EASC2410 Lecture 5

Python Basics: 1-D Numpy Arrays, Matplotlib

Dr. Binzheng Zhang
Department of Earth Sciences

Review of Lecture 4

In Lecture 4, we learned:

- **Define Python functions**
- **Call Python functions**
- **Python Modules**

In Lecture 5, you will learn:

- **The NumPy module and 1-D NumPy arrays**
- **Making 1-D plots with Matplotlib**

Recall Python Lists

Define a list:

```
my_list = ['my', True, 1.0, 15, 2+3j]
```

- Defined using either the square brackets `[]` or the `list()` function
- Can have different data types
- Index slicing using something like `[2:5]`
- build-in methods working with lists, e.g., `index()`, `append()`, `remove()`, `sort()`, etc.

Let's try define two Python lists called "weight" and "height" of a bunch of your patients

```
In [4]: # here are the list of heights and weights
        height = [1.73, 1.68, 1.71, 1.89, 1.79]
        weight = [65.4, 59.2, 63.6, 88.4, 68.7]

        # print the types of the two lists - of course they are lists..
        type(height), type(weight)

Out[4]: (list, list)
```

Now if I want to calculate the body-mass index (BMI) of all the patients, shall I do this?

```
In [5]: bmi = weight / height ** 2

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-5-05b4eda41695> in <module>
        ----> 1 bmi = weight / height ** 2

        TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

So you can’t operator python lists as a single variable. What now?

Question: What can you do?

Recall: the **for-loop**

```
In [1]: # here are the list of heights and weights
height = [1.73, 1.68, 1.71, 1.89, 1.79]
weight = [65.4, 59.2, 63.6, 89.4, 68.7]

N = len(height) # len() function gives the length of the list, then assign to N
bmi = []        # define an empty list named "bmi" to store the values of each BMI
for i in range(N):
    bmi.append(weight[i]/height[i]**2) # calculate BMI for each element and append to the bmi li

print(bmi) # print the results

[21.85171572722109, 20.97505668934241, 21.750282138093777, 25.027294868564713, 21.44127836209
856]
```

What I did is to step through all the elements in the two Python lists using a **for loop** and the .append() method:

- First define a new list called “bmi”, make it empty: `bmi = []`
- Step through each element in the two lists using a for loop: `for i in range(N)`
- Calculate the BMI for each element using list indexing: `weight[i]/height[i]**2`
- Append the corresponding BMI value to the existing list `bmi`
- Done!

Or can use the zip( ) function to step through two lists at the same time:

```
In [5]: height = [1.73, 1.68, 1.71, 1.89, 1.79]
        weight = [65.4, 59.2, 63.6, 89.4, 68.7]

        bmi = [] # create an empty list for bmi
        for (i,j) in zip(weight, height): # here i will iterate through the list "weight",
                                          #  and j will interate through "height"
            bmi.append(i/j**2) # compute BMI

        print(bmi) # print the results

        [21.85171572722109, 20.97505668934241, 21.750282138093777, 25.027294868564713, 21.44127836209856]
```

What I did is to step through all the elements in the two Python lists at the same time using a **for loop** and the zip( ) function

- First define a new list called "bmi", make it empty: `bmi = []`
- Step through each element in the two lists using a for loop: `for (i,j) in zip(weight, height)`
- Calculate the BMI for each element using list indexing: `i/j**2`
- Append the corresponding BMI value to the existing list `bmi`
- Done!

So one for-loop combined with Python lists seems to get the job done. But…

- It's tedious - you need to write for-loops for this kind of calculations every time;
- It's inefficient - Python for-loops are quite slow in terms of computing time - it's an interpretive language
- There are better ways to do it!

Now let's see what we can do using the NumPy module

In [49]:
```
import numpy as np  # or any other variable e.g.:  N

# first let's define two Python lists called height and weight
height = [1.73, 1.68, 1.71, 1.89, 1.79]
weight = [65.4, 59.2, 63.6, 99.4, 98.7]

print(type(weight), type(height))

