their own rhythms.

A code which is not able to respond to users' needs will be forgotten quickly, and instantly replaced with a new, better, and more flexible code. Be prepared for this, and never think that any of your programs is eventually completed. The completion is a transition state and usually passes quickly, after the first bug report. Python itself is a good example how the rule acts.

Growing code is in fact a growing problem. A larger code always means tougher maintenance. Searching for bugs is always easier where the code is smaller (just as finding a mechanical breakage is simpler when the machinery is simpler and smaller).

Moreover, when the code being created is expected to be really big (you can use a total number of source lines as a useful, but not very accurate, measure of a code's size) you may want (or rather, you will be forced) to divide it into many parts, implemented in parallel by a few, a dozen, several dozen, or even several hundred individual developers.

Of course, this cannot be done using one large source file, which is edited by all programmers at the same time. This will surely lead to a spectacular disaster.

If you want such a software project to be completed successfully, you have to have the means allowing you to:

* divide all the tasks among the developers;
* join all the created parts into one working whole.

For example, a certain project can be divided into two main parts:

* the user interface (the part that communicates with the user using widgets and a graphical screen)
* the logic (the part processing data and producing results)

Each of these parts can be (most likely) divided into smaller ones, and so on. Such a process is often called **decomposition**.

For example, if you were asked to arrange a wedding, you wouldn't do everything yourself - you would find a number of professionals and split the task between them all.

How do you divide a piece of software into separate but cooperating parts? This is the question. **Modules** are the answer.

# How to make use of a module?

So what is a module? The [Python Tutorial](https://docs.python.org/3/tutorial/modules.html) defines it as **a file containing Python definitions and statements**, which can be later imported and used when necessary.

The handling of modules consists of two different issues:

* the first (probably the most common) happens when you want to use an already existing module, written by someone else, or created by yourself during your work on some complex project - in this case you are the module's **user**;
* the second occurs when you want to create a brand new module, either for your own use, or to make other programmers' lives easier - you are the module's **supplier**.

Let's discuss them separately.

First of all, a module is identified by its **name**. If you want to use any module, you need to know the name. A (rather large) number of modules is delivered together with Python itself. You can think of them as a kind of "Python extra equipment".

All these modules, along with the built-in functions, form the **Python standard library** - a special sort of library where modules play the roles of books (we can even say that folders play the roles of shelves). If you want to take a look at the full list of all "volumes" collected in that library, you can find it here: <https://docs.python.org/3/library/index.html>.

Each module consists of entities (like a book consists of chapters). These entities can be functions, variables, constants, classes, and objects. If you know how to access a particular module, you can make use of any of the entities it stores.

Let's start the discussion with one of the most frequently used modules, named math. Its name speaks for itself - the module contains a rich collection of entities (not only functions) which enable a programmer to effectively implement calculations demanding the use of mathematical functions, like *sin()* or *log()*.

# Importing a module

To make a module usable, you must **import** it (think of it like of taking a book off the shelf). Importing a module is done by an instruction named import. Note: import is also a keyword (with all the consequences of this fact).

Let's assume that you want to use two entities provided by the math module:

* a symbol (constant) representing a precise (as precise as possible using double floating-point arithmetic) value of π (although using a Greek letter to name a variable is fully possible in Python, the symbol is named **pi** - it's a more convenient solution, especially for that part of the world which neither has nor is going to use a Greek keyboard)
* a function named sin() (the computer equivalent of the mathematical *sine* function)

Both these entities are available through the math module, but the way in which you can use them strongly depends on how the import has been done.

The simplest way to import a particular module is to use the import instruction as follows:

import math

The clause contains:

* the import keyword;
* the **name of the module** which is subject to import.

The instruction may be located anywhere in your code, but it must be placed **before the first use of any of the module's entities**.

If you want to (or have to) import more than one module, you can do it by repeating the import clause (preferred):

import math

import sys

or by listing the modules after the import keyword, like here:

import math, sys

The instruction imports two modules, first the one named math and then the second named sys.

The modules' list may be arbitrarily long.

To continue, you need to become familiar with an important term: **namespace**. Don't worry, we won't go into great detail - this explanation is going to be as short as possible.

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 (i.e., there are not two different objects of the same name). We can say that each social group is a namespace - the group tends to name each of its members in a unique way (e.g., parents won't give their children the same first names).

This uniqueness may be achieved in many ways, e.g., by using nicknames along with the first names (it will work inside a small group like a class in a school) or by assigning special identifiers to all members of the group (the US Social Security Number is a good example of such practice).

**Inside a certain namespace, each name must remain unique**. This may mean that some names may disappear when any other entity of an already known name enters the namespace. We'll show you how it works and how to control it, but first, let's return to imports.

If the module of a specified name **exists and is accessible** (a module is in fact a **Python source file**), Python imports its contents, i.e., **all the names defined in the module become known**, but they don't enter your code's namespace.

This means that you can have your own entities named sin or pi and they won't be affected by the import in any way.

At this point, you may be wondering how to access the pi coming from the math module.

To do this, you have to qualify the pi with the name of its original module.

Look at the snippet below, this is the way in which you qualify the names of pi and sin with the name of its originating module:

math.pi

math.sin

It's simple, you put:

* the **name of the module** (e.g., math)
* a **dot** (i.e., .)
* the **name of the entity** (e.g., pi)

Such a form clearly indicates the namespace in which the name exists.

Note: using this qualification is **compulsory** if a module has been imported by the import module instruction. It doesn't matter if any of the names from your code and from the module's namespace are in conflict or not.

This first example won't be very advanced - we just want to print the value of **sin(½π)**.

Look at the code in the editor. This is how we test it.

The code outputs the expected value: 1.0.

Note: removing any of the two qualifications will make the code erroneous. There is no other way to enter math's namespace if you did the following:

import math



Now we're going to show you how the two namespaces (yours and the module's one) can coexist.

Take a look at the example in the editor window.

We've defined our own pi and sin here.

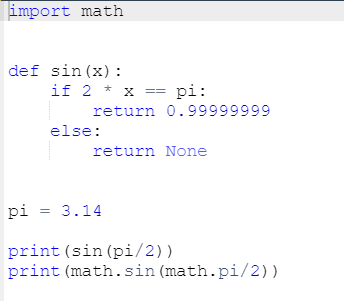
Run the program. The code should produce the following output:

0.99999999

1.0

**output**

As you can see, the entities don't affect each other.



In the second method, the import's syntax precisely points out which module's entity (or entities) are acceptable in the code:

from math import pi

The instruction consists of the following elements:

* the from keyword;
* the **name of the module** to be (selectively) imported;
* the import keyword;
* the **name or list of names of the entity/entities** which are being imported into the namespace.

The instruction has this effect:

* the listed entities (and only those ones) are **imported from the indicated module**;
* the names of the imported entities are **accessible without qualification**.

Note: no other entities are imported. Moreover, you cannot import additional entities using a qualification - a line like this one:

print(math.e)

will cause an error (e is Euler's number: 2.71828...)

Let's rewrite the previous script to incorporate the new technique.

Here it is:

from math import sin, pi

print(sin(pi/2))

The output should be the same as previously, as in fact we've used the same entities as before: 1.0. Copy the code, paste it in the editor, and run the program.

Does the code look simpler? Maybe, but the look is not the only effect of this kind of import. Let's show you that.

Look at the code in the editor. Analyze it carefully:

* line 1: carry out the selective import;
* line 3: make use of the imported entities and get the expected result (1.0)
* lines 5 through 12: redefine the meaning of pi and sin - in effect, they supersede the original (imported) definitions within the code's namespace;
* line 15: get 0.99999999, which confirms our conclusions.

Let's do another test. Look at the code below:

pi = 3.14

def sin(x):

if 2 \* x == pi:

return 0.99999999

else:

return None

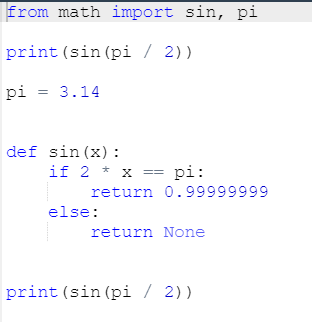
print(sin(pi / 2))

from math import sin, pi

print(sin(pi / 2))

Here, we've reversed the sequence of the code's operations:

* lines 1 through 8: define our own pi and sin;
* line 11: make use of them (0.99999999 appears on the screen)
* line 13: carry out the import - the imported symbols **supersede their previous definitions within the namespace**;
* line 15: get 1.0 as a result.



In the third method, the import's syntax is a more aggressive form of the previously presented one:

from module import \*

As you can see, the name of an entity (or the list of entities' names) is replaced with a single asterisk (\*).

Such an instruction **imports all entities from the indicated module**.

Is it convenient? Yes, it is, as it relieves you of the duty of enumerating all the names you need.

Is it unsafe? Yes, it is - unless you know all the names provided by the module, **you may not be able to avoid name conflicts**. Treat this as a temporary solution, and try not to use it in regular code.

## Importing a module: the as keyword

If you use the import module variant and you don't like a particular module's name (e.g., it's the same as one of your already defined entities, so qualification becomes troublesome) you can give it any name you like - this is called **aliasing**.

Aliasing causes the module to be identified under a different name than the original. This may shorten the qualified names, too.

Creating an alias is done together with importing the module, and demands the following form of the import instruction:

import module as alias

The "module" identifies the original module's name while the "alias" is the name you wish to use instead of the original.

Note: as is a keyword.

If you need to change the word math, you can introduce your own name, just like in the example:

import math as m

print(m.sin(m.pi/2))

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

In turn, when you use the from module import name variant and you need to change the entity's name, you make an alias for the entity. This will cause the name to be replaced by the alias you choose.

This is how it can be done:

from module import name as alias

As previously, the original (unaliased) name becomes inaccessible.

The phrase name as alias can be repeated - use commas to separate the multiplied phrases, like this:

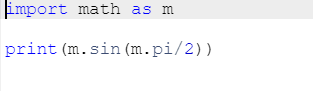
from module import n as a, m as b, o as c

The example may look a bit weird, but it works:

from math import pi as PI, sin as sine

print(sine(PI/2))

Now you're familiar with the basics of using modules. Let us show you some modules and some of their useful entities.



**Key takeaways**

1. If you want to import a module as a whole, you can do it using the import module\_name statement. You are allowed to import more than one module at once using a comma-separated list. For example:

import mod1

import mod2, mod3, mod4

although the latter form is not recommended due to stylistic reasons, and it's better and prettier to express the same intention in more a verbose and explicit form, such as:

import mod2

import mod3

import mod4

2. If a module is imported in the above manner and you want to access any of its entities, you need to prefix the entity's name using **dot notation**. For example:

import my\_module

result = my\_module.my\_function(my\_module.my\_data)

The snippet makes use of two entities coming from the my\_module module: a function named my\_function() and a variable named my\_data. Both names **must be prefixed by** my\_module. None of the imported entity names conflicts with the identical names existing in your code's namespace.

3. You are allowed not only to import a module as a whole, but to import only individual entities from it. In this case, the imported entities **must not be** prefixed when used. For example:

from module import my\_function, my\_data

result = my\_function(my\_data)

The above way - despite its attractiveness - is not recommended because of the danger of causing conflicts with names derived from importing the code's namespace.

4. The most general form of the above statement allows you to import **all entities** offered by a module:

from my\_module import \*

result = my\_function(my\_data)

**Note**: this import's variant is not recommended due to the same reasons as previously (the threat of a naming conflict is even more dangerous here).

5. You can change the name of the imported entity "on the fly" by using the as phrase of the import. For example:

from module import my\_function as fun, my\_data as dat

result = fun(dat)

**Exercise 1**

You want to invoke the function make\_money() contained in the module named mint. Your code begins with the following line:

import mint

What is the proper form of the function's invocation?

