PythonWarmup

AAB

January 31, 2022

Image Processing

Python Warmup

0. Introductory Remarks

This is for *all* students. Test/refresh your *Python* familiarity with the following relatively basic exercises. If you have problems with any one of these, please ask a demonstrator for assistance. If the first problems are *way* too easy, you don't need to attend the rest of the first session. But if they are not, you may want to go through *An Informal Introduction to Python*. This exercise sheet has been recently rewritten to move from *Matlab* to *Python* and tailored to meet the needs of the IP course, so feedback is welcome: any 'howlers' that you find should be pointed out.

1. Python Lists vs Numpy Arrays

If you have experience with *Matlab* or programming languages such as C/C++, you will be familiar with the idea of arrays. Arrays are variables which contain multiple items of the same type that can be addressed by specifying a location relative to the start (or end, in some languages) location in active memory.

A source of confusion may be that *Python* has similar stuctures to arrays, but by design, and widely used as its *primary* equivalent data structure, is the `list`, which is not quite the same as an array.

For those familiar with languages such as *Matlab*, lists in *Python* do not recognise the application of something like an arithmetic operation to all elements: if variable `a` holds a list, something like:

a = [1, 2, 3]

then the command:

b = a + 3

is badly specified, as far as *Python* goes (indeed, it is treated as a concatenation operation). Try the two commands above in the cell below:

[1]:
```
a = [1,2,3]
b = a + 3 # This will give you an error! Make sure you *understand* the error␣
 ↪msg
print(b)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-cae026ca8c07> in <module>
      1 a = [1,2,3]
----> 2 b = a + 3 # This will give you an error! Make sure you *understand* the
 ↪error msg
      3 print(b)

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

On the other hand, if we use the very useful *numpy* module, we find that the operations defined above work very nicely indeed.

[2]:
```
import numpy # This loads the numpy module, which contains math-like and vector
 ↪like definitions

a = numpy.asarray([1,2,3], dtype=int) # Creates a numpy array, with elements of
 ↪type integer
b = a + 3
print(b)
```

```
[4 5 6]
```

You will note that *Python* now "knows" how to perform the scalar addition on each element of array `a`, so we are (so far) doing something that is legal, which is adding a scalar to a vector. Now, we can try a few more useful operations.

Exercise 1 Write *Python* code to create an *array* of numbers containing the even integers between 31 and 75. *Hint* you can either write this out manually, or (ideally) use a loop to do this in *Python*. Try both!

To do this using a loop, you need the following pieces of information:

1. You can create an empty list using something like `a = [ ]`; you might try to guess how to turn that into an array!
2. You can use the command `numpy.append()`.
3. To construct a loop, you may use the construct as shown below:

   ```
   for n in range(start, stop, step):
       _do something in my loop_
   ```

   where the *do something...* line should be indented by a Tab character. ■

Note: The indentation of lines with control statements is **not** optional, as it is in other langages. In *C*, *Matlab*, *Fortran* etc, you can get away with (this is *pseudocode*, and does not correspond to any particular language syntax):

```
for i = start _to_ finish {
```

`print('hello')` # This should be indented (i.e. shifted to the right)

`}`

You can't get away with this in *Python*: instead of having something indicating the beggining and end of the inner part of the loop, the indentation itself plays this role! So everything that is indented will be repeated according to the loop control conditions. Note also the presence of the colon symbol `:`, which is also a requirement.

A similar arrangement exists for conditional statements.

Python has the in-built `range` class which returns an object that is of type `sequence`; a `sequence` is similar to the more commonly known `list` and `tuple`. We will not explore this too much in this Notebook, even though it is very widely used in controlling loops. You should note that the sequence produced by the syntax `range(start, stop, step)` or the one we have used above `range(stop)` includes `start` but not `stop`.

One Possible Solution to Exercise 1.

```
import numpy
a = numpy.asarray([],dtype=int) # makes an empty array
for n in range(32,75,2):    # Loop "control" statement
    a = numpy.append(a, n)  # Inside of loop

print(a)
```

Finishing up on this

1. It is common to use the following sort of convention for importing the *numpy* library:

   ```
   import numpy as np
   ```

   and then the commands become instead something like this (important: compare with earlier use of the `numpy.append` command):

   ```
   a = np.asarray([1,2,3], dtype = int)

   b = np.append.... etc
   ```

2. Apart from float, you can also have `dtype = float, uint8, double` etc. If you do not (and cannot guess) what the difference is between these types (`float`,`double`), please speak with a GTA for assistance.

3. We can address specific elements of the array `a` defined above as follows:

   ```
   print(a[0])  # This is the first element

   print(a[2])  # The *third* element in the array
   ```

Interlude 1 - Some useful commands

It turns out that figuring out the data type of a variable - either one you create or one that is returned by some in-built or library function - is very handy (that's an understatement!). Amongst the most useful of these is `len` which allows us to examine the number of elements within an array or list and the command `type` to find the class of a particular variable or object.

Thus, if we had the following:

```
import numpy as np
a1 = [1,2,3]
print(type(a))
a2 = np.asarray(a, dtype=int)
print(type(a1))
```

we would be able to tell the difference between `a1` and `a2`.

Another *super* useful numpy command is `np.shape()`. So trying