# now let's make corresponding numpy arrays using the .array() method
np_height = np.array(height)
np_weight = np.array(weight)

print(type(np_weight), type(np_height))

bmi = np_weight/np_height**2

print(type(bmi))
print(bmi)
```

```
<class 'list'> <class 'list'>
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
<class 'numpy.ndarray'>
[21.85171573 20.97505669 21.75028214 27.82676857 30.80428201]
```

Let's take a look at what that code does:

```
import numpy as np  # or any other variable e.g.:  N
```

Import the **numpy** library and give it a nickname called **np**

```
# first let's define two Python lists called height and weight
height = [1.73, 1.68, 1.71, 1.89, 1.79]
weight = [65.4, 59.2, 63.6, 99.4, 98.7]
```

Define two Python lists called **height** and **weight**

```
# now let's make corresponding numpy arrays using the .array() method
np_height = np.array(height)
np_weight = np.array(weight)
```

Now convert **height** and **weight** to be **numpy arrays** using the `np.array()` function, the two arrays are called **`np_height`** and **`np_weight`** now.

```
bmi = np_weight/np_height**2
```

Then calculate the bmi index using **`np_height`** and **`np_weight`** directly

So you can work with NumPy arrays directly like a single variable! the calculations are done in an element-wise way that you don't need to write for-loops anymore

Element-wise Operations

Codes:

```
import numpy as np  # or any other variable e.g.:  N

# first let's define two Python lists called height and weight
height = [1.73, 1.68, 1.71, 1.89, 1.79]
weight = [65.4, 59.2, 63.6, 99.4, 98.7]

print(type(weight), type(height))

# now let's make corresponding numpy arrays using the .array() method
np_height = np.array(height)
np_weight = np.array(weight)

print(type(np_weight), type(np_height))

bmi = np_weight/np_height**2

print(type(bmi))
print(bmi)
```

Outputs:

```
[21.85171573 20.97505669 21.75028214 27.82676857 30.80428201]
```

65.4/1.73**2 59.2/1.68**2 63.6/1.71**2 99.4/1.89**2 98.7/1.79**2

Why we need NumPy Arrays?

So the numpy arrays are better solutions to this kind of element-wise calculations for numerical lists:

- Numeric calculations in Python
- Numpy arrays are alternative to Python Lists
- Calculations over entire arrays
- Easy to use and very fast

You can work with your numpy arrays using most of the operators we've learned in previous lectures, Python knows that it's a numpy array, it steps through all the elements in a numpy array.

For example, if we want to know whether someone's BMI is greater or less than 25, simply do:

```
In [4]: fat = bmi > 25 # relational operation see whose bmi is greater than 25
        print(type(fat),fat)

        <class 'numpy.ndarray'> [False False False  True  True]
```

If we want to know the BMIs that are greater than 25 in the bmi list, simply do:

```
In [5]: bmi[fat] # slice to the value with BMI > 25

Out[5]: array([27.82676857, 30.80428201])
```

More operations for NumPy Arrays

Note that, to specify a condition, you can also make use of the logical operators or and and. For example you want to see if there's anyone whose weight is greater than 65 and height is less than 1.75, you can simply use the relational operator ">" and "<" we've learned before, and combine them with the logical operator "&" (and): (the "or" operator is "|")

```
In [62]: heavy_and_short = (np_weight > 65.0) & (np_height < 1.75) # find if someone is both heavier thar
         print(heavy_and_short)

[ True False False False False]
```

Now the array **heavy_and_short** consists of a series of booleans which tells you whether someone satisfies both of your criteria. To access the weight and height of that person, you can use the array **heavy_and_short** as your numpy array index:

```
In [63]: print( np_weight[heavy_and_short] ) # print the weight of the persion who's both heavy and short
         print( np_height[heavy_and_short] ) # print the weight of the persion who's both heavy and short