Check

mint.make\_money()

**Exercise 2**

You want to invoke the function make\_money() contained in the module named mint. Your code begins with the following line:

from mint import make\_money

What is the proper form of the function's invocation?

Check

make\_money()

**Exercise 3**

You've written a function named make\_money on your own. You need to import a function of the same name from the mint module and don't want to rename any of your previously defined names. Which variant of the import statement may help you with the issue?

Check

# sample solution

from mint import make\_money as make\_more\_money

**Exercise 4**

What form of the make\_money function invocation is valid if your code starts with the following line?

from mint import \*

Check

make\_money()

**Working with standard modules**

Before we start going through some standard Python modules, we want to introduce the dir() function to you. It has nothing to do with the dir command you know from Windows and Unix consoles, as dir() doesn't show the contents of a disk directory/folder, but there is no denying that it does something really similar - it is able to reveal all the names provided through a particular module.

There is one condition: the module has to have been previously imported as a whole (i.e., using the import module instruction - from module is not enough).

The function returns an **alphabetically sorted list** containing all entities' names available in the module identified by a name passed to the function as an argument:

dir(module)

Note: if the module's name has been aliased, you must use the alias, not the original name.

Using the function inside a regular script doesn't make much sense, but it is still possible.

For example, you can run the following code to print the names of all entities within the math module:

import math

for name in dir(math):

print(name, end="\t")

The example code should produce the following output:

\_\_doc\_\_ \_\_loader\_\_ \_\_name\_\_ \_\_package\_\_ \_\_spec\_\_ acos acosh asin asinh atan atan2 atanh ceil copysign cos cosh degrees e erf erfc exp expm1 fabs factorial floor fmod frexp fsum gamma hypot isfinite isinf isnan ldexp lgamma log log10 log1p log2 modf pi pow radians sin sinh sqrt tan tanh trunc

**output**

Have you noticed these strange names beginning with \_\_ at the top of the list? We'll tell you more about them when we talk about the issues related to writing your own modules.

Some of the names might bring back memories from math lessons, and you probably won't have any problems guessing their meanings.

Using the dir() function inside a code may not seem very useful - usually you want to know a particular module's contents before you write and run the code.

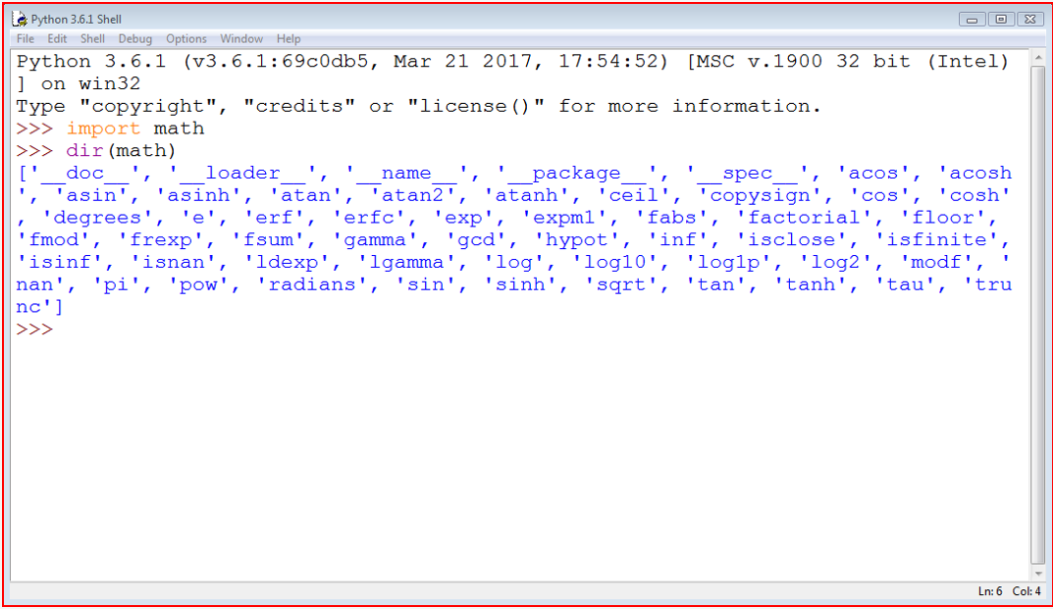
Fortunately, you can execute the function **directly in the Python console** (IDLE), without needing to write and run a separate script.

This is how it can be done:

import math

dir(math)

You should see something similar to this:



**Selected functions from the math module**

Let's start with a quick preview of some of the functions provided by the math module.

We've chosen them arbitrarily, but that doesn't mean that the functions we haven't mentioned here are any less significant. Dive into the modules' depths yourself - we don't have the space or the time to talk about everything in detail here.

The first group of the math's functions are connected with **trigonometry**:

* sin(x) → the sine of x;
* cos(x) → the cosine of x;
* tan(x) → the tangent of x.

All these functions take one argument (an angle measurement expressed in radians) and return the appropriate result (be careful with tan() - not all arguments are accepted).

Of course, there are also their inversed versions:

* asin(x) → the arcsine of x;
* acos(x) → the arccosine of x;
* atan(x) → the arctangent of x.

These functions take one argument (mind the domains) and return a measure of an angle in radians.

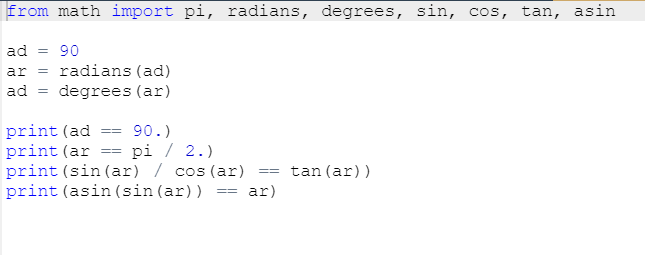
To effectively operate on angle measurements, the math module provides you with the following entities:

* pi → a constant with a value that is an approximation of π;
* radians(x) → a function that converts x from degrees to radians;
* degrees(x) → acting in the other direction (from radians to degrees)

Now look at the code in the editor. The example program isn't very sophisticated, but can you predict its results?

Apart from the circular functions (listed above) the math module also contains a set of their **hyperbolic analogues**:

* sinh(x) → the hyperbolic sine;
* cosh(x) → the hyperbolic cosine;
* tanh(x) → the hyperbolic tangent;
* asinh(x) → the hyperbolic arcsine;
* acosh(x) → the hyperbolic arccosine;
* atanh(x) → the hyperbolic arctangent.



Another group of the math's functions is formed by functions which are connected with **exponentiation**:

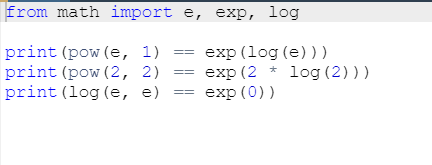
* e → a constant with a value that is an approximation of Euler's number (e)
* exp(x) → finding the value of ex;
* log(x) → the natural logarithm of x
* log(x, b) → the logarithm of x to base b
* log10(x) → the decimal logarithm of x (more precise than log(x, 10))
* log2(x) → the binary logarithm of x (more precise than log(x, 2))

Note: the pow() function:

* pow(x, y) → finding the value of xy (mind the domains)

This is a built-in function, and doesn't have to be imported.

Look at the code in the editor. Can you predict its output?



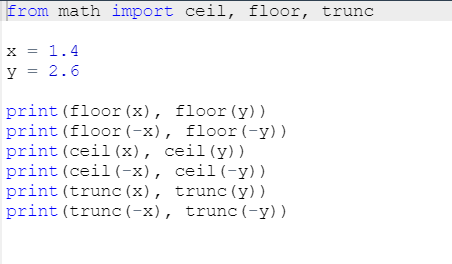
The last group consists of some general-purpose functions like:

* ceil(x) → the ceiling of x (the smallest integer greater than or equal to x)
* floor(x) → the floor of x (the largest integer less than or equal to x)
* trunc(x) → the value of x truncated to an integer (be careful - it's not an equivalent either of ceil or floor)
* factorial(x) → returns x! (x has to be an integral and not a negative)
* hypot(x, y) → returns the length of the hypotenuse of a right-angle triangle with the leg lengths equal to x and y (the same as sqrt(pow(x, 2) + pow(y, 2)) but more precise)

Look at the code in the editor. Analyze the program carefully.

It demonstrates the fundamental differences between ceil(), floor() and trunc().

Run the program and check its output.



# Is there real randomness in computers?

Another module worth mentioning is the one named random.

It delivers some mechanisms allowing you to operate with **pseudorandom numbers**.

Note the prefix **pseudo** - the numbers generated by the modules may look random in the sense that you cannot predict their subsequent values, but don't forget that they all are calculated using very refined algorithms.

The algorithms aren't random - they are deterministic and predictable. Only those physical processes which run completely out of our control (like the intensity of cosmic radiation) may be used as a source of actual random data. Data produced by deterministic computers cannot be random in any way.

A random number generator takes a value called a **seed**, treats it as an input value, calculates a "random" number based on it (the method depends on a chosen algorithm) and produces a **new seed value**.

The length of a cycle in which all seed values are unique may be very long, but it isn't infinite - sooner or later the seed values will start repeating, and the generating values will repeat, too. This is normal. It's a feature, not a mistake, or a bug.

The initial seed value, set during the program start, determines the order in which the generated values will appear.

The random factor of the process may be **augmented by setting the seed with a number taken from the current time** - this may ensure that each program launch will start from a different seed value (ergo, it will use different random numbers).

Fortunately, such an initialization is done by Python during module import.

**Selected functions from the random module**

**The random function**

The most general function named random() (not to be confused with the module's name) **produces a float number**x**coming from the range**(0.0, 1.0) - in other words: (0.0 <= x < 1.0).

The example program below will produce five pseudorandom values - as their values are determined by the current (rather unpredictable) seed value, you can't guess them:

from random import random

for i in range(5):

print(random())

Run the program. This is what we've got:

0.9535768927411208

0.5312710096244534

0.8737691983477731

0.5896799172452125

0.02116716297022092

**sample output**

**The seed function**

The seed() function is able to directly **set the generator's seed**. We'll show you two of its variants:

* seed() - sets the seed with the current time;
* seed(int\_value) - sets the seed with the integer value int\_value.

We've modified the previous program - in effect, we've removed any trace of randomness from the code:

from random import random, seed

seed(0)

for i in range(5):

print(random())

Due to the fact that the seed is always set with the same value, the sequence of generated values always looks the same.

Run the program. This is what we've got:

0.844421851525

0.75795440294

0.420571580831

0.258916750293

0.511274721369

**sample output**

And you?

Note: your values may be slightly different than ours if your system uses more precise or less precise floating-point arithmetic, but the difference will be seen quite far from the decimal point.

**The randrange and randint functions**

If you want integer random values, one of the following functions would fit better:

* randrange(end)
* randrange(beg, end)
* randrange(beg, end, step)
* randint(left, right)

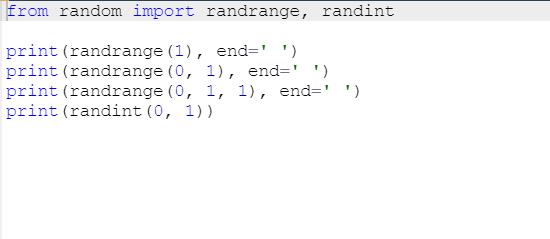
The first three invocations will generate an integer taken (pseudorandomly) from the range (respectively):

* range(end)
* range(beg, end)
* range(beg, end, step)

Note the implicit **right-sided exclusion**!

The last function is an equivalent of randrange(left, right+1) - it generates the integer value i, which falls in the range [left, right] (no exclusion on the right side).

