Lesson 8: Modules and Packages

We're going to switch gears just a bit here and talk about a few different-but-vaguely-related things.

First up:

Modules

In the world of programming, we care a lot about making code reusable. In most cases, we write code so that it can be reusable for ourselves. But sometimes we share code that's helpful across a broad range of situations. (This is why comments are so important.)

Let's explore how to use tools other people have built in Python that are not included automatically for you when you install Python. Python allows us to package code into *files* or *sets of files* called **modules**.

> **Module:** A module is a collection of Python declarations intended broadly to be used as a tool. Modules are also often referred to as "libraries" or "packages" — in Python, a package is really a directory that holds a collection of modules.

The Python Standard Library has lots of built-in modules that contain useful functions and data types for doing specific tasks. You can also use modules from outside the standard library. And you will undoubtedly write your own modules!

A *module* is contained in a file that ends with `.py`. This file can have classes, functions, and other objects. We will not discuss defining your own classes in this bootcamp, so your modules will essentially just contain functions for now.

Usually, to use a module in a script, the basic syntax you need at the top of that script is:

In [ ]:
```
from module_name import object_name
```

Often, a library will include a lot of code that you don't need or that may slow down your program or conflict with existing code. Because of this, it makes sense to only import what you need when you need it.

A common library that comes as part of the Python Standard Library is `datetime`. `datetime` lets you work with dates and times in Python... as the name should imply.

But to use it, first we have to import it:

In [1]:
```
from datetime import datetime
```

Now that we have datetime, we can play with it. Let's first get the current time.

In [2]:
```
current_time = datetime.now()
print(current_time)
```

```
2024-01-16 09:57:07.402691
```

That right there is the date (format year-month-day), followed by the precise time down to the millisecond (format hour : minute : second.millisecond).

Cool, right?

But of course there's hundred of Python modules. Check them all out here.

We can also play around with another module: `random` , which generates numbers or selects items at random.

And with `random` , we'll be using more than one piece of the module's functionality, so the import syntax will look like:

In [3]:
```
import random
```

Right now we'll work with two common `random` functions

- `random.choice()` : takes a list as an argument and returns a number from the list
- `random.randint()` : takes two numbers as arguments and generates a random number between the two numbers you passed in

Let's take randomness to a whole new level by picking a random number from a list of randomly generated numbers between 1 and 100.

In [4]:
```
random_list = [] # creates an empty list of arbitrary length (useful for when you don't kn
random_list = [random.randint(1,101) for i in range(101)] # assigns a random number betwee
super_random_number = random.choice(random_list) # picks an entry from random_list... at r
print(super_random_number)
```

```
87
```

In [6]:
```
super_random_number = random.choice(random_list) # rerun the function to get a new random
print(super_random_number)
```

```
84
```

In [7]:
```
super_random_number = random.choice(random_list)
super_random_number
```

Out[7]:
```
9
```

Modules Python Namespaces

Notice that when we want to invoke the `randint()` function we call `random.randint()` . This is default behavior where Python offers a *namespace* for the module. A **namespace** isolates the functions, classes, and variables defined in the module from the code in the file doing the importing. Your *local namespace* is where your code is run.

Python defaults to naming the namespace after the module being imported, but sometimes this name could be ambiguous or lengthy. Sometimes, the module's name could also conflict with an object you have defined within your local namespace. Fortunately, this name can be altered by aliasing using the `as` keyword:

In [ ]:
```
import module_name as name_you_pick_for_the_module
```

> **Aliasing** is most often done if the name of the library is long and typing the full name every time you want to use one of its functions is laborious.

You might also occasionally encounter `import *`. The `*` is known as a "wildcard" and matches anything and everything. This syntax is considered dangerous because it could pollute our local namespace. Pollution occurs when the same name could apply to two possible things. For example, if you happen to have a function `floor()` focused on floor tiles, using `from math import *` would also import a function `floor()` that rounds down floats.

So be careful.

> **Import Python Modules:** The Python import statement can be used to import Python modules from other files. Modules can be imported in three different ways: `import module`, `from module import functions`, or from `module import *`, `from module import *` is discouraged, as it can lead to a cluttered local namespace and can make the namespace unclear.

But what happens if we import a module or function that conflicts with an object defined in out local namespace?

Short answer: If you import a module or function that conflicts with the name of an object defined in your local namespace, then the object defined in your local namespace will overwrite the one that was imported. However, it will only overwrite if the object in your namespace is defined after the import statement in the code.