[65.4]
[1.73]
```

This property is very useful in filtering data!

More operations for NumPy Arrays

You can also use numpy *methods* to do simple statistics on numpy arrays, let's try the `min, max, mean, sum` and standard deviation functions

In [73]:
```
print (np_weight.min())  # print the minimum value      in the np_weight list
print (np_weight.max())  # print the maximim value      in the np_weight list
print (np_weight.mean()) # print the average value      in the np_weight list
print (np_weight.sum())  # print the sum of the weights in the np_weight list
print (np_weight.std())  # print the standard deviation in the np_weight list
```

```
59.2
99.4
77.26
386.3
17.906825514311574
```

Out[73]:
```
65.4
```

However the **median** *function* works in a slightly different way (but very simple):

In [74]:
```
print (np.median(np_weight))  # print the median value in the np_weight list
```

```
65.4
```

You can also use the **sort** *function* to generate an increase (or decrease) series of the numbers:

In [15]:
```
np.sort(bmi)
```

Out[15]:
```
array([20.97505669, 21.75028214, 21.85171573, 27.82676857, 30.80428201])
```

More Basics on Arrays and NumPy

1-D Arrays:

Name		Elements
`np_height`	=	`[1.73 1.68 1.71 1.89 1.79]`

Index	0	1	2	3	4

1-D NumPy arrays are the basic data type of the NumPy module which contains a series of numerical values with indices starting from zero. Unlike lists, NumPy arrays can have only **one** type of variables

Index Slicing Numpy Arrays:

```
np_height[index_start:index_stop]
```

Inclusive Exclusive

For example

```
np_height[1:4] =[1.68 1.71 1.89]
```

Creating Numpy Arrays:

What people often mean when they say that they are creating “empty” arrays is that they want to make use of initial placeholders, which you can fill up afterwards. You can initialize numpy arrays with ones or zeros, but you can also make arrays that get filled up with evenly spaced values, constant or random values.

Here are more ways to creat 1-D numpy arrays using the following functions:

In [19]:
```
# Create an array of ones
print( np.ones((4)) )

# Create an array of zeros
print( np.zeros((5),dtype=np.int16) )

# Create an array with random values
print( np.random.random((6)) )

# Create an empty array
print( np.empty((7)) )

# Create a full array
print( np.full(8,7) )

# Create an array of evenly-spaced values
print( np.arange(10,25,1) )

# Create an array of evenly-spaced values
print( np.linspace(0,2,9) )
```

```
[1. 1. 1. 1.]
[0 0 0 0 0]
[0.26298286 0.97498731 0.78752137 0.88933936 0.41598039 0.76355419]
[0. 0. 0. 0. 0. 0. 0.]
[7 7 7 7 7 7 7 7]
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
[0.   0.25 0.5  0.75 1.   1.25 1.5  1.75 2.  ]
```

More index slicing in 1-D Numpy Arrays:

```
In [47]: # define an numpy array called num, all integers, starts from 11, stops at 24
         num =  np.arange(11,25,1) # or if you do np.linspace(11,25,15)
         print(num)

         # access the second number in the array "num"
         print( num[1] )

         # access the last number:
         print( num[-1] )

         # access a subset of numbers from the 3rd to 10th
         print( num[2:10] )

         # access a subset of numbers from the 3rd all to 10th, but every other one
         print( num[2:10:2] )

         # access a subset of numbers from the 3rd all the way to the end
         print( num[2:] )
```

```
[11 12 13 14 15 16 17 18 19 20 21 22 23 24]
12
24
[13 14 15 16 17 18 19 20]
[13 15 17 19]
[13 14 15 16 17 18 19 20 21 22 23 24]
```

Pay attention to this one:

```
num[index_start:index_stop:index_increase]
```

index_start	index_stop	index_increase
Inclusive	Exclusive	delta i

Load the Numpy Module and Call NumPy Functions (and constants)

Recall in the last lecture, we used the math module to access pi and sqrt(), here we can access to the same constants and functions in numpy:

In [23]:
```
import numpy

