Python Home Page (https://www.kaggle.com/learn/python)

In this lesson, I'll be talking about **imports** in Python, giving some tips for working with unfamiliar libraries (and the objects they return), and digging into the guts of Python just a bit to talk about **operator overloading**.

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. Others 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.

In [1]:
```
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()`.

In [2]:
```
print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spe
c__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil
', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'ex
pm1', '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']
```

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

In [3]:
```
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`:

In [4]:
```
math.log(32, 2)
```

Out[4]:
```
5.0
```

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

In [5]:
```
help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) o
f 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). Click the "output" button to see the whole `math` help page.

In [6]:
```
help(math)
```

Output

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).

In [7]:
```
import math as mt
mt.pi
```

Out[7]:
```
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:

In [8]:
```
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.

In [9]:
```
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.

In [10]:
```
from math import *
from numpy import *
print(pi, log(32, 2))
```

```
---------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-10-5045b296ad83> in <module>
      1 from math import *
      2 from numpy import *
----> 3 print(pi, log(32, 2))

TypeError: return arrays must be of ArrayType
```

What the what? But 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:

In [11]:
```
from math import log, pi
from numpy import asarray
```

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*.

In [12]:

```
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... ['set_state', 'shuffle', 'standard_cauchy'
, 'standard_exponential', 'standard_gamma', 'standard_normal', 'standar
d_t', 'test', 'triangular', 'uniform', 'vonmises', 'wald', 'warnings',
'weibull', 'zipf']
```

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

In [13]:

```
# Roll 10 dice
rolls = numpy.random.randint(low=1, high=6, size=10)
rolls
```

Out[13]:

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

Oh the places you'll go, oh the objects you'll see

So after 6 lessons, you're a pro with ints, floats, bools, lists, strings, and dicts (right?).

Even if that were true, it doesn't end there. As you work with various libraries for specialized tasks, you'll find that they define their own types which you'll have to learn to work with. For example, if you work with the graphing library `matplotlib`, you'll be coming into contact with objects it defines which represent Subplots, Figures, TickMarks, and Annotations. `pandas` functions will give you DataFrames and Series.

In this section, I want to share with you a quick survival guide for working with strange types.

Three tools for understanding strange objects

In the cell above, we saw that calling a `numpy` function gave us an "array". We've never seen anything like this before (not in this course anyways). But don't panic: we have three familiar builtin functions to help us here.

1: **`type()`** (what is this thing?)

In [14]:
```
type(rolls)
```

Out[14]:
```
numpy.ndarray
```

2: **`dir()`** (what can I do with it?)

In [15]:

```
print(dir(rolls))
```

```
['T', '__abs__', '__add__', '__and__', '__array__', '__array_finalize__
', '__array_function__', '__array_interface__', '__array_prepare__', '_
_array_priority__', '__array_struct__', '__array_ufunc__', '__array_wra
p__', '__bool__', '__class__', '__complex__', '__contains__', '__copy__
', '__deepcopy__', '__delattr__', '__delitem__', '__dir__', '__divmod__
', '__doc__', '__eq__', '__float__', '__floordiv__', '__format__', '__g
e__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd_
_', '__iand__', '__ifloordiv__', '__ilshift__', '__imatmul__', '__imod_
_', '__imul__', '__index__', '__init__', '__init_subclass__', '__int__'
, '__invert__', '__ior__', '__ipow__', '__irshift__', '__isub__', '__it
er__', '__itruediv__', '__ixor__', '__le__', '__len__', '__lshift__', '
__lt__', '__matmul__', '__mod__', '__mul__', '__ne__', '__neg__', '__ne
w__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmo
d__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rl
shift__', '__rmatmul__', '__rmod__', '__rmul__', '__ror__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '_
_setattr__', '__setitem__', '__setstate__', '__sizeof__', '__str__', '_
_sub__', '__subclasshook__', '__truediv__', '__xor__', 'all', 'any', 'a
rgmax', 'argmin', 'argpartition', 'argsort', 'astype', 'base', 'byteswa
p', 'choose', 'clip', 'compress', 'conj', 'conjugate', 'copy', 'ctypes'
, 'cumprod', 'cumsum', 'data', 'diagonal', 'dot', 'dtype', 'dump', 'dum
ps', 'fill', 'flags', 'flat', 'flatten', 'getfield', 'imag', 'item', 'i
temset', 'itemsize', 'max', 'mean', 'min', 'nbytes', 'ndim', 'newbyteor
der', 'nonzero', 'partition', 'prod', 'ptp', 'put', 'ravel', 'real', 'r
epeat', 'reshape', 'resize', 'round', 'searchsorted', 'setfield', 'setf
lags', 'shape', 'size', 'sort', 'squeeze', 'std', 'strides', 'sum', 'sw
apaxes', 'take', 'tobytes', 'tofile', 'tolist', 'tostring', 'trace', 't
ranspose', 'var', 'view']
```

In [16]:

```
# What am I trying to do with this dice roll data? Maybe I want the aver
age roll, in which case the "mean"
# method looks promising...
rolls.mean()
```

Out[16]:

```
2.2
```

In [17]:

```
# Or maybe I just want to get back on familiar ground, in which case I m
ight want to check out "tolist"
rolls.tolist()
```

Out[17]:

```
[4, 1, 2, 2, 1, 3, 1, 1, 5, 2]
```

3: **`help()`** (tell me more)

In [18]:
```
# That "ravel" attribute sounds interesting. I'm a big classical music f
an.
help(rolls.ravel)
```

```
Help on built-in function ravel:

ravel(...) method of numpy.ndarray instance
    a.ravel([order])

    Return a flattened array.

    Refer to `numpy.ravel` for full documentation.

    See Also
    --------
    numpy.ravel : equivalent function

    ndarray.flat : a flat iterator on the array.
```

In [19]:
```
# Okay, just tell me everything there is to know about numpy.ndarr
ay
# (Click the "output" button to see the novel-length output)
help(rolls)
```

Output

(Of course, you might also prefer to check out the online docs (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.ndarray.html))

Operator overloading

What's the value of the below expression?