```
np.shape(a2)
```

gives useful information about the shape of `a2`. Try this now, and note the answer (*really* note it).

It is also useful (assuming that code is being well maintained by the author) to use `help()`. Thus (for example) `help (np.shape)` will give you the documentation about that particular command. You can often find this information on the official documentation page for `numpy`; similar help commands will generally work for good quality *Python* libraries.

Exercise 2 Let x = [1, 7, 2, 3].

Create this as 1 x 4 numpy array in the *Python* workspace.

a. Add 15 to each element

b. Add 2 to just the even-indexed elements (Reminder: we start indexing at `0` in *Python*)!

c. Compute the square root of each element

d. Compute the cube of each element

In each case, consider they *type* of numpy array representation that would be most efficient, yet also obtain correct results. Note that the type of the *result* of some operations above may be automatically determined by `numpy`, and may be different from the input to the calculation. If you want to specify a greater range of possible representations, a list of these can be obtained here. Note that some installations (it is rather bizarre) may require that `dtype` be specified as `np.unit8`, or `np.uint16` rather than simply `uint8` or `uint16`.■

Click here for possible solutions

First: `x = np.asarray([1, 7, 2, 3], dtype=int)` Note: you could also use np.uint8 for this case; but if we had to do arithmetic on this array that meant having numbers greater than 255, or less than zero, we would have to use a *signed* representation rather than unsigned, and a floating point representation if we wanted to include fractions.

a. `x = x+15`

b. You could use a loop to do this; later, you will see how to do it using indexing syntax. So, with a loop:

```
for i in range(0,4,2):
    x[i] = x[i] + 2
```

The *Pythonic* way of doing this (which we will look at later) is:

```
x[0::2] = x[0::2] + 2
```

c. I would recommend casting values to float before doing pretty much any non-trivial mathematical operation. In that case, you would do:

```
np.sqrt(np.asarray(x, dtype=float))
```

but this not *necessary* to get the right answer with `numpy`. You can just do:

```
np.sqrt(x)
```

but we would urge caution in doing calculations without converting arrays into floating point representations.

d. `x**3`

2. Numpy arrays as vectors

We can do certain types of operations between `numpy` arrays. We will find that several fairly obvious operations are understood, just as they would be between two vector operations.

There are some very striking differences with the assumptions made by *Matlab* however, which means that operators such as $*$ and $.*$ in *Matlab* do not necessarily translate in the same way to numpy arrays in *Python*. Specifically, whilst the $*$ operation in *Matlab* **always** obeys the rules of matrix and matrix/vector algebra, the same is not the case for *Python*, which in contrast assumes element-wise operations to so-called binary arithmetic operators (i.e. those that take two arguments).

If we want similar behaviour in *Python* – treating arrays of numbers as vectors – we meet our first slight complexity; *Python*, for example, does not understand the notion of "transpose" *unless* you insist that these are really vectors rather than arrays. The easiest way to do this is to treat the these arrays as *single column 2D arrays* like so:

```
x  =  [[1],
       [7],
       [2]
       [3]]
```

You will find that if you now print this, and look at its shape, it is rather different to what we had at the end of Exercise 2:

[3]:
```
import numpy as np
x  =  np.asarray([[1],[7],[2],[3]], dtype=float)
print('Shape of x is:', np.shape(x))
x = np.transpose(x)
print('Shape of x is now:', np.shape(x))
```

```
Shape of x is: (4, 1)
Shape of x is now: (1, 4)
```

Quiz Can you guess how you would create a 1x4 array?

 Stuck on the Quiz? Click here to see a possible solution

```
x = np.asarray([[1, 7, 2, 3]], dtype=float)
```

Note the use of the nested `[ ]` operators!

Exercise 3 Let

$$\mathbf{x} = [3\,2\,6\,8]^T$$

and

$$\mathbf{y} = [4\,1\,3\,5]^T$$

Create both of these as 4 x 1 "vectors" in the *Python* workspace, and ensure that their type is `float`.

a. Add each element in **x** to each element of **y**; assign the result to vector **z**

b. Raise each element of **x** to the power specified by the corresponding element in **y**. This uses the *Python* exponentiation ** operator. Verify the results using a calculator.

c. Divide each element of **y** by the corresponding element in **x**. Check the result.

d. Multiply each element in **x** by the corresponding element in **y**; assign this to **z**

e. Add up the elements in **z** from Task 3.d and assign the result to a variable w; this requires the `np.sum()` function. What is the value of w?

f. Compute

$$\mathbf{x}^T\mathbf{y} - w$$

and assign it to the variable `result`. Does the calculation make sense to you according to the rules of matrix/vector algebra? *Hint*: You need to think carefully about the product between a row and column vector. We will talk about this in class/

In each case, consider they *type* of numpy array representation you need to be most efficient in memory usage, yet also obtains correct results in accordance with your thinking, treating the operations as you would when doing everyday (floating point!) arithmetic. If you do not understand what this means, please speak with a UTA or GTA. ■

Click here to see the solutions for Ex 3.

```
x = np.asarray([3, 2, 6, 8], dtype=float)
```

```
y = np.asarray([4, 1, 3, 5], dtype=float)
```

a.

```
z = x+y
```

b.

```
x**y
```

c.

```
x/y
```

d.

```
z = x*y
```

e.

```
w = np.sum(z) print(w)
```

f. `result = np.sum(x*y) - w`

```
print(result)
```

But, see "Alternative" solution, below, and the relation to matrix-vector multiplications.

Alternative solution to Ex. 3 f.