# This import command makes all the functions in NumPy available to you for analyzing your data.
# Remember, you must call them with the numpy.FUNC() syntax unless you give it a nickname

print(numpy.pi)   # pi
print(numpy.sqrt(9)) # sqrt function
```

```
3.141592653589793
3.0
```

or:

In [25]:
```
# Here is another way to import a module:
import numpy as np  # or any other variable e.g.:  N
# This does the same as the first, but allows you to set NumPy as a variable

# In this case, you substitute "np" for numpy:

print(np.pi)
print(np.sqrt(9))  # or N.pi in the second case.
```

```
3.141592653589793
3.0
```

or the most lazy way:

In [3]:
```
from numpy import * # now all the functions are available directly, without the initial module r

print(pi)
print(sqrt(9.0))
print(sin(pi))
```

```
3.141592653589793
3.0
1.2246467991473532e-16
```

List of NumPy functions (not inclusive)

Basic functions

function	purpose
absolute(x)	absolute value
degrees(x)	Convert angles from radians to degrees
radians(x)	Convert angles from degrees to radians
deg2rad(x)	Convert angles from degrees to radians
rad2deg(x)	Convert angles from radians to degrees

Rounding functions

function	purpose
round(x)	Round an array to the given number of decimals
floor(y,x)	Return the floor of the input, element-wise
ceil(x)	Return the ceiling of the input, element-wise
trunc(x)	Return the truncated value of the input, element-wise

Trigonometric functions

function	purpose
arccos(x)	arccosine
arcsin(x)	arcsine
arctan(x)	arctangent
arctan2(y,x)	arctangent of y/x in correct quadrant
cos(x)	cosine
sin(x)	sine
tan(x)	tangent

Math functions

function	purpose
sum(x)	Sum of array elements over a given axis
prod(x)	Return the product of array elements over a given axis
trunc(x)	Return the truncated value of the input, element-wise
diff(x)	Calculate the n-th discrete difference along given axis
gradient(x)	Return the gradient of an N-dimensional array
cross(x,y)	Return the cross product of two (arrays of) vectors
trapz(x,y)	Integrate along the given axis using the composite trapezoidal rule
sqrt(x)	Return the positive square-root of an array, element-wise
square(x)	Return the element-wise square of the input
fabs(x)	Compute the absolute values element-wise
sign(x)	Returns an element-wise indication of the sign of a number
max(x)	Element-wise maximum of array elements
min(x)	Element-wise minimum of array elements

Exponential/Logarithm functions

function	purpose
exp(x)	Calculate the exponential of all elements in the input array
log(x)	Natural logarithm, element-wise
log2(x)	Base-2 logarithm of x
log10(x)	Return the base 10 logarithm of the input array, element-wise

Visualize 1-D NumPy Arrays using Matplotlib

Let's try create a new numpy array using the `.arange()` function

In [90]:
```
theta_in_degrees=np.arange(0,360,2) # list of angles from 0 to 359 at two degree intervals