Example:

In [8]:
```
from math import sqrt

print(sqrt(4)) # 2.0

# Defining it after the import statement  l
# overwrite the function with the same name
def sqrt(n):
  return n * 100

print(sqrt(4)) # 400
```

```
2.0
400
```

Let's combine our new knowledge of the `random` library with another fun library called

`matplotlib` , which allows plotting of Python code in 2D. Use a new random function `random.sample()` that takes a range and a number as its arguments. It'll return the specified number of random numbers from that range.

In [ ]:
```
new_list = random.sample(list, k)  # random.sample takes a list and randomly selects k ite
```

In [9]:
```
nums = [1, 2, 3, 4, 5]
sample_nums = random.sample(nums, 3)
print(sample_nums)
```

```
[3, 2, 4]
```

Importing Modules & Organization

Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants, grouped in the following order:

- standard library imports
- related third party imports
- local application/library specific imports

Ideally, you'd also put a blank line between each group of imports.

You should follow this guide, Jupyter notebooks included, with the first code cell having all of the imports needed. Therefore, going forward all of these lessons will have all necessary imports at the top of the notebook. The only exception is when we're explicitly demonstrating a concept that requires an import.

Imports and updates

Once you have imported a module or package, the interpreter stores its contents in memory. You can't update the contents of the package and expect the interpreter to know about the changes. You'll need to restart the kernel and then import the package again in a fresh instance. This can seem annoying, but it's good design. It ensures that code you're currently running doesn't change as you go through executing a notebook. However, when developing modules, it can be more convenient to have an imported module be updated as you run through the notebook as you're editing. To enable this, you can use the autoreload extension. To activate it, run the following in a code cell.

In [ ]:
```
%load_ext autoreload
%autoreload 2
```

Now, whenever you run a cell, imported packages and modules will be automatically reloaded.

Packages

A **package** contains several related modules that are all grouped together under one name. Some well-known and often used packages are the NumPy, SciPy, and Pandas packages. I'm sure you'll find your favorites as your progress in your Python skills, applying the language to your particular needs. But for now, we'll start with NumPy.

Say there's a list of numbers and we want to compute the mean. This happens all the time; you repeat a measurement multiple times and you want to compute the mean. We could write a function to do this.

In [10]:
```
def mean(values):
    """Compute the mean of a sequence of numbers."""
    return sum(values) / len(values)
```

In [11]:
```
print(mean([1, 2, 3, 4, 5]))
print(mean((4.5, 1.2, -1.6, 9.0)))
```

```
3.0
3.275
```

It seems to work fairly well.

In addition to the mean, you might also want to compute the median, the standard deviation, etc. These are really common tasks. Remember: don't reinvent the whell. If you want to do something that seems really common, a good programmer (or a team of them) probably already wrote something to do that. Means, medians, standard deviations, and lots and lots and lots of other numerical things are included in the `Numpy` module. If you haven't installed `numpy` before, just type into your command console: `pip install numpy` and let it run.

Then, to access `numpy` every other time, just import it!

> A similar `pip` installation will work for other Python packages as well. If you're ever unsure, find the documentation page for a package and read through the installation instructions.

In [12]:
```
import numpy
```

That's it! Now you've got the `numpy` module available for use. Remember, in Python everything is an object, so if you want to access the methods and attributes available in the `numpy` module, just use dot syntax. In a Jupyter notebook or in the JupyterLab console, type `numpy.` (note the dot) and hit tab, to see what is available. For Numpy, there is a huge number of options!

In [ ]:
```
numpy.
```

Let's try to use Numpy's `numpy.mean()` function to compute a mean.

In [13]:
```
print(numpy.mean([1, 2, 3, 4, 5]))
print(numpy.mean((4.5, 1.2, -1.6, 9.0)))
```

```
3.0
3.275
```

Note that we get the same values as from our function up above! Now, we can use the `numpy.median()` function to compute the median.

In [14]:
```
print(numpy.median([1, 2, 3, 4, 5]))
print(numpy.median((4.5, 1.2, -1.6, 9.0)))
```

```
3.0
2.85
```

This is nice. It gives the median, including when we have an even number of elements in the sequence of numbers, in which case it automatically interpolates. It is really important to know that it does this interpolation, since if you aren't expecting it, it can give unexpected results. So, here is an important piece of advice:

Always check the doc strings of functions.