```
import numpy as np # not needed if already imported!
x = np.asarray([[3],[2],[6],[8]], dtype=float)
y = np.asarray([[4],[1],[3],[5]], dtype=float)
print('Note: both x and y are of dimensions:', np.shape(x))
result=np.dot(np.transpose(x),y)-np.sum(x*y)
print('Result is:',result)
print('Shape of result is:',np.shape(result))
```

This little example provided in Exercise 3f is quite a useful reminder about vector algebra. **x** and **y** start lives as column vectors; a transposed column vector is a row vector; multipliying this by a column vector gives the dot product between the two vectors. You will recall that multiplying a $1 \times N$ row vector by a $N \times 1$ column vector "contracts" the vectors to a scalar. It is also known as the linear projection of the first vector against the second (or vice versa).

The mechanics of the dot product are that we take the *sum* of the *products* of the corresponding elements of each vector. Hence, the result of 0 in the final evaluation of Ex 3.6. You should make sure you understand this result, by way of reminding yourself about vector/vector multiply operations according to the rules of matrix/vector algebra.

But you should also note that the result has a size of (1,1), which is weird. Indeed, if you do:

[4]:
```
# Type the code below only if you have "result" defined from 3f!
# If you get that charming, welcome pink regurgitation
# of text below), you have not defined the variable "result".....
result=np.squeeze(result) # gets rid of extraneous "dimensions" in the variable
print('Result is:',result)
print('Shape of result is:',np.shape(result))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-6a3f42909457> in <module>
      2 # If you get that charming, welcome pink regurgitation
      3 # of text below), you have not defined the variable "result"…
```

```
----> 4 result=np.squeeze(result) # gets rid of extraneous "dimensions" in the␣
 ↪variable
      5 print('Result is:',result)
      6 print('Shape of result is:',np.shape(result))

NameError: name 'result' is not defined
```

…you will see that `result` is now a single (scalar) value. So rather than having this value stored in something like a 2D array (array of pointers to address locations), which is what the `[[]]` notation implies, we now have `result` as simply a pointer to the address of where the scalar is to be found.

Exercise 4 Evaluate the following *Python* expressions by hand and use a calculator to check the answers:

a. `2 / 2 * 3`

b. `6 - 2 / 5 + 7 ** 2 - 1`

c. `3 ** 2 / 4`

d. `3 ** 2 ** 2`

e. `2 + np.round(6 / 9 + 3 * 2) / 2 - 3`

f. `2 + np.floor(6 / 9 + 3 * 2) / 2 - 3`

g. `2 + np.ceil(6 / 9 + 3 * 2) / 2 - 3`

If there are any results you do not understand, please contact a demonstrator. ■

3. Control Statements

So, we have seen that indented statements and *colons* can be used to indicate loops. We use similar sorts of things for other types of control statements. Here is an example of using a simple (and pretty useless!) `if` statement:

```
for x in range(10):
    if x == 5:
        print('x is 5')
```

Note the indentation after the `if` statement. We can also have an `if ... else...` construct:

```
for x in range(10):
    if x == 5:
        print('x is 5')

    else:
        print('x is not 5...')
```

We can have conditional beyond the first `if`; in some languages, the keywords *else if* are used; these are contracted in *Python* to `elif`:

```
for x in range(10):
    if x == 5:
        print('x is 5')

    elif x==4:
        print('x is 4...')

    else:
        print('x is neither 4 nor 5...')
```

Above, we make use of the equality comparison ==. The full list of comparison operators (*less than*, *greater than* etc) can be found here. If you need help with this section, please consult a teaching assistant; note that the best way to understand how to use these is by coding some things up yourself, to make sure you understand the use of the operators: very similar symbols are used to other programming languages, but one key difference (with, say, *C* programming) is the *precedence* of comparisons with respect to arithmetic performed on one either or both of the arguments.

4. List Comprehension

Exercise 5 Create `numpy` arrays with the elements given by (treat each of (a)-(d) separately). It is suggested that you do these first using loops (similar to what we did in Exercise 1), as a way of remembering how to do loops, and the syntax of using the `range()` function.

a. `2, 4, 6, 8, ..., 20`

b. `10, 8, 6, 4, 2, 0, -2, -4`

c. `1, 1/2, 1/3, 1/4, 1/5, ..., 1/20`

d. `0, 1/2, 2/3, 3/4, 4/5, ... , 19/20`

Once completed using loops (just to remind ourselves of the syntax), we will also look at doing these using *list comprehension*, which is quite a common contruct in the *Python* language. ■

Click here for some solutions

a)

```
x = np.asarray([],dtype=int)

for i in range(2,22,2):
    x = np.append(x,i)
```

b)

```
x = np.asarray([],dtype=int)

for i in range(10,-6,-2):
```

```
    x = np.append(x,i)
```

c)

```
x = np.asarray([],dtype=int)

for i in range(1,21):
    x = np.append(x,1/i)
```

d)

```
x = np.asarray([],dtype=int)

for i in range(0,20):
    x = np.append(x,i/(i+1))
```

List comprehension is a very nifty way of building lists, which can then be turned into arrays, conveniently. For example, if we type the command:

```
[m for m in range(5)]
```

(which looks a little clumsy), we will get a list containing 5 elements, starting from 0 to 4 in steps of 1. We can even add some conditions to this, so that we can eliminate a particular value in this list, if we wish with a condition:

```
[m for m in range(5) if m>0]
```

Furthermore, the first occurrence of `m` in the line above can be replaced with pretty much any simple expression involving `m`. Given this, how would you solve **Ex 5-a.**?

 Solution to Ex. 5 a using LC

```
x = np.asarray([2*(n+1) for n in range(10)], dtype=int)
```

So…. did you remember to convert the list to a numpy array :-)?

Once you have got this done, work on the solutions for **5-b** to **5-d** using list comprehension as well. If you get stuck, please speak with a demonstrator rather than just trying to get hold of the answers; in this way, we can learn what pieces you find difficult, and can then proceed accordingly.

5. Dictionaries

Dictionaries are a *Python* structure similar to associative memory. An example will explain how this works sufficiently well:

[5]:
```
x = {'Key-1': 'Chocolate',
     'Key-2': 'Vanilla'}
```

[6]:
```
MyFavourite = x['Key-1']
print(MyFavourite)
```