theta_in_radians=np.deg2rad(theta_in_degrees) # convert to radians
sine_theta=np.sin(theta_in_radians) # calculate the sine values for all the thetas
sine_theta #output sine_thetas_in_degrees
```

Out[26]:
```
array([ 0.00000000e+00,  3.48994967e-02,  6.97564737e-02,  1.04528463e-01,
        1.39173101e-01,  1.73648178e-01,  2.07911691e-01,  2.41921896e-01,
        2.75637356e-01,  3.09016994e-01,  3.42020143e-01,  3.74606593e-01,
        4.06736643e-01,  4.38371147e-01,  4.69471563e-01,  5.00000000e-01,
        5.29919264e-01,  5.59192903e-01,  5.87785252e-01,  6.15661475e-01,
        6.42787610e-01,  6.69130606e-01,  6.94658370e-01,  7.19339800e-01,
        7.43144825e-01,  7.66044443e-01,  7.88010754e-01,  8.09016994e-01,
        8.29037573e-01,  8.48048096e-01,  8.66025404e-01,  8.82947593e-01,
        8.98794046e-01,  9.13545458e-01,  9.27183855e-01,  9.39692621e-01,
        9.51056516e-01,  9.61261696e-01,  9.70295726e-01,  9.78147601e-01,
        9.84807753e-01,  9.90268069e-01,  9.94521895e-01,  9.97564050e-01,
        9.99390827e-01,  1.00000000e+00,  9.99390827e-01,  9.97564050e-01,
        9.94521895e-01,  9.90268069e-01,  9.84807753e-01,  9.78147601e-01,
        9.70295726e-01,  9.61261696e-01,  9.51056516e-01,  9.39692621e-01,
        9.27183855e-01,  9.13545458e-01,  8.98794046e-01,  8.82947593e-01,
```

To get a sense of what your array looks like, use the following functions:

In [38]:
```
print(np.ndim(theta_in_degrees)) # the ndim() function gives you the dimension of your array

print(np.size(theta_in_degrees)) # the size() function gives you how many elements in total

print(np.shape(theta_in_degrees)) # the shape() function gives you the length of the array in ea
```

```
1
180
(180,)
```

Visualize 1-D NumPy Arrays using Matplotlib

When you've done calculation the sines of theta and type in sine_theta, here's what you've got:

```
In [26]: theta_in_degrees=np.arange(0,360,2) # list of angles from 0 to 359 at two degree intervals

         theta_in_radians=np.deg2rad(theta_in_degrees) # convert to radians
         sine_theta=np.sin(theta_in_radians) # calculate the sine values for all the thetas
         sine_theta #output sine_thetas_in_degrees

