

# Useful Modules

## 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:

<code>__doc__</code>	<code>__loader__</code>	<code>__name__</code>	<code>__package__</code>	<code>__spec__</code>	<code>acos</code>	<code>acosh</code>	<code>asin</code>			
<code>asinh</code>	<code>atan</code>	<code>atan2</code>	<code>atanh</code>	<code>ceil</code>	<code>copysign</code>	<code>cos</code>	<code>cosh</code>	<code>degrees</code>	<code>e</code>	<code>erf</code> <code>erfc</code>
<code>exp</code>	<code>expm1</code>	<code>fabs</code>	<code>factorial</code>	<code>floor</code>	<code>fmod</code>	<code>frexp</code>	<code>feum</code>	<code>gamma</code>	<code>hypot</code>	<code>isfinite</code>
<code>isinf</code>	<code>isnan</code>	<code>ldexp</code>	<code>lgamma</code>	<code>log</code>	<code>log10</code>	<code>log1p</code>	<code>log2</code>	<code>modf</code>	<code>pi</code>	<code>pow</code> <code>radians</code> <code>sin</code>
<code>sinh</code>	<code>sqrt</code>	<code>tan</code>	<code>tanh</code>	<code>trunc</code>						

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.

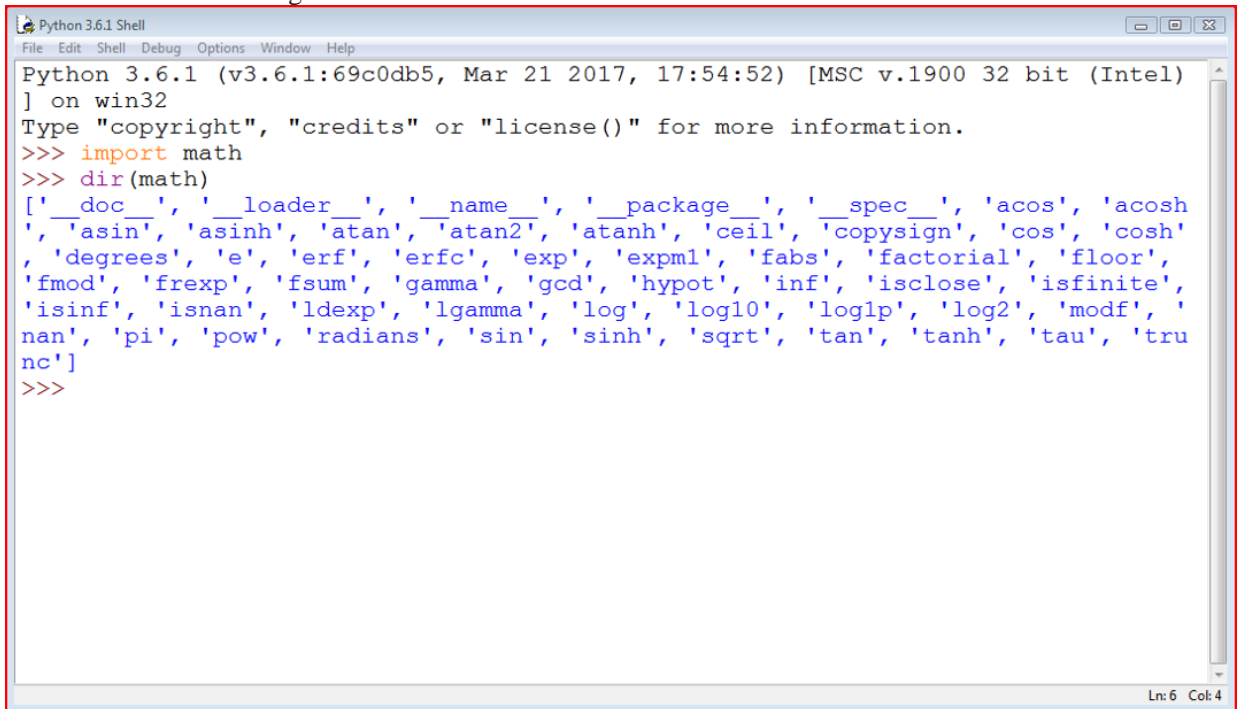
Fortunately, you can execute the function **directly in the Python console** (IDLE), without needing to write and run a separate script.