```
Chocolate
```

[7]:
```
YourFavourite = x['Key-2']
print(YourFavourite)
```

```
Vanilla
```

The *Keys* in this case are the strings `'Key_1'` and `'Key_2'`, and the *Values* (in this case) are the strings 'Vanilla' and 'Chocolate'. The values can also be numbers, arrays or other *Python* objects:

[8]:
```
import numpy as np

x = {'Key-1': np.asarray([1,2,3,], dtype=int),
     'Key-2': np.asarray([1.0/2.0, 3.0/4.0], dtype=float)}
```

[9]:
```
x['Key-2']
```

[9]:
```
array([0.5 , 0.75])
```

Arrays of dictionaries can also be defined, using a syntax like this:

[10]:
```
myList = [
    {
        'apples':12,
        'pears':14
    },
    {
        'almonds':52,
        'cashews':641
    },
    {
        'lettuce':6,
        'tomatoes':84
    }
]
```

Then, we can use address items on the list using the order, but also need to keep track of the possible dictionary elements of each element.

[11]:
```
myList[1]['almonds']
```

[11]:
```
52
```

but:

[12]:
```
myList[0]['almonds'] # Won't work
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
```

```
<ipython-input-1-6d4ee5a9cb72> in <module>
----> 1 myList[0]['almonds'] # Won't work

KeyError: 'almonds'
```

Luckily, we can find out the *Keys* for each entry using something like this:

[13]:
```
myList[0].keys() #WOW!!!
```

[13]:
```
dict_keys(['apples', 'pears'])
```

Exercise 6 Using the `for a in b` construct, print a little table (just using `print()` commands) that presents each item on the list, and the quantities of each from `myList`. ■

 Solution to Ex. 6.

```
for item in myList:
    for category in item.keys():
        print('Entry:',category,'Quantity:',item[category])
```

Question Who on earth counts individual nuts?!?!?

A well written solution to Exercise 6 should convince you that, actually, good *Python* code can be *very* easy to read, and I would argue that this is one the attractions of the language. But it is also true that good code in general can be easy to read; what *Python* gives us is some language constructs that support a very readable style of writing that may be associated with being *Pythonic*. Opinion: being *too* passionate about style borders on being moronic….

6. Tuples

Tuples are collections of data entities; they are what is called *immutable*, so the entries can't be changed, and do not have any particular order. They look a bit crazy in definition:

[14]:
```
mytuple = 'apples', 'pears', [1,2,3,4]
```

But can be addressed in a familiar way:

[15]:
```
mytuple[1] # This addresses the second element of the tuple
```

[15]:
```
'pears'
```

[16]:
```
len(mytuple) # So we can figure out how many elements there are....
```

[16]:
```
3
```

[17]:
```
# And below, a useful way of looping over the entries
for entry in mytuple:
    print(entry)
```

```
apples
pears
[1, 2, 3, 4]
```

[18]:
```
# An alternative way of grouping tuples - easier to read?
mytuple = ('apples', 'pears', [1,2,3,4])
```

[19]:
```
len(mytuple)
```

[19]: 3

[20]:
```
for entry in mytuple:
    print(entry)
```

```
apples
pears
[1, 2, 3, 4]
```

Tuples, when used as arguments to a function, can also be automatically unpacked (very confusing, if you ask me) by adding a * before the tuple when calling the function. To see the difference, contrast the following two expressions:

[21]:
```
print(mytuple)
```

```
('apples', 'pears', [1, 2, 3, 4])
```

and...

[22]:
```
print(*mytuple)
```

```
apples pears [1, 2, 3, 4]
```

...but note that you can't simply use `*tuple` on its own: it needs to be "decently clothed" in a function call!

7. Sets

Yep, we've got yet another data type! Sets are often used to hold collections of objects, such as strings, and are designed to support set-based operators, such as intersections and unions.

So, we can have an (incomplete) set of mythical creatures:

[23]:
```
mythical_creatures = {'Unicorn', 'Centaur', 'Dragon', 'Mermaid'}
```

Sets are, however, not subscriptable, so you can't do something like `mythical_creatures[2]`.

But you can iterate over the items of a set like you can can with tuples:

[24]:
```
for creature in mythical_creatures:
    print(creature)
```

```
Centaur
Mermaid
Dragon
Unicorn
```

Personally, I tend to use sets to hold strings of characters that I might be looking for in a text document or the header of a file (more on this in later practicals).

Interlude 2

Now is the time to have a more complete look at all of these constructs, and to get used to reading **proper** technical documentation on languages. Check out the official Python documentation on data structures. This covers pretty much what we have presented here, but goes a bit further into the operations and restrictions of the data structures supported by Python. *Learning to read documentation is a skill that you have to develop*, and so further elaboration of some of the language contstructs of *Python*, you will be referred to the reference material, which is usually best accessed online (not to mention free!).

Next, we will move onto a more detailed look at working with `numpy` arrays, which is vital for image processing and many areas of data science and machine learning.

8. More on Numpy Arrays

We looked at the idea of mapping a list to a numpy array which supported opperations. We also looked at how to create an array that supported the transpose operation - you basically need to have a 2D array structure (an array of 1D arrays, if you like), which can be created from lists.

Starting from our *row* and *column* vector examples back in Section 2, we can now look at building a 2D array like so:

[25]:
```
import numpy as np

A = np.asarray([[1,2,3],[0,0,0],[-1,-2,-3]],dtype=float)
```

[26]:
```
print(A)
```