Out[26]: array([ 0.00000000e+00,  3.48994967e-02,  6.97564737e-02,  1.04528463e-01,
                 1.39173101e-01,  1.73648178e-01,  2.07911691e-01,  2.41921896e-01,
                 2.75637356e-01,  3.09016994e-01,  3.42020143e-01,  3.74606593e-01,
                 4.06736643e-01,  4.38371147e-01,  4.69471563e-01,  5.00000000e-01,
                 5.29919264e-01,  5.59192903e-01,  5.87785252e-01,  6.15661475e-01,
                 6.42787610e-01,  6.69130606e-01,  6.94658370e-01,  7.19339800e-01,
                 7.43144825e-01,  7.66044443e-01,  7.88010754e-01,  8.09016994e-01,
                 8.29037573e-01,  8.48048096e-01,  8.66025404e-01,  8.82947593e-01,
                 8.98794046e-01,  9.13545458e-01,  9.27183855e-01,  9.39692621e-01,
                 9.51056516e-01,  9.61261696e-01,  9.70295726e-01,  9.78147601e-01,
                 9.84807753e-01,  9.90268069e-01,  9.94521895e-01,  9.97564050e-01,
                 9.99390827e-01,  1.00000000e+00,  9.99390827e-01,  9.97564050e-01,
                 9.94521895e-01,  9.90268069e-01,  9.84807753e-01,  9.78147601e-01,
                 9.70295726e-01,  9.61261696e-01,  9.51056516e-01,  9.39692621e-01,
                 9.27183855e-01,  9.13545458e-01,  8.98794046e-01,  8.82947593e-01,
                 8.66025404e-01,  8.48048096e-01,  8.29037573e-01,  8.09016994e-01,
                 7.88010754e-01,  7.66044443e-01,  7.43144825e-01,  7.19339800e-01,
                 6.94658370e-01,  6.69130606e-01,  6.42787610e-01,  6.15661475e-01,
                 5.87785252e-01,  5.59192903e-01,  5.29919264e-01,  5.00000000e-01,
                 4.69471563e-01,  4.38371147e-01,  4.06736643e-01,  3.74606593e-01,
                 3.42020143e-01,  3.09016994e-01,  2.75637356e-01,  2.41921896e-01,
                 2.07911691e-01,  1.73648178e-01,  1.39173101e-01,  1.04528463e-01,
                 6.97564737e-02,  3.48994967e-02,  1.22464680e-16, -3.48994967e-02,
                -6.97564737e-02, -1.04528463e-01, -1.39173101e-01, -1.73648178e-01,
                -2.07911691e-01, -2.41921896e-01, -2.75637356e-01, -3.09016994e-01,
                -3.42020143e-01, -3.74606593e-01, -4.06736643e-01, -4.38371147e-01,
                -4.69471563e-01, -5.00000000e-01, -5.29919264e-01, -5.59192903e-01,