Look at the code in the editor. This sample program will consequently output a line consisting of three zeros and either a zero or one at the fourth place.



The previous functions have one important disadvantage - they may produce repeating values even if the number of subsequent invocations is not greater than the width of the specified range.

Look at the code below - the program very likely outputs a set of numbers in which some elements are not unique:

from random import randint

for i in range(10):

print(randint(1, 10), end=',')

This is what we got in one of the launches:

9,4,5,4,5,8,9,4,8,4,

**sample output**

**The choice and sample functions**

As you can see, this is not a good tool for generating numbers in a lottery. Fortunately, there is a better solution than writing your own code to check the uniqueness of the "drawn" numbers.

It's a function named in a very suggestive way - choice:

* choice(sequence)
* sample(sequence, elements\_to\_choose)

The first variant chooses a "random" element from the input sequence and returns it.

The second one builds a list (a sample) consisting of the elements\_to\_choose element "drawn" from the input sequence.

In other words, the function chooses some of the input elements, returning a list with the choice. The elements in the sample are placed in random order. Note: the elements\_to\_choose must not be greater than the length of the input sequence.

Look at the code below:

from random import choice, sample

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

print(choice(my\_list))

print(sample(my\_list, 5))

print(sample(my\_list, 10))

Again, the output of the program is not predictable. Our results looked like this:

4

[3, 1, 8, 9, 10]

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

# How to know where you are?

Sometimes, it may be necessary to find out information unrelated to Python. For example, you may need to know the location of your program within the greater environment of the computer.

Imagine your program's environment as a pyramid consisting of a number of layers or platforms.



The layers are:

* your (running) code is located at the top of it;
* Python (more precisely - its runtime environment) lies directly below it;
* the next layer of the pyramid is filled with the OS (operating system) - Python's environment provides some of its functionalities using the operating system's services; Python, although very powerful, isn't omnipotent - it's forced to use many helpers if it's going to process files or communicate with physical devices;
* the bottom-most layer is hardware - the processor (or processors), network interfaces, human interface devices (mice, keyboards, etc.) and all other machinery needed to make the computer run; the OS knows how to drive it, and uses lots of tricks to conduct all parts in a consistent rhythm.

This means than some of your (or rather your program's) actions have to travel a long way to be successfully performed - imagine that:

* **your code** wants to create a file, so it invokes one of Python's functions;
* **Python** accepts the order, rearranges it to meet local OS requirements (it's like putting the stamp "approved" on your request) and sends it down (this may remind you of a chain of command)
* the **OS** checks if the request is reasonable and valid (e.g., whether the file name conforms to some syntax rules) and tries to create the file; such an operation, seemingly very simple, isn't atomic - it consists of many minor steps taken by...
* the **hardware**, which is responsible for activating storage devices (hard disk, solid state devices, etc.) to satisfy the OS's needs.

Usually, you're not aware of all that fuss - you want the file to be created and that's that.

But sometimes you want to know more - for example, the name of the OS which hosts Python, and some characteristics describing the hardware that hosts the OS.

There is a module providing some means to allow you to know where you are and what components work for you. The module is named **platform**. We'll show you some of the functions it provides to you.

**Selected functions from the platform module**

**The platform function**

The platform module lets you access the underlying platform's data, i.e., hardware, operating system, and interpreter version information.

There is a function that can show you all the underlying layers in one glance, named platform, too. It just returns a string describing the environment; thus, its output is rather addressed to humans than to automated processing (you'll see it soon).

This is how you can invoke it:

platform(aliased = False, terse = False)

And now:

* aliased → when set to True (or any non-zero value) it may cause the function to present the alternative underlying layer names instead of the common ones;
* terse → when set to True (or any non-zero value) it may convince the function to present a briefer form of the result (if possible)

We ran our sample program using three different platforms - this is what we got:

**Intel x86 + Windows ® Vista (32 bit)**:

Windows-Vista-6.0.6002-SP2

Windows-Vista-6.0.6002-SP2

Windows-Vista

**output**

**Intel x86 + Gentoo Linux (64 bit)**:

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-gentoo-2.3

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-gentoo-2.3

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-glibc2.3.4

**output**

**Raspberry PI2 + Raspbian Linux (32 bit)**:

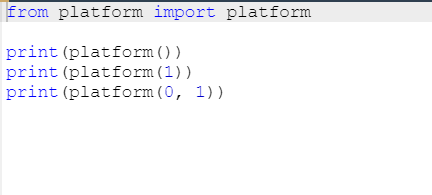
Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0

Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0

Linux-4.4.0-1-rpi2-armv7l-with-glibc2.9

**output**

You can also run the sample program in IDLE on your local machine to check what output you will have.



**The machine function**

Sometimes, you may just want to know the generic name of the processor which runs your OS together with Python and your code - a function named machine() will tell you that. As previously, the function returns a string.

Again, we ran the sample program on three different platforms:

**Intel x86 + Windows ® Vista (32 bit)**:

x86

**output**

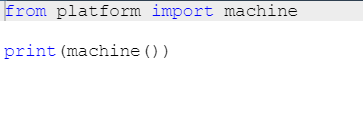
**Intel x86 + Gentoo Linux (64 bit)**:

x86\_64

**output**

**Raspberry PI2 + Raspbian Linux (32 bit)**:

armv7l



**The processor function**

The processor() function returns a string filled with the real processor name (if possible).

Once again, we ran the sample program on three different platforms:

**Intel x86 + Windows ® Vista (32 bit)**:

x86

**output**

**Intel x86 + Gentoo Linux (64 bit)**:

Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

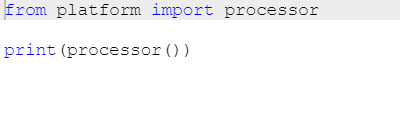
**output**

**Raspberry PI2 + Raspbian Linux (32 bit)**:

armv7l

**output**

Test this on your local machine.



**The system function**

A function named system() returns the generic OS name as a string.

Our example platforms presented themselves like this:

**Intel x86 + Windows ® Vista (32 bit)**:

Windows

**output**

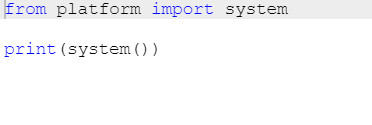
**Intel x86 + Gentoo Linux (64 bit)**:

Linux

**output**

**Raspberry PI2 + Raspbian Linux (32 bit)**:

Linux



**The version function**

The OS version is provided as a string by the version() function.

Run the code and check its output. This is what we got:

**Intel x86 + Windows ® Vista (32 bit)**:

6.0.6002

**output**

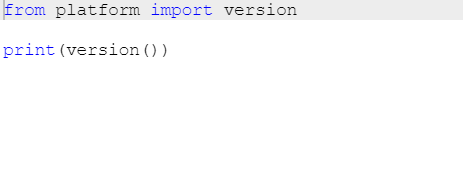
**Intel x86 + Gentoo Linux (64 bit)**:

#1 SMP PREEMPT Fri Jul 21 22:44:37 CEST 2017

**output**

**Raspberry PI2 + Raspbian Linux (32 bit)**:

#1 SMP Debian 4.4.6-1+rpi14 (2016-05-05)



**The python\_implementation and the python\_version\_tuple functions**

If you need to know what version of Python is running your code, you can check it using a number of dedicated functions - here are two of them:

* python\_implementation() → returns a string denoting the Python implementation (expect CPython here, unless you decide to use any non-canonical Python branch)

* python\_version\_tuple() → returns a three-element tuple filled with:
  + the **major** part of Python's version;
  + the **minor** part;
  + the **patch** level number.

Our example program produced the following output:

CPython

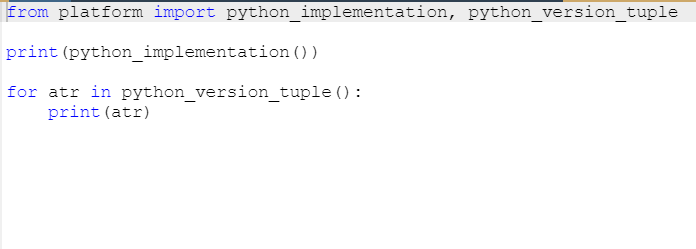
3

7

7

**sample output**

It's very likely that your version of Python will be different.



# Python Module Index

We have only covered the basics of Python modules here. Python's modules make up their own universe, in which Python itself is only a galaxy, and we would venture to say that exploring the depths of these modules can take significantly more time than getting acquainted with "pure" Python.

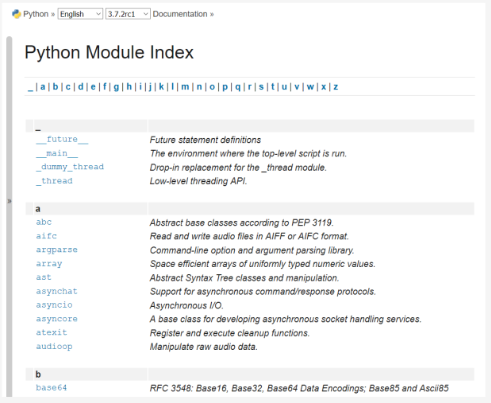
Moreover, the Python community all over the world creates and maintains hundreds of additional modules used in very niche applications like genetics, psychology, or even astrology.

These modules aren't (and won't be) distributed along with Python, or through official channels, which makes the Python universe broader - almost infinite.

You can read about all standard Python modules here: <https://docs.python.org/3/py-modindex.html>.

Don't worry - you won't need all these modules. Many of them are very specific.

All you need to do is find the modules you want, and teach yourself how to use them. It's easy.



In the next section we'll take a look at something else. We're going to show you how to write your own module.

**Key takeaways**

1. A function named dir() can show you a list of the entities contained inside an imported module. For example:

import os

dir(os)

prints out the list of all the os module's facilities you can use in your code.

2. The math module couples more than 50 symbols (functions and constants) that perform mathematical operations (like sine(), pow(), factorial()) or providing important values (like **π** and the Euler symbol **e**).

3. The random module groups more than 60 entities designed to help you use pseudo-random numbers. Don't forget the prefix "random", as there is no such thing as a real random number when it comes to generating them using the computer's algorithms.

4. The platform module contains about 70 functions which let you dive into the underlaying layers of the OS and hardware. Using them allows you to get to know more about the environment in which your code is executed.

5. **Python Module Index** (<https://docs.python.org/3/py-modindex.html> is a community-driven directory of modules available in the Python universe. If you want to find a module fitting your needs, start your search there.

**Exercise 1**

What is the expected value of the result variable after the following code is executed?

import math

result = math.e == math.exp(1)

Check

True

**Exercise 2**

(Complete the sentence) Setting the generator's seed with the same value each time your program is run guarantees that...

Check

... the pseudo-random values emitted from the random module will be exactly the same.

**Exercise 3**

Which of the platform module's functions will you use to determine the name of the CPU running inside your computer?

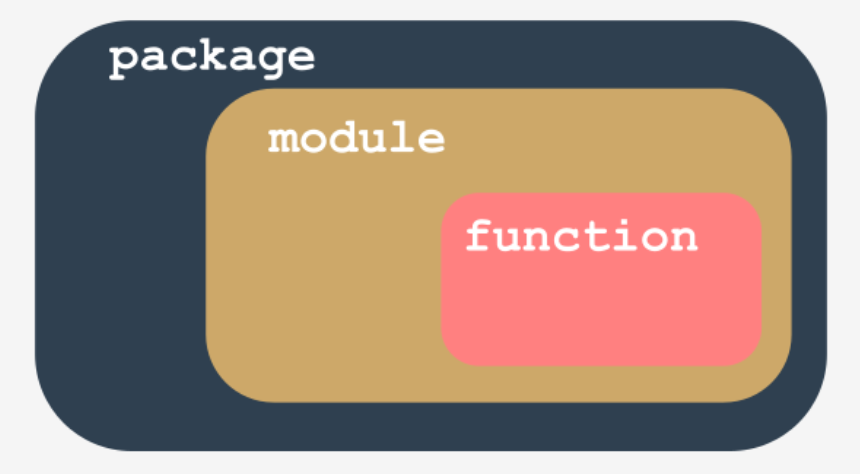
Check

The processor() function

# What is a package?

Writing your own modules doesn't differ much from writing ordinary scripts.