```
[[ 1.  2.  3.]
 [ 0.  0.  0.]
 [-1. -2. -3.]]
```

A strong recommendation is to make use of the `np.shape()` command to verify the size and dimensions of the 2D array. This will return a `tuple` containing elements that describe the shape of the argument (which should be of type `numpy array`:

[27]:
```
theShape = np.shape(A)
```

[28]:
```
# Typing the theShape gives us (echoes) the contents of
# this tuple:
theShape
```

[28]:
```
(3, 3)
```

And, yes, this is a tuple:

[29]:
```
# `type` (see Interlude 1) tells us this is a tuple
type(theShape)
```

[29]:
```
tuple
```

Thus, we can address its elements, like `theShape[0]` and `theShape[1]`

`numpy` also gives us some standard ways of creating fairly commonly used 2D arrays, by passing in an argument of the desired shape:

[30]:
```
a = np.zeros((3,3))
a
```

[30]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[31]:
```
a = np.ones((3,3))
a # 'echos' the value of a
```

[31]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Rather annoyingly, the command to create an identity matrix forces one to be explicit about this being square, so passing in a tuple does not work; we simply pass in the dimension of the square identity matrix:

[32]:
```
a = np.eye(2)
a
```

[32]:
```
array([[1., 0.],
       [0., 1.]])
```

Or, more excitingly:

[33]:
```
a = np.eye(7)
a
```

[33]:
```
array([[1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
```

```
       [0., 0., 0., 0., 0., 0., 1.]])
```

8.1 3D Arrays We will need to make use of 3D arrays in image processing. So, we start with the most logical way of defining these, using the `[ ]` construction:

[34]:
```
# Define a 3D array of type int
a = np.asarray([[[1,2],[3,4]],[[4,3],[2,1]]], dtype=int)

# Confirm this is indeed a 3D array
np.shape(a)
```

[34]:
```
(2, 2, 2)
```

There are some methods to help us create predefined 'standard' 3D (even ND) arrays:

`a = np.zeros((3,3,3))`

Confirm this is indeed a 3D array:

`np.shape(a)`

Finally, here is a new one for you; we can generate uniformly distributed random numbers shaped into *N*-D using the following (this is 4-D):

[35]:
```
a = np.random.rand(3,3,3,3)
np.shape(a)
```

[35]:
```
(3, 3, 3, 3)
```

Exercise 7 Examine the entries of `a` from the immediately preceeding command. If you have any questions about the entries, ask a UTA/GTA. ■

Note You can see a *major* inconsistency between the arguments taken by `np.random.rand()` and `np.zeros()`. The former expects to take N arguments for N-dimensional arrays, and the latter takes a single `tuple` argument with the tuple containing N entries. This is one of those quirks that comes about from the way that *Python* and its popular libraries have been developed from open-source contributions. Changing libraries such as this to be consistent would break many other existing libraries and code, so I would guess this is here to stay…

8.2 Array Slicing Understanding *numpy* array slicing is vitally important for image processing. We'll start by defining a `3x3` numpy array of floating point numbers:

[36]:
```
X = np.asarray([[1,2,3],[4,5,6],[7,8,9]], dtype=float)
```

Here is our first example of slicing:

[37]:
```
a = X[0,:]
a
```

[37]:
```
array([1., 2., 3.])
```

..and here is an exmaple of slicing in the other direction:

[38]:
```
b = X[:,0]
b
```

[38]:
```
array([1., 4., 7.])
```

In contrast to *Matlab*, which would preserve the shape of the slice, making a distinction between whether slicing a 2D array produces a row or a column vector, the slicing operation here produces only an array. You should make sure you understand this byr referring to the results of `np.shape()` in the part of this Notebook where we first met arrays and also where we looked at the difference between row and column vectors:

[39]:
```
print('Slicing one row:', np.shape(a))
print('Slicing one column:', np.shape(b))
```

```
Slicing one row: (3,)
Slicing one column: (3,)
```

We can also extract sub-arrays, like this:

[40]:
```
Y = X[0:2,0:2] # Note that column/row 2 are *not* included!
Y
```

[40]:
```
array([[1., 2.],
       [4., 5.]])
```

Contrast this with

[41]:
```
Y = X[1:3,1:3] # There *is* no column or row 3!
Y
```

[41]:
```
array([[5., 6.],
       [8., 9.]])
```

This might be slightly confusing, but - like the `range` object we met earlier, the indexing does not run up to the end point (`b`) in the slicing syntax `a:b`.

Quiz Given what we have just seen, can you now see how you might extract row and column vectors from a 2D array in a manner which preserves the row/column "orientation" that that extracting either a row or column of a matrix (think matrix algebra!) might suggest?

Stuck on the Quiz? Click here to see one solution

`Y = X[:,0:1]`

extracts the first column of `X` as a column vector, and

`Y = X[0:1,:]`

extracts the first row of `X` as a row vector.

[42]:
```
Y = X[0:3:2,0:3:2] # What's happening here?
Y
```

[42]:
```
array([[1., 3.],
       [7., 9.]])
```

Rather confusingly, we can index going backwards:

[43]:
```
Y = X[:-1,:]
Y
```

[43]:
```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

[44]:
```
Y = X[:-2,:]
Y
```

[44]:
```
array([[1., 2., 3.]])
```

[45]:
```
Y = X[:, 1:-1]
Y
```

[45]:
```
array([[2.],
       [5.],
       [8.]])
```

Now, check this out:

[46]:
```
Y = X[:, 3:0:-1]
Y
```

[46]:
```
array([[3., 2.],
       [6., 5.],
       [9., 8.]])
```

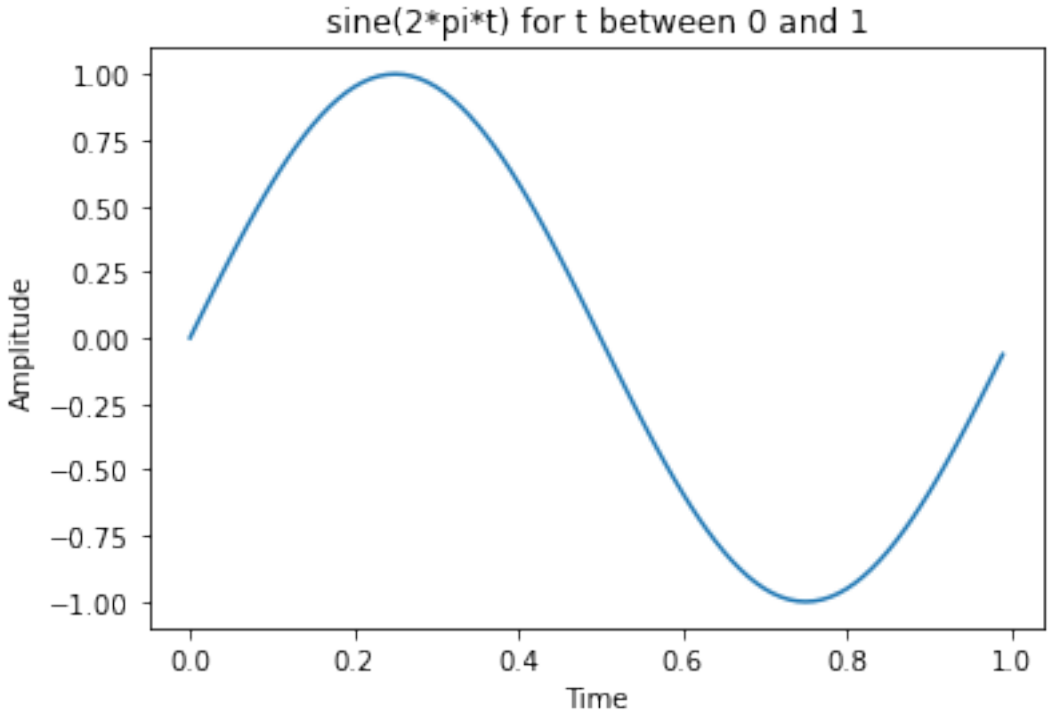
What happened here? If you can't figure it out, speak with a UTA/GTA. To test your understanding, try a few more examples that you construct yourself….

8.3 *numpy* array and vector operations *numpy* also provides useful mathematical operations that are "vectorised", as in *Matlab*. By "vectorised" we mean that a mathematical operation such as taking the sine (`np.sin()`) or cosine (`np.cosine()`) of *all* values in a numpy array can be performed without *explicitly* writing loops. Here is an obvious example of what this means:

[47]:
```
import numpy as np
import matplotlib.pyplot as plt

NSamples = 100
dt = 0.01
```

```
t = np.zeros(NSamples, dtype=float)
y = np.zeros(NSamples, dtype=float)

# Note the loop
for i in range(0,NSamples):
    t[i] = float(i)*dt
    y[i] = np.sin(2*np.pi*t[i])

# Plotting functions, which are always handy
# to scientists and engineers!
plt.plot(t,y)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('sine(2*pi*t) for t between 0 and 1')
```

[47]: Text(0.5, 1.0, 'sine(2*pi*t) for t between 0 and 1')



First, note the existence of `np.pi` which provides an approximation to the constant π.

Now, let's look at one step of vectorization. Let's say that we have the array `t` already defined. We can simply write the following:

```
# Assuming t is already defined as above, as a 100-element numpy array with values
# increasing in constant steps of dt:
```

```
y = np.sin(2*np.pi*t) # This can be done *once* outside of the loop
```

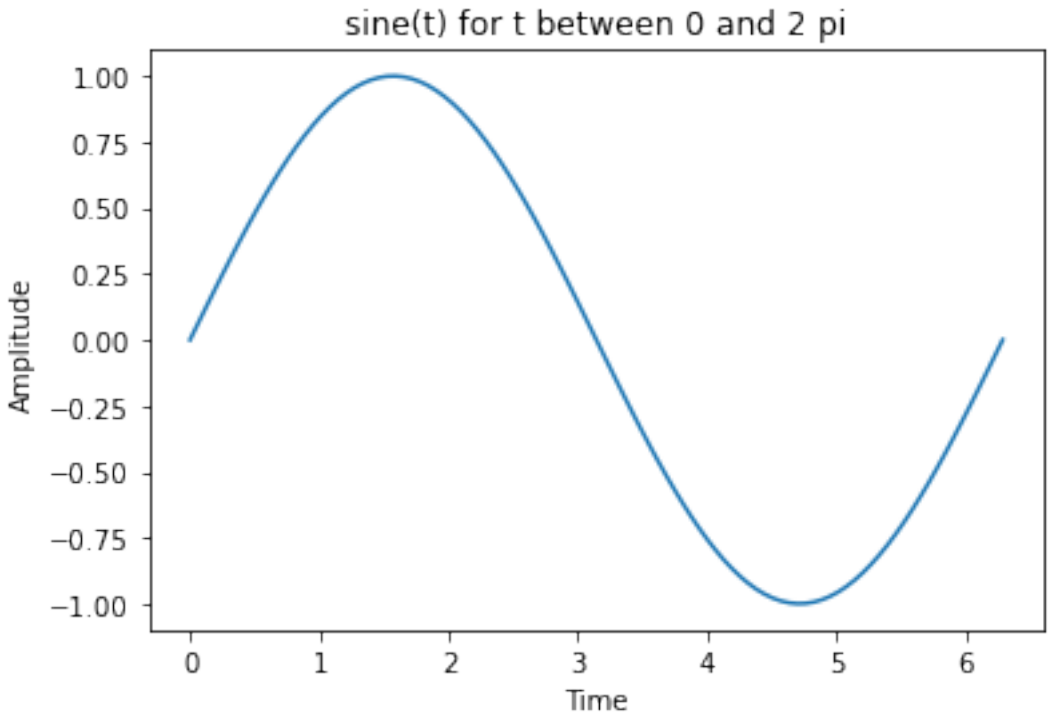
But this seems to still be a pain, as we seem to still need a loop to define the variable `t`. But it turns out that there is a helper function provided for things like this:

```
t = np.linspace(0,1,100) # This can be done *once*
```