                -5.87785252e-01, -6.15661475e-01, -6.42787610e-01, -6.69130606e-01,
                -6.94658370e-01, -7.19339800e-01, -7.43144825e-01, -7.66044443e-01,
                -7.88010754e-01, -8.09016994e-01, -8.29037573e-01, -8.48048096e-01,
                -8.66025404e-01, -8.82947593e-01, -8.98794046e-01, -9.13545458e-01,
                -9.27183855e-01, -9.39692621e-01, -9.51056516e-01, -9.61261696e-01,
                -9.70295726e-01, -9.78147601e-01, -9.84807753e-01, -9.90268069e-01,
                -9.94521895e-01, -9.97564050e-01, -9.99390827e-01, -1.00000000e+00,
                -9.99390827e-01, -9.97564050e-01, -9.94521895e-01, -9.90268069e-01,
                -9.84807753e-01, -9.78147601e-01, -9.70295726e-01, -9.61261696e-01,
                -9.51056516e-01, -9.39692621e-01, -9.27183855e-01, -9.13545458e-01,
                -8.98794046e-01, -8.82947593e-01, -8.66025404e-01, -8.48048096e-01,
                -8.29037573e-01, -8.09016994e-01, -7.88010754e-01, -7.66044443e-01,
                -7.43144825e-01, -7.19339800e-01, -6.94658370e-01, -6.69130606e-01,
                -6.42787610e-01, -6.15661475e-01, -5.87785252e-01, -5.59192903e-01,
                -5.29919264e-01, -5.00000000e-01, -4.69471563e-01, -4.38371147e-01,
                -4.06736643e-01, -3.74606593e-01, -3.42020143e-01, -3.09016994e-01,
                -2.75637356e-01, -2.41921896e-01, -2.07911691e-01, -1.73648178e-01,
                -1.39173101e-01, -1.04528463e-01, -6.97564737e-02, -3.48994967e-02])
```

So you've got the job done, but what's the problem here?

Nobody (at least not me..) has any idea about what the numbers mean - human eyes are just not sensitive to such a list of numbers - it's a headache

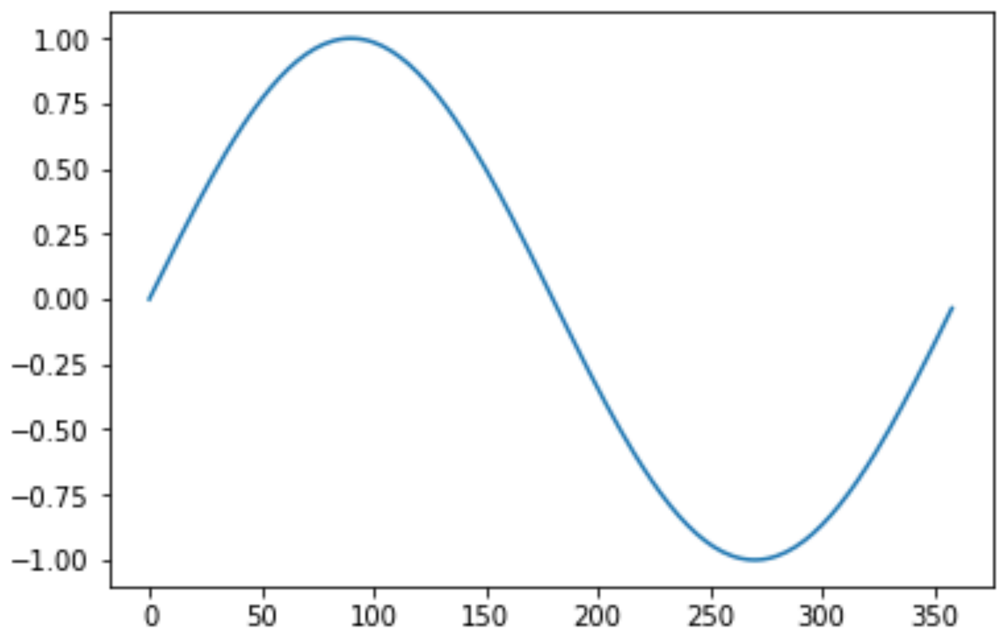
Here's when visualisation comes in handy. Let's try make some "plots" using another Python module called Matplotlib

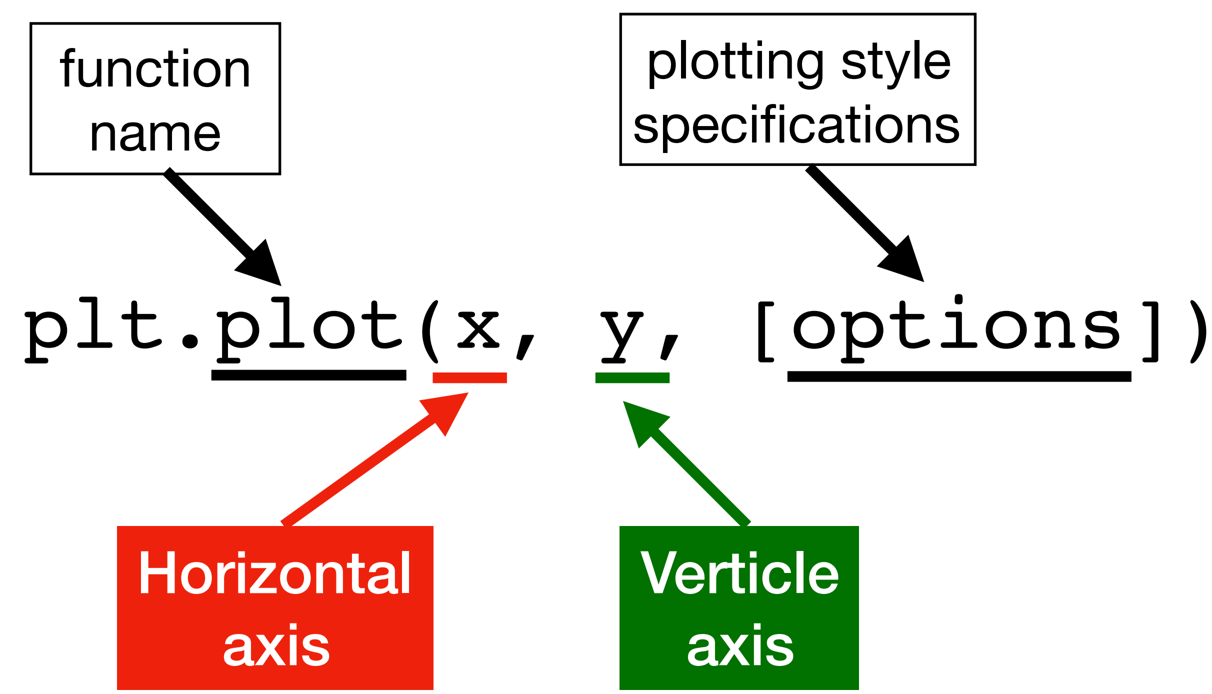
Visualize 1-D NumPy Arrays using Matplotlib

The easiest way to do this is using the package **matplotlib** which has many plotting functions, among them a whole module called **pyplot**. We **import** the **matplotlib.pyplot** module as **plt**.

In [91]:
```
import matplotlib.pyplot as plt