There are some specific aspects you must be aware of, but it definitely isn't rocket science. You'll see this soon enough.

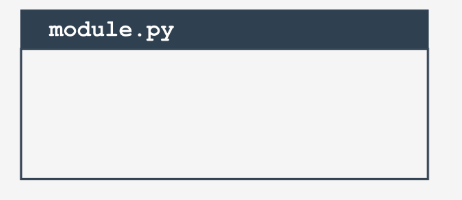


Let's summarize some important issues:

* a **module is a kind of container filled with functions** - you can pack as many functions as you want into one module and distribute it across the world;
* of course, it's generally a good idea not to mix functions with different application areas within one module (just like in a library - nobody expects scientific works to be put among comic books), so group your functions carefully and name the module containing them in a clear and intuitive way (e.g., don't give the name arcade\_games to a module containing functions intended to partition and format hard disks)
* making many modules may cause a little mess - sooner or later you'll want to **group your modules** exactly in the same way as you've previously grouped functions - is there a more general container than a module?
* yes, there is - it's a **package**; in the world of modules, a package plays a similar role to a folder/directory in the world of files.

# Your first module: step 1

In this section you're going to be working locally on your machine. Let's start from scratch. Create an empty file, just like this:

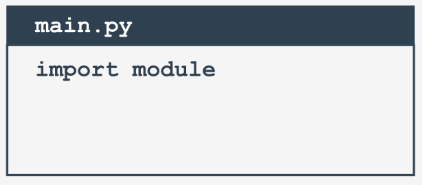


You will need two files to repeat these experiments. The first of them will be the module itself. It's empty now. Don't worry, you're going to fill it with actual code soon.

We've named the file module.py. Not very creative, but simple and clear.

## Your first module: step 2

The second file contains the code using the new module. Its name is main.py. Its content is very brief so far:



import module

main.py

Note: **both files have to be located in the same folder**. We strongly encourage you to create an empty, new folder for both files. Some things will be easier then.

Launch IDLE (or any other IDE you prefer) and run the main.py file. What do you see?

You should see nothing. This means that Python has successfully imported the contents of the module.py file.

It doesn't matter that the module is empty for now. The very first step has been done, but before you take the next step, we want you to take a look into the folder in which both files exist.

Do you notice something interesting?

A new subfolder has appeared - can you see it? Its name is \_\_pycache\_\_. Take a look inside. What do you see?

There is a file named (more or less) module.cpython-xy.pyc where x and y are digits derived from your version of Python (e.g., they will be 3 and 8 if you use Python 3.8).

The name of the file is the same as your module's name (module here). The part after the first dot says which Python implementation has created the file (CPython here) and its version number. The last part (pyc) comes from the words *Python* and *compiled*.

You can look inside the file - the content is completely unreadable to humans. It has to be like that, as the file is intended for Python's use only.

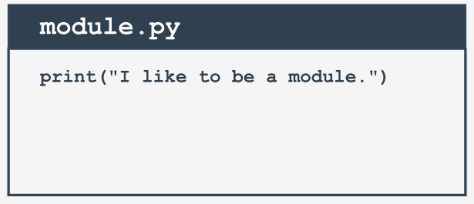
When Python imports a module for the first time, it **translates its contents into a somewhat compiled shape**.

The file doesn't contain machine code - it's internal Python **semi-compiled code**, ready to be executed by Python's interpreter. As such a file doesn't require lots of the checks needed for a pure source file, the execution starts faster, and runs faster, too.

Thanks to that, every subsequent import will go quicker than interpreting the source text from scratch.

Python is able to check if the module's source file has been modified (in this case, the pyc file will be rebuilt) or not (when the pyc file may be run at once). As this process is fully automatic and transparent, you don't have to keep it in mind.

Now we've put a little something into the module file:



print("I like to be a module.")

module.py

Can you notice any differences between a module and an ordinary script? There are none so far.

It's possible to run this file like any other script. Try it for yourself.

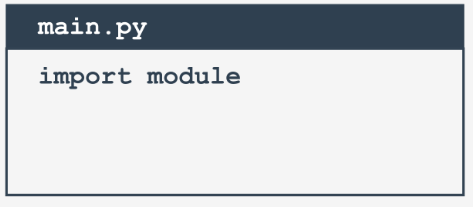
What happens? You should see the following line inside your console:

I like to be a module.

**output**

## Your first module: step 4

Let's go back to the main.py file:



import module

main.py

Run it. What do you see? Hopefully, you see something like this:

I like to be a module.

**output**

What does it actually mean?

When a module is imported, its content is **implicitly executed by Python**. It gives the module the chance to initialize some of its internal aspects (e.g., it may assign some variables with useful values).

Note: the **initialization takes place only once**, when the first import occurs, so the assignments done by the module aren't repeated unnecessarily.

Imagine the following context:

* there is a module named mod1;
* there is a module named mod2 which contains the import mod1 instruction;
* there is a main file containing the import mod1 and import mod2 instructions.

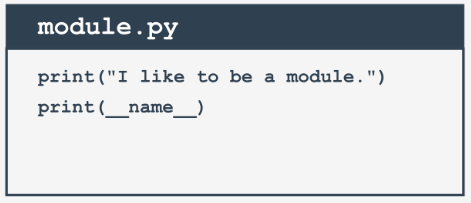
At first glance, you may think that mod1 will be imported twice - fortunately, **only the first import occurs**. Python remembers the imported modules and silently omits all subsequent imports.

## Your first module: step 5

Python can do much more. It also creates a variable called \_\_name\_\_.

Moreover, each source file uses its own, separate version of the variable - it isn't shared between modules.

We'll show you how to use it. Modify the module a bit:



print("I like to be a module.")

print(\_\_name\_\_)

module.py

Now run the module.py file. You should see the following lines:

I like to be a module

\_\_main\_\_

**output**

Now run the main.py file. And? Do you see the same as us?

I like to be a module

module

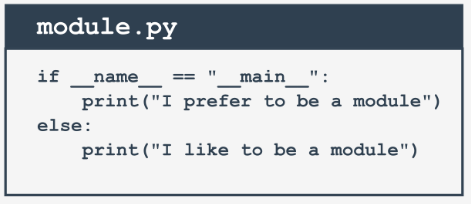
**output**

We can say that:

* when you run a file directly, its \_\_name\_\_ variable is set to \_\_main\_\_;
* when a file is imported as a module, its \_\_name\_\_ variable is set to the file's name (excluding .py)

## Your first module: step 6

This is how you can make use of the \_\_main\_\_ variable in order to detect the context in which your code has been activated:



if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

else:

print("I like to be a module.")

module.py

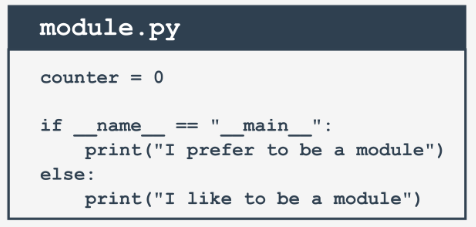
There's a cleverer way to utilize the variable, however. If you write a module filled with a number of complex functions, you can use it to place a series of tests to check if the functions work properly.

Each time you modify any of these functions, you can simply run the module to make sure that your amendments didn't spoil the code. These tests will be omitted when the code is imported as a module.

# Your first module: step 7

This module will contain two simple functions, and if you want to know how many times the functions have been invoked, you need a counter initialized to zero when the module is being imported.

You can do it this way:



counter = 0

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

else:

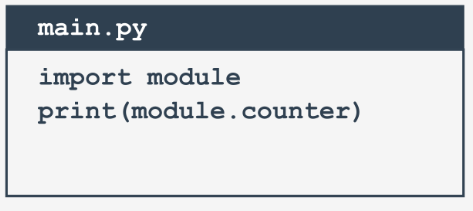
print("I like to be a module.")

module.py

## Your first module: step 8

Introducing such a variable is absolutely correct, but may cause important **side effects** that you must be aware of.

Take a look at the modified main.py file:



import module

print(module.counter)

main.py

As you can see, the main file tries to access the module's counter variable. Is this legal? Yes, it is. Is it usable? It may be very usable. Is it safe?

That depends - if you trust your module's users, there's no problem; however, you may not want the rest of the world to see your **personal/private variable**.

Unlike many other programming languages, Python has no means of allowing you to hide such variables from the eyes of the module's users.

You can only inform your users that this is your variable, that they may read it, but that they should not modify it under any circumstances.

This is done by preceding the variable's name with \_ (one underscore) or \_\_ (two underscores), but remember, it's only a **convention**. Your module's users may obey it or they may not.

Of course, we'll follow the convention. Now let's put two functions into the module - they'll evaluate the sum and product of the numbers collected in a list.

In addition, let's add some ornaments there and remove any superfluous remnants.

## Your first module: step 9

Okay. Let's write some brand new code in our module.py file. The updated module is ready here:

#!/usr/bin/env python3

""" module.py - an example of a Python module """

\_\_counter = 0

def suml(the\_list):

global \_\_counter

\_\_counter += 1

the\_sum = 0

for element in the\_list:

the\_sum += element

return the\_sum

def prodl(the\_list):

global \_\_counter

\_\_counter += 1

prod = 1

for element in the\_list:

prod \*= element

return prod

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module, but I can do some tests for you.")

my\_list = [i+1 for i in range(5)]

print(suml(my\_list) == 15)

print(prodl(my\_list) == 120)

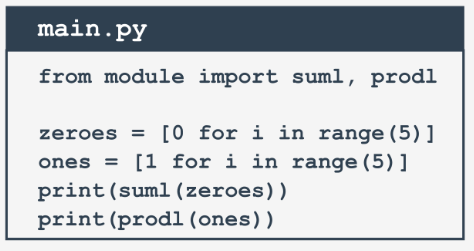
module.py

A few elements need some explanation, we think:

* the line starting with #! has many names - it may be called *shabang*, *shebang*, *hashbang*, *poundbang* or even *hashpling* (don't ask us why). The name itself means nothing here - its role is more important. From Python's point of view, it's just a **comment** as it starts with #. For Unix and Unix-like OSs (including MacOS) such a line **instructs the OS how to execute the contents of the file** (in other words, what program needs to be launched to interpret the text). In some environments (especially those connected with web servers) the absence of that line will cause trouble;
* a string (maybe a multiline) placed before any module instructions (including imports) is called the **doc-string**, and should briefly explain the purpose and contents of the module;
* the functions defined inside the module (suml() and prodl()) are available for import;
* we've used the \_\_name\_\_ variable to detect when the file is run stand-alone, and seized this opportunity to perform some simple tests.

## Your first module: step 10

Now it's possible to use the updated module - this is one way:



from module import suml, prodl

zeroes = [0 for i in range(5)]

ones = [1 for i in range(5)]

print(suml(zeroes))

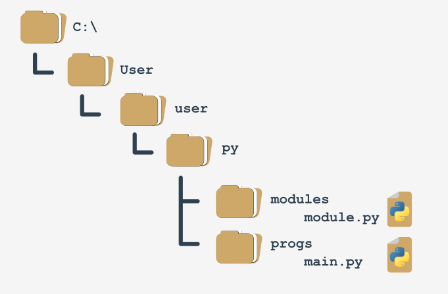
print(prodl(ones))

**Your first module: step 11**

It's time to make our example more complicated - so far we've assumed that the main Python file is located in the same folder/directory as the module to be imported.