In [20]:
```
[3, 4, 1, 2, 2, 1] + 10
```

```
---------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-20-a2508fc27c2b> in <module>
----> 1 [3, 4, 1, 2, 2, 1] + 10

TypeError: can only concatenate list (not "int") to list
```

What a silly question. Of course it's an error.

But what about...

In [21]:
```
rolls + 10
```

Out[21]:
```
array([14, 11, 12, 12, 11, 13, 11, 11, 15, 12])
```

We might think that Python strictly polices how pieces of its core syntax behave such as `+`, `<`, `in`, `==`, or square brackets for indexing and slicing. But in fact, it takes a very hands-off approach. When you define a new type, you can choose how addition works for it, or what it means for an object of that type to be equal to something else.

The designers of lists decided that adding them to numbers wasn't allowed. The designers of `numpy` arrays went a different way (adding the number to each element of the array).

Here are a few more examples of how `numpy` arrays interact unexpectedly with Python operators (or at least differently from lists).

In [22]:

```
# At which indices are the dice less than or equal to 3?
rolls <= 3
```

Out[22]:

```
array([False,  True,  True,  True,  True,  True,  True,  True, False,
        True])
```

In [23]:

```
xlist = [[1,2,3],[2,4,6],]
# Create a 2-dimensional array
x = numpy.asarray(xlist)
print("xlist = {}\nx =\n{}".format(xlist, x))
```

```
xlist = [[1, 2, 3], [2, 4, 6]]
x =
[[1 2 3]
 [2 4 6]]
```

In [24]:

```
# Get the last element of the second row of our numpy array
x[1,-1]
```

Out[24]:

```
6
```

In [25]:
```
# Get the last element of the second sublist of our nested list?
xlist[1,-1]
```

```
---------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-25-e2f4c7f35788> in <module>
      1 # Get the last element of the second sublist of our nested lis
t?
----> 2 xlist[1,-1]

TypeError: list indices must be integers or slices, not tuple
```

numpy's `ndarray` type is specialized for working with multi-dimensional data, so it defines its own logic for indexing, allowing us to index by a tuple to specify the index at each dimension.

When does 1 + 1 not equal 2?

Things can get weirder than this. You may have heard of (or even used) tensorflow, a Python library popularly used for deep learning. It makes extensive use of operator overloading.

In [26]:
```
import tensorflow as tf
# Create two constants, each with value 1
a = tf.constant(1)
b = tf.constant(1)
# Add them together to get...
a + b
```

Out[26]:
```
<tf.Tensor 'add:0' shape=() dtype=int32>
```

`a + b` isn't 2, it is (to quote tensorflow's documentation)...

> a symbolic handle to one of the outputs of an `Operation`. It does not hold the values of that operation's output, but instead provides a means of computing those values in a TensorFlow `tf.Session`.

It's important just to be aware of the fact that this sort of thing is possible and that libraries will often use operator overloading in non-obvious or magical-seeming ways.

Understanding how Python's operators work when applied to ints, strings, and lists is no guarantee that you'll be able to immediately understand what they do when applied to a tensorflow `Tensor`, or a numpy `ndarray`, or a pandas `DataFrame`.

Once you've had a little taste of DataFrames, for example, an expression like the one below starts to look appealingly intuitive:

```
# Get the rows with population over 1m in South America
df[(df['population'] > 10**6) & (df['continent'] == 'South America')]
```

But why does it work? The example above features something like **5** different overloaded operators. What's each of those operations doing? It can help to know the answer when things start going wrong.

Curious how it all works?

Have you ever called `help()` or `dir()` on an object and wondered what the heck all those names with the double-underscores were?

In [27]:
```
print(dir(list))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute
__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__in
it__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof
__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

This turns out to be directly related to operator overloading.

When Python programmers want to define how operators behave on their types, they do so by implementing methods with special names beginning and ending with 2 underscores such as `__lt__`, `__setattr__`, or `__contains__`. Generally, names that follow this double-underscore format have a special meaning to Python.

So, for example, the expression `x in [1, 2, 3]` is actually calling the list method `__contains__` behind-the-scenes. It's equivalent to (the much uglier) `[1, 2, 3].__contains__(x)`.

If you're curious to learn more, you can check out Python's official documentation (https://docs.python.org/3.4/reference/datamodel.html#special-method-names), which describes many, many more of these special "underscores" methods.

We won't be defining our own types in these lessons (if only there was time!), but I hope you'll get to experience the joys of defining your own wonderful, weird types later down the road.

Your turn!

Head over to **the final coding exercise (https://www.kaggle.com/kernels/fork/1275190)** for one more round of coding questions involving imports, working with unfamiliar objects, and, of course, more gambling.

Python Home Page (https://www.kaggle.com/learn/python)

Have questions or comments? Visit the Learn Discussion forum (https://www.kaggle.com/learn-forum/161283) to chat with other Learners.