plt.plot(thetas_in_degrees,sines); # plot the sines with the angles
plt.show() # if you don't have this .show() function, Python will not bother showing the plot!
```





Import **pyplot** from **matplotlib**

```
import matplotlib.pyplot as plt # import the pyplot sub-module from the matplotlib module
```

Use the **plot()** function from **plt** to make a line plot

```
plt.plot(thetas_in_degrees,sines); # plot the sines with the angles
```

Show the plot you just made using the **show()** function from **plt**

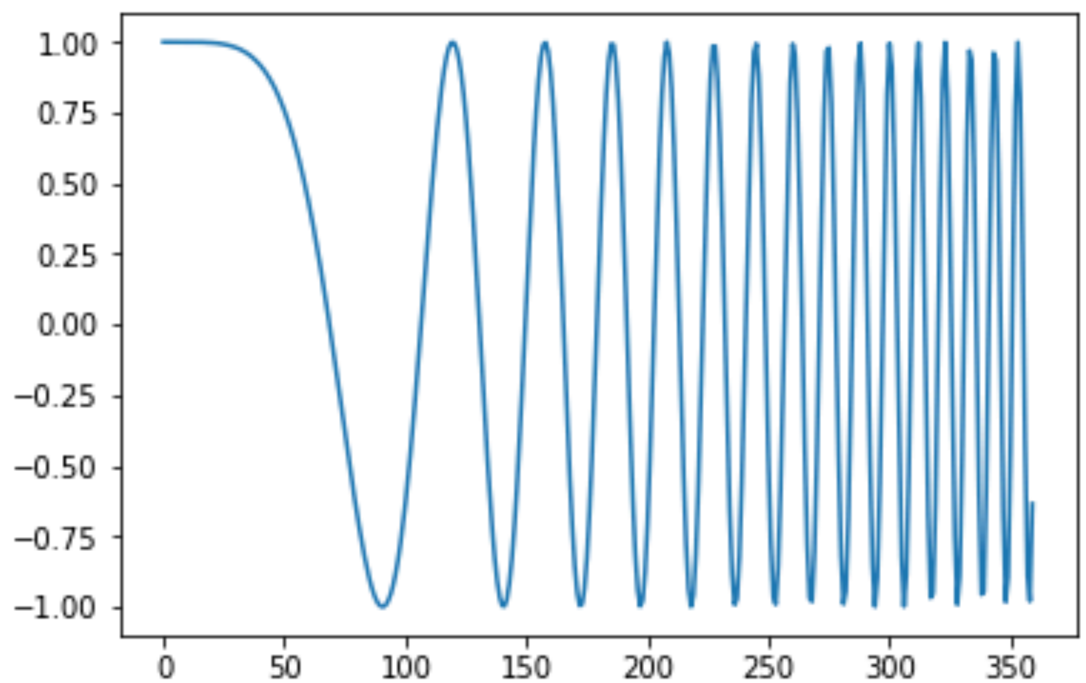
```
plt.show() # if you don't have this .show() function, Python will not bother showing the plot!
```

Visualize 1-D NumPy Arrays using Matplotlib

You can also put numpy array operations within a plot() function, for example:

In [107]:
```
import numpy as np # import the numpy module ad name it as np
import matplotlib.pyplot as plt # import the pyplot sub-module from the matplotlib module

theta = np.arange(0,360,1) # here we used two numpy functions within one line of code: arange()
plt.plot(theta,np.cos(np.deg2rad(theta)**2.5))       # use the plot function from plt to make th
plt.show()                            # show results, make sure you type this everything when a
```



What happened in the above code is that Python will first calculate the results of cos(theta**2.5), and then plot the results as a function of theta using the plot() command

That's all! Let's practise!