Let's give up this assumption and conduct the following thought experiment:

* we are using Windows ® OS (this assumption is important, as the file name's shape depends on it)
* the main Python script lies in C:\Users\user\py\progs and is named main.py
* the module to import is located in C:\Users\user\py\modules



How to deal with it?

To answer this question, we have to talk about **how Python searches for modules**. There's a special variable (actually a list) storing all locations (folders/directories) that are searched in order to find a module which has been requested by the import instruction.

Python browses these folders in the order in which they are listed in the list - if the module cannot be found in any of these directories, the import fails.

Otherwise, the first folder containing a module with the desired name will be taken into consideration (if any of the remaining folders contains a module of that name, it will be ignored).

The variable is named path, and it's accessible through the module named sys. This is how you can check its regular value:

import sys

for p in sys.path:

print(p)

We've launched the code inside the C:\User\user folder, and this is what we've got:

C:\Users\user

C:\Users\user\AppData\Local\Programs\Python\Python36-32\python36.zip

C:\Users\user\AppData\Local\Programs\Python\Python36-32\DLLs

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib

C:\Users\user\AppData\Local\Programs\Python\Python36-32

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib\site-packages

**sample output**

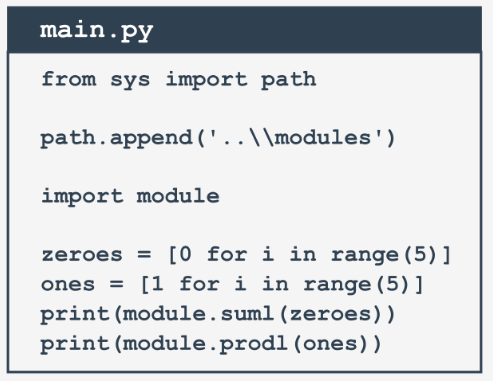
Note: the folder in which the execution starts is listed in the **first path's element**.

Note once again: there is a zip file listed as one of the path's elements - it's not an error. Python is able to treat zip files as ordinary folders - this can save lots of storage.

Can you figure out how we can solve our problem now? We can add a folder containing the module to the path variable (it's fully modifiable).

## Your first module: step 12

One of several possible solutions looks like this:



from sys import path

path.append('..\\modules')

import module

zeroes = [0 for i in range(5)]

ones = [1 for i in range(5)]

print(module.suml(zeroes))

print(module.prodl(ones))

main.py

Note:

* we've doubled the \ inside folder name - do you know why?  
    
  Check

Because a backslash is used to escape other characters - if you want to get just a backslash, you have to escape it.

* we've used the relative name of the folder - this will work if you start the main.py file directly from its home folder, and won't work if the current directory doesn't fit the relative path; you can always use an absolute path, like this:  
    
  path.append('C:\\Users\\user\\py\\modules')

* we've used the append() method - in effect, the new path will occupy the last element in the path list; if you don't like the idea, you can use insert() instead.

# Your first package: step 1

Imagine that in the not-so-distant future you and your associates write a large number of Python functions.

Your team decides to group the functions in separate modules, and this is the final result of the ordering:



#! /usr/bin/env python3

""" module: alpha """

def funA():

return "Alpha"

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module.")

alpha.py

Note: we've presented the whole content for the alpha.py module only - assume that all the modules look similar (they contain one function named funX, where *X* is the first letter of the module's name).

## Your first package: step 2

Suddenly, somebody notices that these modules form their own hierarchy, so putting them all in a flat structure won't be a good idea.

After some discussion, the team comes to the conclusion that the modules have to be grouped. All participants agree that the following tree structure perfectly reflects the mutual relationships between the modules:



Let's review this from the bottom up:

* the ugly group contains two modules: psi and omega;
* the best group contains two modules: sigma and tau;
* the good group contains two modules (alpha and beta) and one subgroup (best)
* the extra group contains two subgroups (good and bad) and one module (iota)

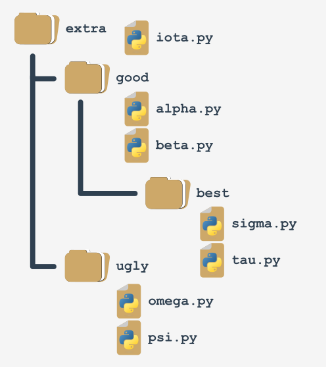
Does it look bad? Not at all - analyze the structure carefully. It resembles something, doesn't it?

It looks like a **directory structure**.

Let's build a tree reflecting projected dependencies between the modules.

# Your first package: step 3

This is how the tree currently looks:



Such a structure is almost a package (in the Python sense). It lacks the fine detail to be both functional and operative. We'll complete it in a moment.

If you assume that extra is the name of a **newly created package** (think of it as the **package's root**), it will impose a naming rule which allows you to clearly name every entity from the tree.

For example:

* the location of a function named funT() from the tau package may be described as:
* extra.good.best.tau.funT()
* a function marked as:
* extra.ugly.psi.funP()

comes from the psi module being stored in the ugly subpackage of the extra package.

## Your first package: step 4

There are two questions to answer:

* **how** do you transform such a tree (actually, a subtree) into a real Python **package** (in other words, how do you convince Python that such a tree is not just a bunch of junk files, but a set of modules)?
* **where** do you put the subtree to make it accessible to Python?

The first question has a surprising answer: **packages, like modules, may require initialization**.

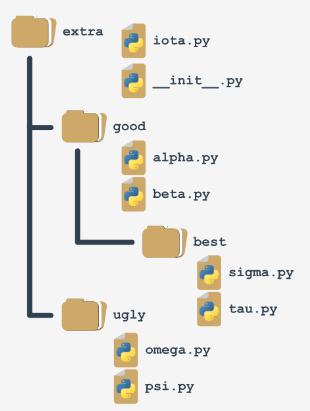
The initialization of a module is done by an *unbound* code (not a part of any function) located inside the module's file. As a package is not a file, this technique is useless for initializing packages.

You need to use a different trick instead - Python expects that there is a file with a very unique name inside the package's folder: \_\_init\_\_.py.

The content of the file is executed when any of the package's modules is **imported**. If you don't want any special initializations, you can leave the file empty, but you mustn't omit it.

# Your first package: step 5

Remember: **the presence of the \_\_init.py\_\_ file finally makes up the package**.



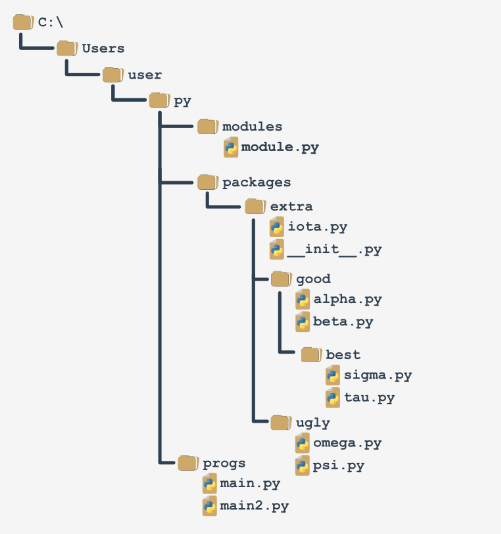
Note: it's not only the *root* folder that can contain \_\_init.py\_\_ file - you can put it inside any of its subfolders (subpackages) too. It may be useful if some of the subpackages require individual treatment and special kinds of initialization.

Now it's time to answer the second question - the answer is simple: **anywhere**. You only have to ensure that Python is aware of the package's location. You already know how to do that.

You're ready to make use of your first package.

## Your first package: step 6

Let's assume that the working environment looks as follows:



We've prepared a zip file containing all the files from the packages branch. You can download it and use it for your own experiments, but **remember to unpack it in the folder presented in the scheme**, otherwise, it won't be accessible to the code from the main file.

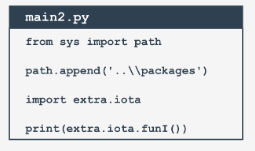
**DOWNLOAD**   [Modules and Packages ZIP file](https://edube.org/uploads/media/default/0001/01/39a03067f85a9b39c0c94ca50b5d445b7df18a04.zip)

You'll be continuing your experiments using the main2.py file.

# Your first package: step 7

We are going to access the funI() function from the iota module from the top of the extra package. It forces us to use qualified package names (associate this with naming folders and subfolders - the conventions are very similar).

This is how to do it:



from sys import path

path.append('..\\packages')

import extra.iota

print(extra.iota.funI())

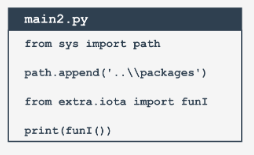
main2.py

Note:

* we've modified the path variable to make it accessible to Python;
* the import doesn't point directly to the module, but specifies the fully qualified path from the top of the package;

replacing import extra.iota with import iota will cause an error.

The following variant is valid too:



from sys import path

path.append('..\\packages')

from extra.iota import funI

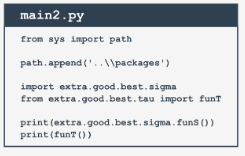
print(funI())

main2.py

Note the qualified name of the iota module.

# Your first package: step 8

Now let's reach all the way to the bottom of the tree - this is how to get access to the sigma and tau modules:



from sys import path

path.append('..\\packages')

import extra.good.best.sigma

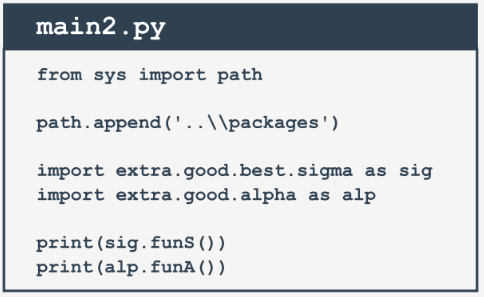
from extra.good.best.tau import funT

print(extra.good.best.sigma.funS())

print(funT())

main2.py

You can make your life easier by using aliasing:



from sys import path

path.append('..\\packages')

import extra.good.best.sigma as sig

import extra.good.alpha as alp

print(sig.funS())

print(alp.funA())

main2.py

## Your first package: step 9

Let's assume that we've zipped the whole subdirectory, starting from the extra folder (including it), and let's get a file named extrapack.zip. Next, we put the file inside the packages folder.

Now we are able to use the zip file in a role of packages:

from sys import path

path.append('..\\packages\\extrapack.zip')

import extra.good.best.sigma as sig

import extra.good.alpha as alp

from extra.iota import funI

from extra.good.beta import funB

print(sig.funS())

print(alp.funA())

print(funI())

print(funB())

main2.py

If you want to conduct your own experiments with the package we've created, you can download it below. We encourage you to do so.

**DOWNLOAD**   [Extrapack ZIP file](https://edube.org/uploads/media/default/0001/01/d9df38daa0410952c4cbf85b47892954c45b9215.zip" \t "_blank)

Now you can create modules and combine them into packages. It's time to start a completely different discussion - about errors, failures and crashes.

**Key takeaways**

1. While a **module** is designed to couple together some related entities (functions, variables, constants, etc.), a **package** is a container which enables the coupling of several related modules under one common name. Such a container can be distributed as-is (as a batch of files deployed in a directory sub-tree) or it can be packed inside a zip file.

2. During the very first import of the actual module, Python translates its source code into the **semi-compiled** format stored inside the **pyc** files, and deploys these files into the \_\_pycache\_\_ directory located in the module's home directory.