So, the complete code would be this:

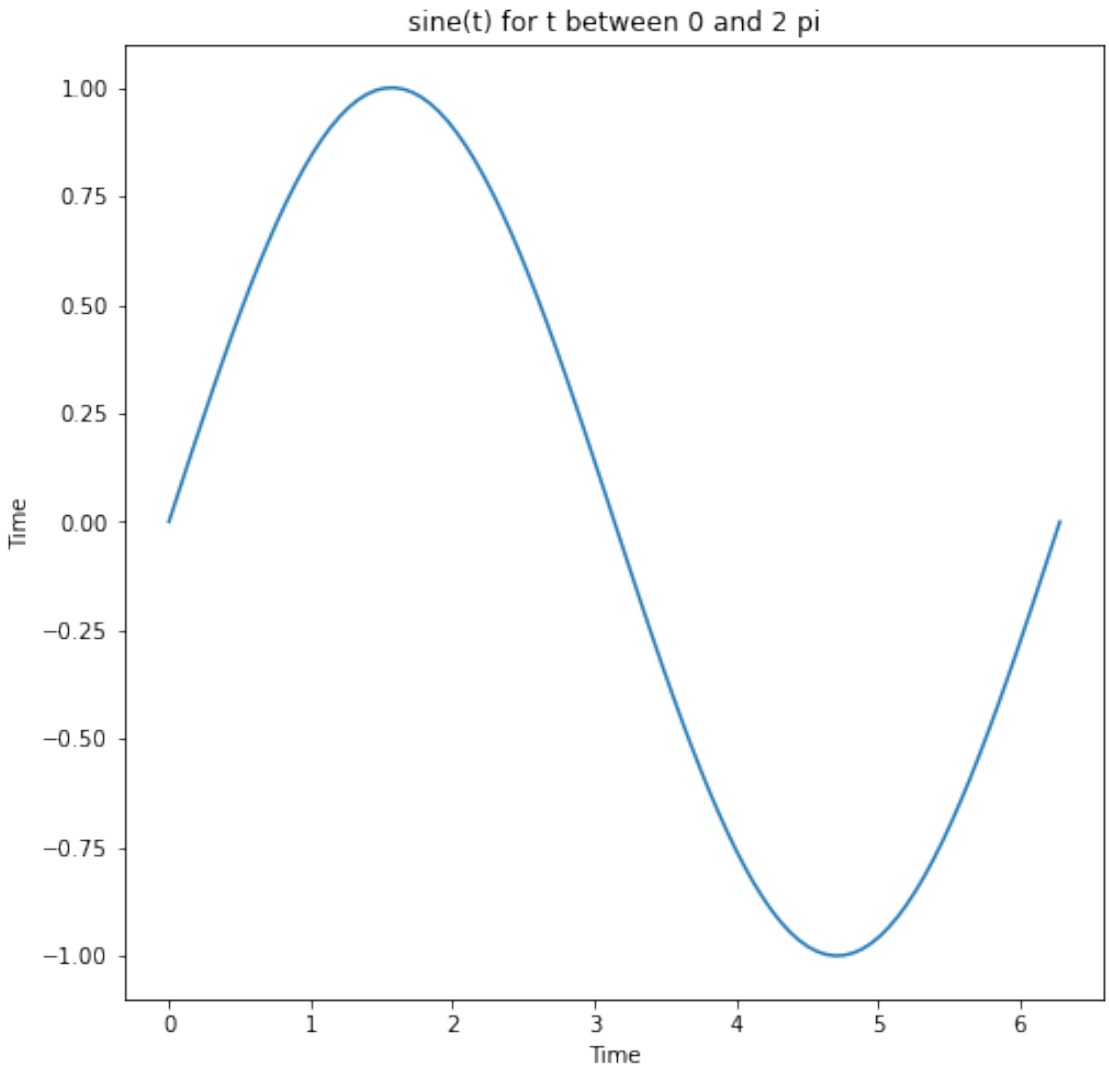
[48]:
```
import numpy as np
import matplotlib.pyplot as plt

NSamples = 100

# Note that this is slightly more transparent, but it is
# a matter of convenience/elegance/choice over where to
# include the 2 pi factor
t = np.linspace(0, 2*np.pi, NSamples)  # Look, Ma...
y = np.sin(t)                          # ...no explicit loops!

# Plotting functions, which are always handy
# to scientists and engineers!
plt.plot(t,y)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('sine(t) for t between 0 and 2 pi') # Agree with title :-)?
```

[48]:
```
Text(0.5, 1.0, 'sine(t) for t between 0 and 2 pi')
```



[49]:
```
# Finally, it is recommended to obtain axes for plotting using specific
# plot objects that are designed to return axes; this will be particularly
# useful when we overlay annotations onto images in later practicals
# Ther example below takes the opportunity to add an explicit size
# Compare with the code above....
fig, ax = plt.subplots(figsize=(8,8))

ax.plot(t, y) # Note: this is a method of ax, not plt!
ax.set_xlabel('Time') # How annoying is THAT inconsistency in syntax?!?!
ax.set_ylabel('Time')
ax.set_title('sine(t) for t between 0 and 2 pi') # I know, right?
```