This is how it can be done:

```
import math
```

```
dir(math)
```

You should see something similar to this:



```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> dir(math)
['_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', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
```

## 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  $\pi$ ;
- `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 below.

```
1 from math import pi, radians, degrees, sin, cos, tan, asin
2
3 ad = 90
4 ar = radians(ad)
5 ad = degrees(ar)
6
7 print(ad == 90.)
8 print(ar == pi / 2.)
9 print(sin(ar) / cos(ar) == tan(ar))
10 print(asin(sin(ar)) == ar)
```

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  $e^x$ ;
- `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  $x^y$  (mind the domains)

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

Look at the code below. Can you predict its output?

```
1 from math import e, exp, log
2
3 print(pow(e, 1) == exp(log(e)))
4 print(pow(2, 2) == exp(2 * log(2)))
5 print(log(e, e) == exp(0))
```

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 below.

```
1 from math import ceil, floor, trunc
2
3 x = 1.4
4 y = 2.6
5
6 print(floor(x), floor(y))
7 print(floor(-x), floor(-y))
8 print(ceil(x), ceil(y))
9 print(ceil(-x), ceil(-y))
10 print(trunc(x), trunc(y))
11 print(trunc(-x), trunc(-y))
```

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 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:  $0.0 \leq x < 1.0$ .

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.

```
1 from random import random
2
3 for i in range(5):
4     print(random())
```

Run the program.

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
```

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.

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 below.

```
1 from random import randrange, randint
2
3 print(randrange(1), end=' ')
4 print(randrange(0, 1), end=' ')
5 print(randrange(0, 1, 1), end=' ')
6 print(randint(0, 1))
```

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.

```
1 from random import randint
2
3 for i in range(10):
4     print(randint(1, 10), end=',')
```

The program very likely outputs a set of numbers in which some elements are not unique.

This is what we got in one of the launches:

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

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=1)`

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 (which defaults to `1`) "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
```

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

```
print(choice(lst))
```

```
print(sample(lst, 5))
```

```
print(sample(lst, 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` 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

```
1 from platform import platform
2
3 print(platform())
4 print(platform(1))
5 print(platform(0, 1))
```

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
```

### **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
```

### **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
```

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

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

```
1 from platform import machine
2
3 print(machine())
```

on three different platforms:

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

```
x86
```

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

```
x86_64
```

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

```
armv7l
```

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

Once again, we ran the sample program

```
1 from platform import processor
2
3 print(processor())
```

on three different platforms:

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

```
x86
```

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

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

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

```
armv7l
```

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

```
1 from platform import system
2
3 print(system())
```

Our example platforms presented themselves like this:

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

```
Windows
```

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

```
Linux
```

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

```
Linux
```

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

Run the code and check its output.

```
1 from platform import version
2
3 print(version())
```

This is what we got:

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

```
6.0.6002
```

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

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

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

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

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

```
1 from platform import python_implementation, python_version_tuple
2
3 print(python_implementation())
4
5 for atr in python_version_tuple():
6     print(atr)
```

produced the following output:

```
CPython
3
6
4
```

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.

 Python » English » 3.7.2rc1 » Documentation »

## Python Module Index

[\\_](#)[a](#)[b](#)[c](#)[d](#)[e](#)[f](#)[g](#)[h](#)[i](#)[j](#)[k](#)[l](#)[m](#)[n](#)[o](#)[p](#)[q](#)[r](#)[s](#)[t](#)[u](#)[v](#)[w](#)[x](#)[z](#)

<a href="#">_future_</a>	<i>Future statement definitions</i>
<a href="#">_main_</a>	<i>The environment where the top-level script is run.</i>
<a href="#">_dummy_thread</a>	<i>Drop-in replacement for the <a href="#">_thread</a> module.</i>
<a href="#">_thread</a>	<i>Low-level threading API.</i>
<b>a</b>	
<a href="#">abc</a>	<i>Abstract base classes according to PEP 3119.</i>
<a href="#">aifc</a>	<i>Read and write audio files in AIFF or AIFC format.</i>
<a href="#">argparse</a>	<i>Command-line option and argument parsing library.</i>
<a href="#">array</a>	<i>Space efficient arrays of uniformly typed numeric values.</i>
<a href="#">ast</a>	<i>Abstract Syntax Tree classes and manipulation.</i>
<a href="#">asynchat</a>	<i>Support for asynchronous command/response protocols.</i>
<a href="#">asyncio</a>	<i>Asynchronous I/O.</i>
<a href="#">asyncore</a>	<i>A base class for developing asynchronous socket handling services.</i>
<a href="#">atexit</a>	<i>Register and execute cleanup functions.</i>
<a href="#">audioop</a>	<i>Manipulate raw audio data.</i>
<b>b</b>	
<a href="#">base64</a>	<i>RFC 3548: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85</i>

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