3. If you want to instruct your module's user that a particular entity should be treated as **private** (i.e. not to be explicitly used outside the module) you can mark its name with either the \_ or \_\_ prefix. Don't forget that this is only a recommendation, not an order.

4. The names *shabang*, *shebang*, *hasbang*, *poundbang*, and *hashpling* describe the digraph written as #!, used to instruct Unix-like OSs how the Python source file should be launched. This convention has no effect under MS Windows.

5. If you want convince Python that it should take into account a non-standard package's directory, its name needs to be inserted/appended into/to the import directory list stored in the path variable contained in the sys module.

6. A Python file named \_\_init\_\_.py is implicitly run when a package containing it is subject to import, and is used to initialize a package and/or its sub-packages (if any). The file may be empty, but must not be absent.

**Exercise 1**

You want to prevent your module's user from running your code as an ordinary script. How will you achieve such an effect?

Check

import sys

if \_\_name\_\_ == "\_\_main\_\_":

print "Don't do that!"

sys.exit()

**Exercise 2**

Some additional and necessary packages are stored inside the D:\Python\Project\Modules directory. Write a code ensuring that the directory is traversed by Python in order to find all requested modules.

Check

import sys

# note the double backslashes!

sys.path.append("D:\\Python\\Project\\Modules")

**Exercise 3**

The directory mentioned in the previous exercise contains a sub-tree of the following structure:

abc

|\_\_ def

|\_\_ mymodule.py

Assuming that D:\Python\Project\Modules has been successfully appended to the sys.path list, write an import directive letting you use all the mymodule entities.

Check

import abc.def.mymodule

**Python packaging ecosystem and how to use it**

Python is a very powerful instrument – we hope you've experienced this yourself already. Many people from around the world feel this way, and they use Python on a regular basis to develop what they can do in many completely different fields of activity. This means that Python has become **an interdisciplinary tool** employed in countless applications. We can’t go through all the spheres in which Python brilliantly shows off its abilities, so let us just tell you about the most impressive ones.

First of all, Python has turned into **a leader of research on artificial intelligence**. Data mining, one of the most promising modern scientific disciplines, utilizes Python as well. Mathematicians, psychologists, geneticists, meteorologists, linguists – all these people already use Python, or if they don’t already, we’re sure that they will very soon. There is no escaping this trend.

Of course, it doesn’t make any sense to get all Python users to write their code from scratch, keeping them perfectly isolated from the outside world and from other programmers' achievements. This would be both unnatural and counterproductive.

The most preferable and efficient thing is to enable all Python community members to freely exchange their codes and experiences. In this model, nobody is forced to start work from scratch, as there’s a high probability that someone else has been working on the same (or a very similar) problem.

As you know, Python was created as open-source software, and this also works as an invitation for all coders to maintain the whole Python ecosystem as an open, friendly, and free environment. To make the model work and evolve, some additional tools should be provided, tools that help the creators to publish, maintain, and take care of their code.

These same tools should help users to make use of the code, both the already existing code, and also the new code appearing every day. Thanks to that, writing new code for new challenges is not like building a new house, starting at the foundations.

Moreover, the programmer is free to modify someone else's code in order to adapt it to their own needs, and in effect build a completely new product that can be used by another developer. The process seems to have no end. Fortunately.

To make this world go round, two basic entities have to be established and kept in motion: **a centralized repository** of all available software packages; and a tool allowing users to **access the repository**. Both these entities already exist and can be used at any time.

**Python packaging ecosystem and how to use it: continued**

The repository (or *repo* for short) we mentioned before is named **PyPI** (it's short for Python Package Index) and it's maintained by a workgroup named the Packaging Working Group, a part of the Python Software Foundation, whose main task is to support Python developers in efficient code dissemination.

You can find their website here:  
<https://wiki.python.org/psf/PackagingWG>.

The PyPI website (administered by PWG) is located at the address:  
<https://pypi.org/>.

When we popped in there for a while at the beginning of June 2020, we found that PyPI was home to 237,515 projects, consisting of 2,877,545 files managed by 427,487 users.

These three numbers alone clearly show the potency of the Python community and the importance of developer cooperation.

We must point out that PyPI is not the only existing Python repository. On the contrary, there are lots of them, created for projects and led by many larger and smaller Python communities. It's likely that someday you and your colleagues may want to **create your own repos**.

Anyway, PyPI is the most important Python repo in the world. If we modify the classic saying a little, we can state that “all Python roads lead to PyPl”, and that’s no exaggeration at all.

# The PyPI repo: the Cheese Shop

The PyPI repo is sometimes referred to as the **Cheese Shop**. Really.

Does that sound a little strange to you? Don't worry, it’s all perfectly innocent.

We refer to the repo as a shop, because you go there for the same reasons you go to other shops: to fulfill your needs. If you want some cheese, you go to the cheese shop. If you want a piece of software, you go to the software shop. Fortunately, the analogy ends here – you don't need any money to take some software out of the repo shop.

PyPI is completely free, and you can just pick a code and use it – you’ll encounter neither cashier nor security guard. Of course, it doesn't absolve you from being polite and honest. You have to obey all the licensing terms, so don't forget to read them.

“OK”, you say, “the shop is clear now, but what does cheese have to do with Python?”

The Cheese Shop is one of the most famous Monty Python sketches. It depicts the surrealist adventure of an Englishman trying to buy some cheese. Unfortunately, the shop he visits (immodestly named Ye National Cheese Emporium) has no cheese in stock at all.

Of course, it's meant to be ironic. As you already know, PyPI has lots of software in stock and it's available 24/7. It's fully entitled to identify itself as *Ye International Python Software Emporium*.

# The PyPI repo: the Cheese Shop (continued)

PyPI is a very specific shop, not just because it offers all its products for free. It also requires a special tool to make use of it.

Fortunately, this tool is also free, so if you want to make your own digital cheeseburger by using the goods offered by the PyPI Shop, you’ll need a free tool named *pip*.

No, you haven't misheard. Just *pip*. It's another acronym, of course, but its nature is more complex than the previously mentioned PyPI, as it's an example of a recursive acronym, which means that the acronym refers to itself, which means that explaining it is an infinite process.

Why? Because *pip* means *“pip installs packages”*, and the *pip* inside *“pip installs packages”* means *“pip installs packages”* and ...

Let’s stop there. Thank you for your cooperation.

By the way, there are a few other very famous recursive acronyms. One of them is *Linux*, which can be interpreted as *“Linux is Not Unix”*.

# How to install *pip*

The question that should be put now is: how to get a proper cheese knife? In other words, how to ensure that *pip* is installed and ready to work?

The most precise answer is “it depends”. Really.

Some Python installations come with *pip*, some don't. What’s more, it doesn’t only depend on the OS you use, although this is a very important factor.

Let's start with MS Windows.

***pip* on MS Windows**

The MS Windows Python installer already contains *pip*, and so no other steps need to be taken in order to install it. Unfortunately, if the PATH variable is misconfigured, *pip* may unavailable.

To verify that we haven’t misled you, try to do this:

* open the Windows console (*CMD* or *PowerShell*, whatever you prefer)
* execute the following command:
* pip --version
* in the most optimistic scenario (and we really want that to happen) you’ll see something like this:
* the absence of this message may mean that the PATH variable either incorrectly points to the location of the Python binaries, or doesn't point to it at all; for example, our PATH variable contains the following substring:  
    
  C:\Program Files\Python3\Scripts\;C:\Program Files\Python3\;
* the easiest way to reconfigure the PATH variable is to **reinstall Python**, instructing the installer to set it for you.

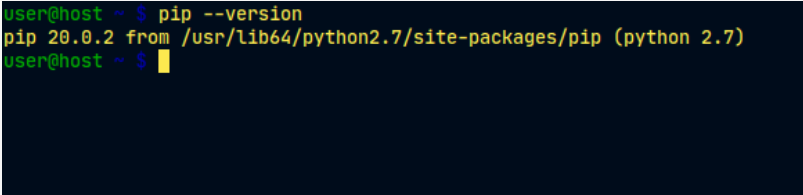
***pip* on Linux**

Different Linux distributions may behave differently when it comes to using *pip*. Some of them (like *Gentoo*), which are closely bound to Python and which use it internally, may have *pip* preinstalled and are instantly ready to work.

Don't forget that some Linuces may utilize more than one Python version concurrently, e.g., one Python 2 and one Python 3 coexisting side by side. Such systems may launch Python 2 as the default version, and it may be necessary to explicitly specify the program name as python3. In this case there may be two different *pip*s identified as *pip* (or *pip2*) and *pip3*. Check it carefully.

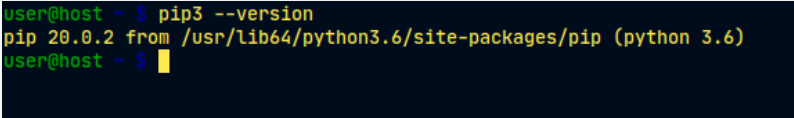
Open the terminal window and issue the following command:

pip --version



An answer similar to the one shown in the previous picture determines that you've launched *pip* from Python 2, so the next try should look as follows:

pip3 --version



As you can see, we’re now sure that we’re using the appropriate version of *pip*.

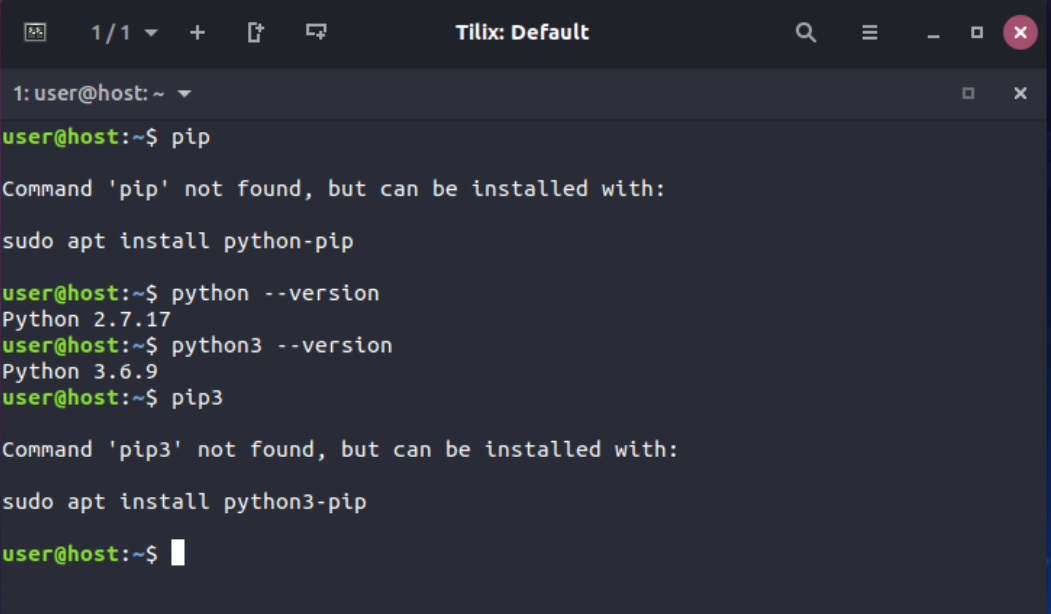
***pip* on Linux: continued**

Unfortunately, some Linux distributions don't have *pip* preinstalled, even if Python itself is installed by default (some versions of Ubuntu may behave this way). In this case, you have two possibilities:

* install *pip* as a system package using a dedicated package manager (e.g., *apt* in Debian-like systems)
* install *pip* using internal Python mechanisms.

The former is definitely better. Although there are some smart scripts that are able to download and install *pip* by ignoring the OS, we discourage you from using them. This method can get you into trouble.