Access the doc string of the `numpy.median()` function in JupyterLab by typing

In [15]:
```
numpy.median?
```

```
Signature:       numpy.median(a, axis=None, out=None, overwrite_input=False, keepdims=Fals
e)
Call signature:  numpy.median(*args, **kwargs)
Type:            _ArrayFunctionDispatcher
String form:     <function median at 0x000001E8249FB740>
File:            c:\users\ayla\appdata\local\programs\python\python312\lib\site-packages\nu
mpy\lib\function_base.py
Docstring:
Compute the median along the specified axis.

Returns the median of the array elements.

Parameters
----------
a : array_like
    Input array or object that can be converted to an array.
axis : {int, sequence of int, None}, optional
    Axis or axes along which the medians are computed. The default
    is to compute the median along a flattened version of the array.
    A sequence of axes is supported since version 1.9.0.
out : ndarray, optional
    Alternative output array in which to place the result. It must
    have the same shape and buffer length as the expected output,
    but the type (of the output) will be cast if necessary.
overwrite_input : bool, optional
   If True, then allow use of memory of input array `a` for
   calculations. The input array will be modified by the call to
   `median`. This will save memory when you do not need to preserve
   the contents of the input array. Treat the input as undefined,
   but it will probably be fully or partially sorted. Default is
   False. If `overwrite_input` is ``True`` and `a` is not already an
   `ndarray`, an error will be raised.
keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the original `arr`.

    .. versionadded:: 1.9.0

Returns
-------
median : ndarray
    A new array holding the result. If the input contains integers
    or floats smaller than ``float64``, then the output data-type is
    ``np.float64``.  Otherwise, the data-type of the output is the
    same as that of the input. If `out` is specified, that array is
    returned instead.

See Also
--------
mean, percentile

Notes
-----
Given a vector ``V`` of length ``N``, the median of ``V`` is the
middle value of a sorted copy of ``V``, ``V_sorted`` - i
e., ``V_sorted[(N-1)/2]``, when ``N`` is odd, and the average of the
```

```
two middle values of ``V_sorted`` when ``N`` is even.

Examples
--------
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> np.median(a)
3.5
>>> np.median(a, axis=0)
array([6.5, 4.5, 2.5])
>>> np.median(a, axis=1)
array([7.,  2.])
>>> m = np.median(a, axis=0)
>>> out = np.zeros_like(m)
>>> np.median(a, axis=0, out=m)
array([6.5,  4.5,  2.5])
>>> m
array([6.5,  4.5,  2.5])
>>> b = a.copy()
>>> np.median(b, axis=1, overwrite_input=True)
array([7.,  2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.median(b, axis=None, overwrite_input=True)
3.5
>>> assert not np.all(a==b)
Class docstring:
Class to wrap functions with checks for __array_function__ overrides.

All arguments are required, and can only be passed by position.

Parameters
----------
dispatcher : function or None
    The dispatcher function that returns a single sequence-like object
    of all arguments relevant.  It must have the same signature (except
    the default values) as the actual implementation.
    If ``None``, this is a ``like=`` dispatcher and the
    ``_ArrayFunctionDispatcher`` must be called with ``like`` as the
    first (additional and positional) argument.
implementation : function
    Function that implements the operation on NumPy arrays without
    overrides.  Arguments passed calling the ``_ArrayFunctionDispatcher``
    will be forwarded to this (and the ``dispatcher``) as if using
    ``*args, **kwargs``.

Attributes
----------
_implementation : function
    The original implementation passed in.
```

This is where the documentation tells you that the median will be reported as the average of two middle values when the number of elements is even. Note that you could also read the documentation here, which is a bit easier to read.

Package management

Whenever you write your own module, it will be stored in the directory that you're currently working in, or the `pwd`. But what if you write a module that you want to use regardless of what directory your are in? To allow this kind of usage, you can use the `setuptools` module of the standard library to manage your packages. This is outside the scope of this bootcamp, but the documentation on Python packages and modules is a good place to start to understand the details of how this is done.

Third party packages

Standard Python installations come with the standard library. Numpy and other useful packages are not in the standard library. Outside of the standard library, there are several packages available. Several. Ha! There are currently over 239,000 packages available through the Python Package Index, PyPI. Usually, you can ask Google about what you're trying to do, and there's *usually* a third party module to help you do it. The most useful (for scientific computing) and thoroughly tested packages and modules are available using `conda`. Others can be installed using `pip`, just like what we did to install `numpy`.

Proceed to Assignment 8.