Python: 7th lesson – Working With External Libraries

Imports:

So far, we've talked about types and functions which are built-in to the language. But one of the best things about Python (especially if you're a data scientist) is the vast number of high-quality custom libraries that have been written for it.

Some of these libraries are in the "standard library", meaning you can find them anywhere you run Python. Other libraries can be easily added, even if they aren't always shipped with Python. Either way, we'll access this code with imports.

We'll start our example by importing math from the standard library.

import math

print("It's math! It has type {}".format(type(math)))

It's math! It has type <class 'module'>

math is a module. A module is just a collection of variables (a namespace, if you like) defined by someone else. We can see all the names in math using the built-in function dir().

print(dir(math))

['\_\_doc\_\_', '\_\_file\_\_', '\_\_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', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

We can access these variables using dot syntax. Some of them refer to simple values, like math.pi:

print("pi to 4 significant digits = {:.4}".format(math.pi))

pi to 4 significant digits = 3.142

But most of what we'll find in the module are functions, like math.log:

math.log(32, 2)

5.0

Of course, if we don't know what math.log does, we can call help() on it:

help(math.log)

Help on built-in function log in module math:

log(...)

log(x, [base=math.e])

Return the logarithm of x to the given base.

If the base not specified, returns the natural logarithm (base e) of x.

We can also call help() on the module itself. This will give us the combined documentation for all the functions and values in the module (as well as a high-level description of the module).

Other import syntax:

If we know we'll be using functions in math frequently we can import it under a shorter alias to save some typing (though in this case "math" is already pretty short).

import math as mt

mt.pi

3.141592653589793

You may have seen code that does this with certain popular libraries like Pandas, Numpy, Tensorflow, or Matplotlib. For example, it's a common convention to import numpy as np and import pandas as pd. The as simply renames the imported module. It's equivalent to doing something like:

import math

mt = math

Wouldn't it be great if we could refer to all the variables in the math module by themselves? i.e. if we could just refer to pi instead of math.pi or mt.pi? Good news: we can do that.

from math import \*

print(pi, log(32, 2))

3.141592653589793 5.0

import \* makes all the module's variables directly accessible to you (without any dotted prefix).

Bad news: some purists might grumble at you for doing this.

Worse: they kind of have a point.

What has happened? It worked before!

These kinds of "star imports" can occasionally lead to weird, difficult-to-debug situations. The problem in this case is that the math and numpy modules both have functions called log, but they have different semantics. Because we import from numpy second, its log overwrites (or "shadows") the log variable we imported from math.

A good compromise is to import only the specific things we'll need from each module:

from math import log, pi

from numpy import as array

Submodules:

We've seen that modules contain variables which can refer to functions or values. Something to be aware of is that they can also have variables referring to other modules.

import numpy

print("numpy.random is a", type(numpy.random))

print("it contains names such as...",

dir(numpy.random)[-15:]

)

numpy.random is a <class 'module'>

it contains names such as... ['seed', 'set\_state', 'shuffle', 'standard\_cauchy', 'standard\_exponential', 'standard\_gamma', 'standard\_normal', 'standard\_t', 'test', 'triangular', 'uniform', 'vonmises', 'wald', 'weibull', 'zipf']

So if we import numpy as above, then calling a function in the random "submodule" will require two dots.

# Roll 10 dice

rolls = numpy.random.randint(low=1, high=6, size=10)

rolls

array([3, 4, 3, 4, 5, 5, 2, 1, 3, 3])