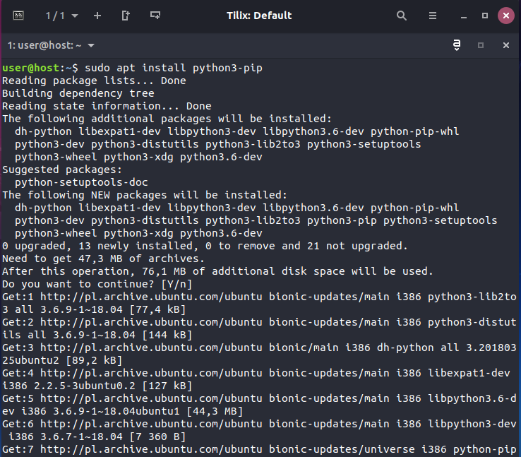
Look – we've tried to launch *pip3* and we’ve failed. Our OS (we used *Ubuntu Budgie* this time) suggested using *apt* in order to install the package named *python3-pip*:



That's good advice, and we're going to follow it, but it has to be stated that we’ll need administrative rights to do it. Don't forget that different Linuces may use different package managers (e.g., it could be *pacman* if you use Arch Linux, or *yum* used by distributions derived from Red Hat).

Anyway, all these methods should get *pip* (or *pip3*) installed and working.

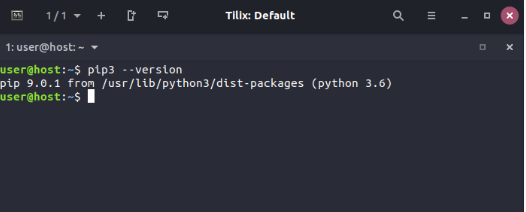
Look what happened when we followed the OS suggestion:



As you can see, the OS decided to install not only *pip* itself, but also a couple of additional components needed by *pip*. This is normal – don't be alarmed.

# *pip* on Linux: continued

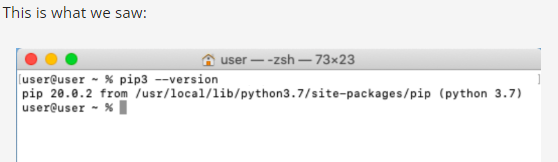
When *apt* finishes its job, we are finally able to utilize *pip3*:



If you're a Mac user and you've installed Python 3 using the *brew* installer, *pip* is already present in your system and ready to work. Check it by issuing the previously mentioned command:

pip3 --version

and wait for the response.



# Dependencies

Now that we’re sure that *pip* is ready at our command, we’re going to limit our focus to MS Windows only, as its behavior is (should be) the same in all OSes, but before we start, we need to explain an important issue and tell you about **dependencies**.

Imagine that you've created a brilliant Python application named *redsuspenders*, able to predict stock exchange rates with 99% accuracy (by the way, if you actually do that, please contact us immediately).

Of course, you've used some existing code to achieve this goal – e.g., your app imports a package named *nyse* containing some crucial functions and classes. Moreover, the *nyse* package imports another package named *wallstreet*, while the *wallstreet* package imports other two essential packages named *bull* and *bear*.

As you’ve probably already guessed, the connections between these packages are crucial, and if somebody decides to use your code (but remember, we've already called dibs on it) they will also have to ensure that all required packages are in place.

To make a long story short, we can say that **dependency is a phenomenon that appears every time you're going to use a piece of software that relies on other software**. Note that dependency may include (and generally does include) more than one level of software development.

Does this mean that a potential *nyse* package user is obliged to trace all dependencies and manually install all the needed packages? That would be horrible, wouldn't it?

Yes, it's definitely horrible, so you shouldn't be surprised that the process of arduously fulfilling all the subsequent requirements has its own name, and it's called *dependency hell*.

How do we deal with that? Is every user doomed to visit hell in order to run the code for the first time?

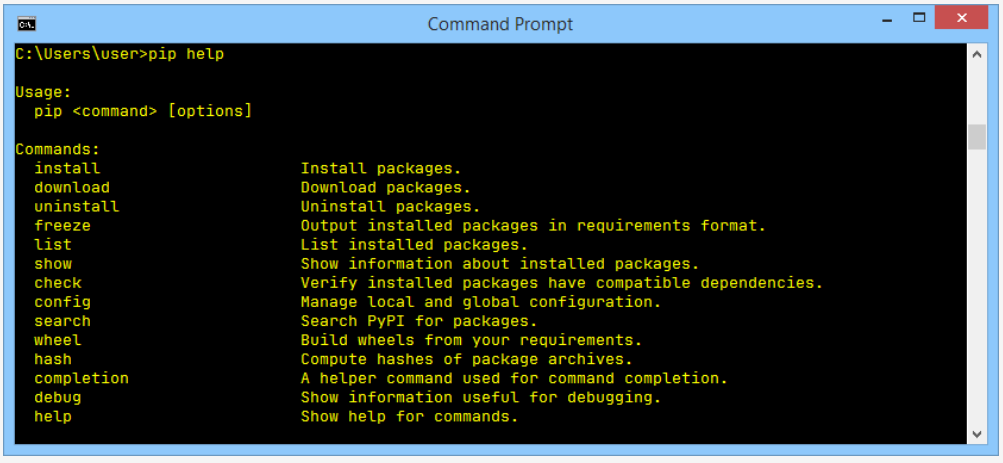
Fortunately not - *pip* can do all of this for you. Really. It can discover, identify, and resolve all dependencies. Moreover, it can do it in the cleverest way, avoiding any unnecessary downloads and reinstalls.

**How to use *pip***

Now we’re ready to ask *pip* what it can do for us. Let's do it – issue the following command:

pip help

and wait for *pip*'s response. This is what it looks like:



Don't forget that you may be obliged to replace *pip* with *pip3* if your environment requires this.

The list produced by *pip* summarizes all the available operations, and the last of them is help, which we've just used already.

If you want to know more about any of the listed operations, you can use the following form of *pip* invocation:

pip help operation

For example, the line:

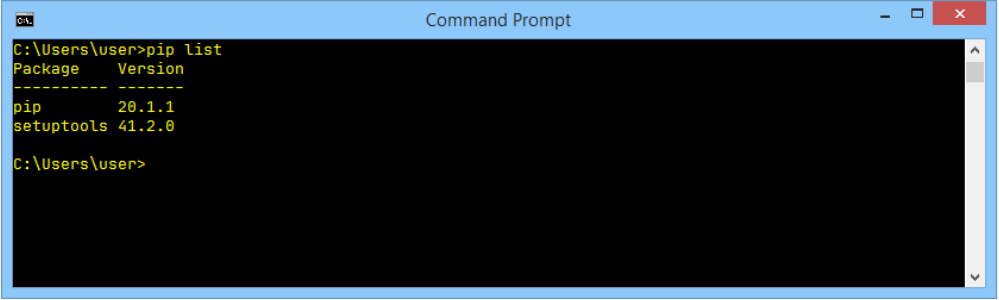
pip help install

will show you detailed information about using and parameterizing the install command.

If you want to know what Python packages have been installed so far, you can use the list operation – just like this:

pip list

The output you’ll see is rather unpredictable. Don't be surprised if your screen ends up being filled with completely different content. Ours look as follows:



As you can see, there are two columns in the list, one showing the name of the installed package, and the other showing the version of the package. We can’t predict the state of your Python installation.

The only thing we know for sure is that your list contains the two lines we see on our list: *pip* and *setuptools*. This happens because the OS is convinced that a user wanting pip will very likely need the *setuptools* soon. It’s not wrong.

**How to use *pip*: continued**

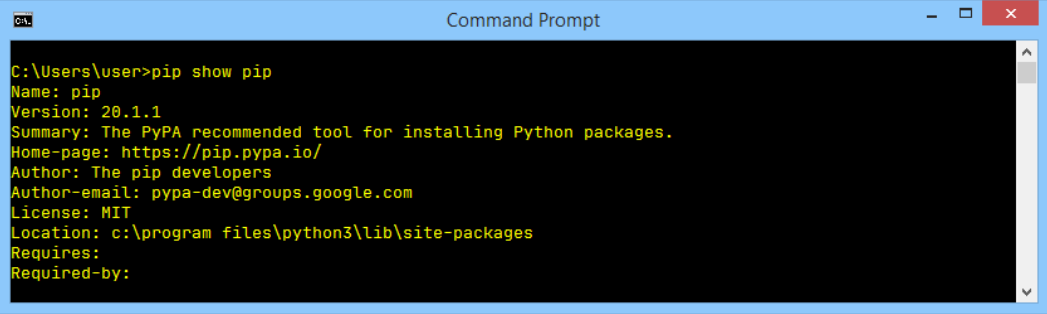
The *pip* list isn't very informative, and it may happen that it won't satisfy your curiosity. Fortunately, there’s a command that can tell you more about any of the installed packages (note the word **installed**). The syntax of the command looks as follows:

pip show package\_name

We’re going to use it in a slightly deceptive way – we want to convince *pip* to confess something about itself. This is how we do it:

pip show pip

It looks a bit odd, doesn't it? Despite this, it works fine, and *pip*'s self-presentation looks consistent and current:



You may ask where this data comes from? Is *pip* really so perceptive? Not at all – the information appearing on the screen is taken from inside the package being shown. In other words, the package's creator is obliged to equip it with all the needed data (or to express it more precisely – metadata).

Look at the two lines at the bottom of the output. They show:

* which packages are needed to successfully utilize the package (Requires:)
* which packages need the package to be successfully utilized (Required-by:)

As you can see, both properties are empty. Feel free to try to use the show command in relation to any other installed package.

The power of *pip* comes from the fact that it’s actually a gateway to the Python software universe. Thanks to that, you can browse and install any of the hundreds of ready-to-use packages gathered in the PyPI repositories. Don't forget that *pip* is not able to store all PyPI content locally (it’s unnecessary and it would be uneconomical).

In effect, *pip* uses the Internet to query PyPI and to download the required data. This means that you have to have a network connection working whenever you’re going to ask *pip* for anything that may involve direct interactions with the PyPI infrastructure.

One of these cases occurs when you want to search through PyPI in order to find a desired package. This kind of search is initiated by the following command:

pip search anystring

The anystring provided by you will be searched in:

* the names of all the packages;
* the summary strings of all the packages.

Be aware of the fact that some searches may generate a real avalanche of data, so try to be as specific as possible. For example, an innocent-looking query like this one:

pip search pip

produces more than 100 lines of results (try it yourself – don't take our word for it). By the way – the search is case insensitive.

If you’re not a fan of console reading, you can use the alternative way of browsing PyPI content offered by a search engine, available at <https://pypi.org/search>.

**How to use *pip*: continued**

Assuming that your search is successful (or you’re determined to install a specific package of an already known name) you can use pip to install the package onto your computer.

Two possible scenarios may be put into action now:

* you want to install a new package for you only – it won't be available for any other user (account) existing on your computer; this procedure is the only one available if you can’t elevate your permissions and act as a system administrator;
* you’ve decided to install a new package system-wide – you have administrative rights and you're not afraid to use them.

To distinguish between these two actions, pip uses a dedicated option named --user (note the double dash). The presence of this option instructs pip to act locally on behalf of your (non-administrative) user.

If you don’t add this, pip assumes that you’re as a system administrator and it’ll do nothing to correct you if you’re not.

In our case, we’re going to install a package named pygame – it's an extended and complex library allowing programmers to develop computer games using Python.

The project has been in development since the year 2000, so it's a mature and reliable piece of code. If you want to know more about the project and about the community which leads it, visit [https://www.pygame.org](https://www.pygame.org/).

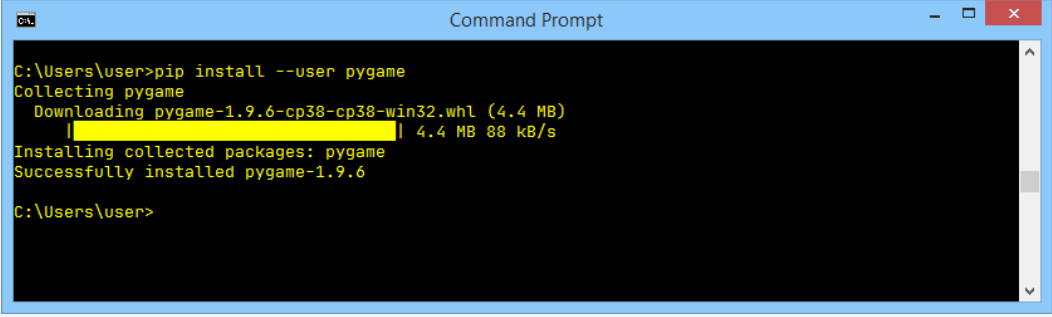
If you’re a system administrator, you can install pygame using the following command:

pip install pygame

If you're not an admin, or you don't want to fatten up your OS by installing pygame system-wide, you can install it for you only:

pip install --user pygame

It's up to you which of the above procedures you want to take place.



Pip has a habit of displaying fancy textual animation indicating the installation progress, so watch the screen carefully – don't miss the show! If the process is successful, you’ll see something like this:

We encourage you to use:

pip show pygame

and

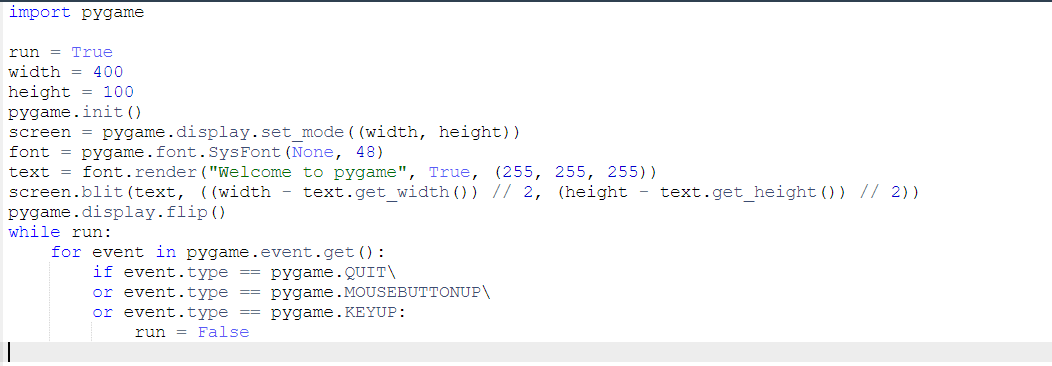
pip list

to get more information about what actually happened.

**How to use pip: a simple test program**

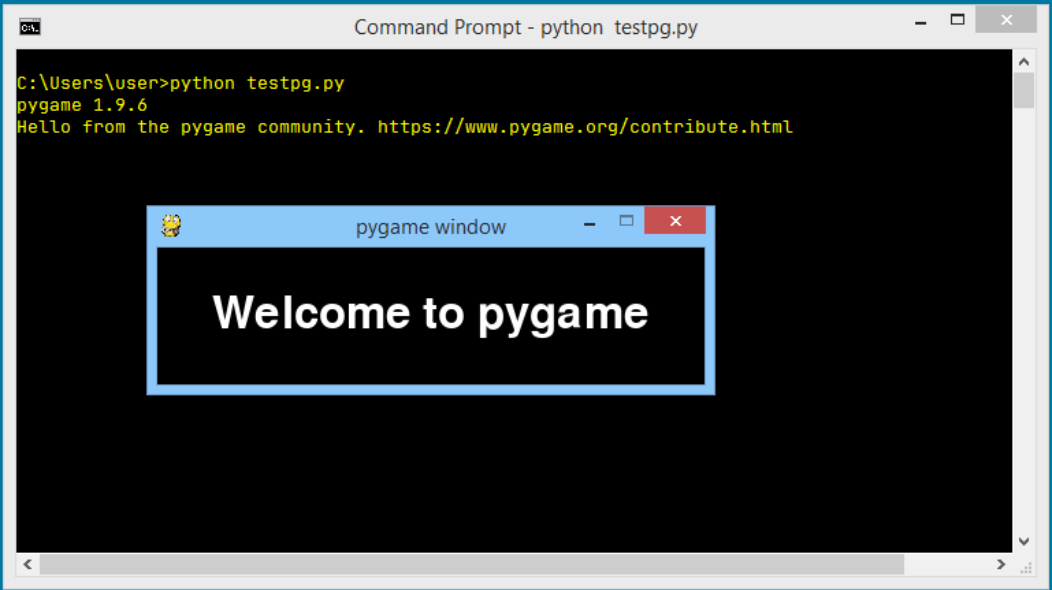
Now that *pygame* is finally accessible, we can try to use it in a very simple test program. Let’s comment on it briefly.

* line 1: import *pygame* and let it serve us;
* line 3: the program will run as long as the run variable is True;
* lines 4 and 5: determine the window's size;
* line 6: initialize the *pygame* environment;
* line 7: prepare the application window and set its size;
* line 8: make an object representing the default font of size 48 points;
* line 9: make an object representing a given text – the text will be anti-aliased (True) and white (255,255,255)
* line 10: insert the text into the (currently invisible) screen buffer;
* line 11: flip the screen buffers to make the text visible;
* line 12: the *pygame* main loop starts here;
* line 13: get a list of all pending *pygame* events;
* lines 14 through 16: check whether the user has closed the window or clicked somewhere inside it or pressed any key;
* line 15: if yes, stop executing the code.



# How to use *pip*: continued

This is what we expect from our impressive code:



The pip install has two important additional abilities:

* it is able to **update** a locally installed package – e.g., if you want to make sure that you’re using the latest version of a particular package, you can run the following command:
* pip install -U package\_name

where -U means update. Note: this form of the command makes use of the --user option for the same purpose as presented previously;

* it is able to **install a user-selected version** of a package (*pip* installs the **newest** available version by default); to achieve this goal you should use the following syntax:
* pip install package\_name==package\_version

(note the double equals sign) e.g.,

pip install pygame==1.9.2

**How to use *pip*: continued**

If any of the currently installed packages are **no longer needed** and you want to get rid of them, *pip* will be useful, too. Its uninstall command will execute all the needed steps.

The required syntax is clear and simple:

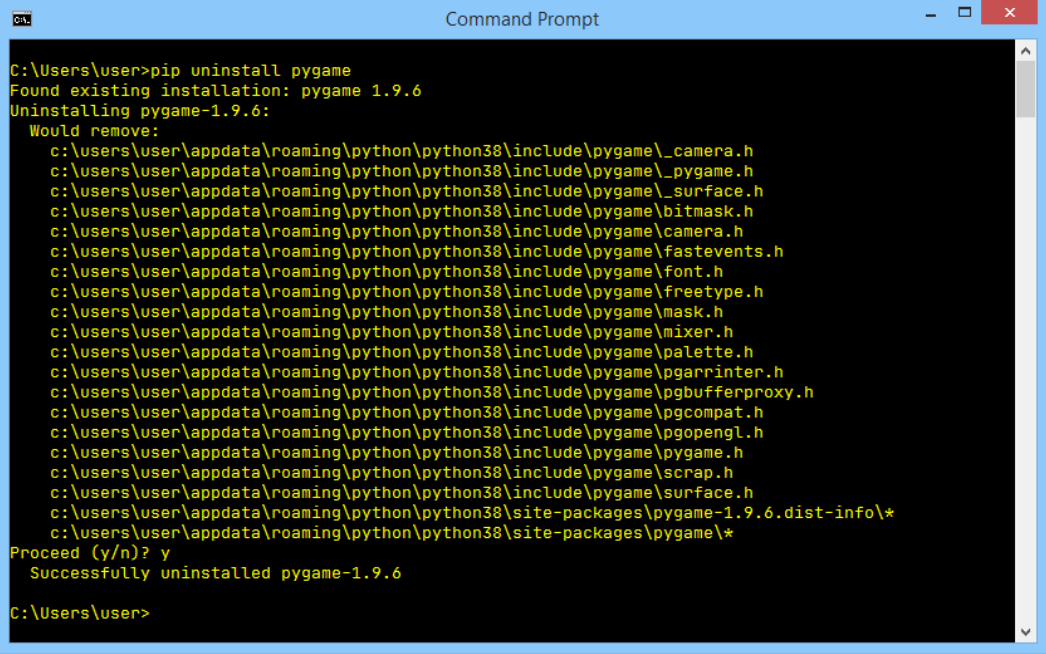
pip uninstall package\_name

so if you don't want *pygame* anymore you can execute the following command:

pip uninstall pygame

*Pip* will want to know if you’re sure about the choice you're making – be prepared to give the right answer.

The process looks like this:



# Use *pip*!

*Pip*'s capabilities don't end here, but the command set we've presented to you is enough to start successfully managing packages that aren't a part of the regular Python installation.

We hope we’ve encouraged you to carry out your own experiments with *pip* and the Python package universe. PyPI invites you to dive into its extensive resources.

Some say that one of the most important programming virtues is **laziness**. Don't get us wrong – we don't want you to spend all day napping on the couch and dreaming of Python code.

A lazy programmer is a programmer who looks for existing solutions and analyzes the available code before they start to develop their own software from scratch.

This is why PyPI and *pip* exist – use them!

**Key takeaways**

1. A **repository** (or **repo** for short) designed to collect and share free Python code exists and works under the name **Python Package Index** (**PyPI**) although it's also likely that you come across a very niche name **The Cheese Shop**. The Shop's website is available at <https://pypi.org/>.

2. To make use of The Cheese Shop the specialized tool has been created and its name is **pip** (***p****ip****i****nstalls****p****ackages* while *pip* stands for... ok, don't mind). As pip may not be deployed as a part of standard Python installation, it is possible that you will need to install it manually. Pip is a console tool.

3. To check pip's version one the following commands should be issued:

pip --version

or

pip3 --version

Check yourself which of these works for you in your OS' environment.

4. List of main **pip** activities looks as follows:

* pip help *operation* - shows brief pip's description;
* pip list - shows list of currently installed packages;
* pip show *package\_name* - shows *package\_name* info including package's dependencies;
* pip search *anystring* - searches through PyPI directories in order to find packages which name contains *anystring*;
* pip install *name* - installs *name* system-wide (expect problems when you don't have administrative rights);
* pip install --user *name* - install *name* for you only; no other your platform's user will be able to use it;
* pip install -U *name* - updates previously installed package;
* pip uninstall *name* - uninstalls previously installed package;

**Exercise 1**

Where does the name "The Cheese Shop" come from?

Check

It's a reference to an old Monty Python's sketch of the same name.

**Exercise 2**

Why should I ensure which one of *pip* and *pip3* works for me?

Check

When Python 2 and Python 3 coexist in your OS, it's likely that *pip* identifies the instance of pip working with Python 2 packages only.

**Exercise 3**

How can I determine if my *pip* works with either Python 2 or Python 3?

Check

pip --version will tell you that.

**Exercise 4**

Unfortunately, I don't have administrative right. What should I do to install a package system-wide?

Check

You have to ask your *sysadmin* - don't try to hack your OS!

**Congratulations! You have completed *PE2: Module 1*.**

Well done! You've reached the end of Module 1 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 1:

* working with Python modules; importing, creating, and using modules;
* using selected Python STL modules (*math*, *random*, and *platform*)
* constructing and using packages in Python;
* PIP (Python Package Installer.)

You are now ready to take the module quiz and attempt the final challenge: Module 1 Test, which will help you gauge what you've learned so far.