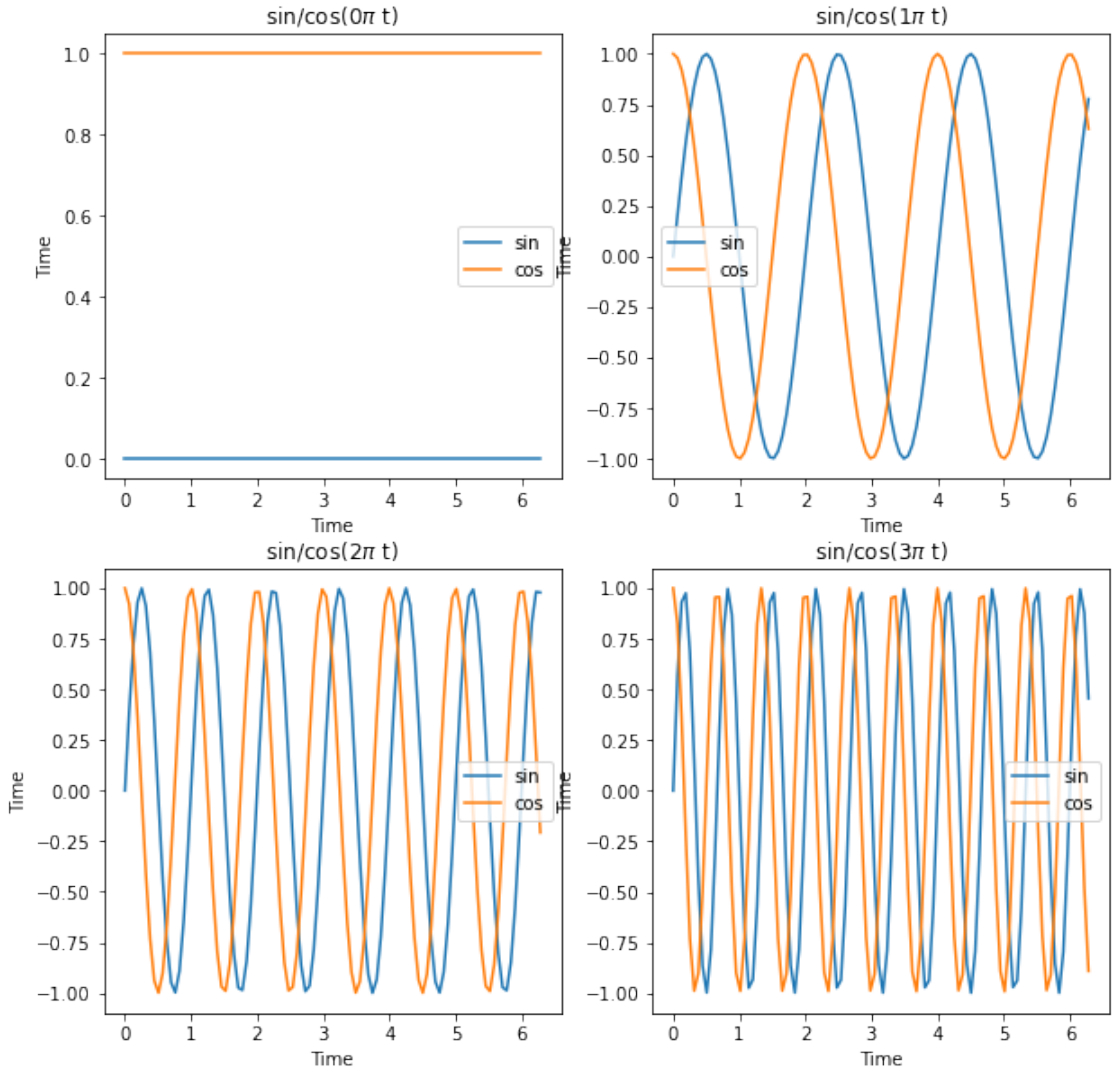
[49]:
```
Text(0.5, 1.0, 'sine(t) for t between 0 and 2 pi')
```



Exercise 8 Generate plots for one cycle of periodic functions *sine* and *cosine* on the same figure (use different colours). You will have to check out the **matplotlib** documentation to do this.

Then, introduce a variable ω representing angular frequency, and instead of plotting $y = \sin(t)$ and $y = \cos(t)$, plot $y = \sin(\omega t)$ and $y = \cos(\omega t)$, where ω is known as an angular frequency. Vary the values of ω in integer multiples of π, i.e. $n\pi, n = 0, 1, 2...$ and see how the plots change. ■

[89]:



Click here for solutions to Ex. 8.

```
fig, axes = plt.subplots(2,2, figsize=(10,10))
axes = axes.flatten()

for n in range(0,4):

    omega = n*np.pi
    y1 = np.sin(omega*t)
    y2 = np.cos(omega*t)

    axes[n].plot(t, y1) # Note: this is a method of ax, not plt!
    axes[n].plot(t, y2)
```

```
    axes[n].legend(['sin','cos'])

    axes[n].set_xlabel('Time') # How annoying is THAT inconsistency in syntax?!?!
    axes[n].set_ylabel('Time')
    axes[n].set_title('sin/cos('+str(n)+'$\pi$ t)')
```

9. Functions, Modules and Packages

9.1 Functions Like all programming languages, *Python* supports the definition of functions. Typically, a function takes and returns *arguments* that represent the inputs and outputs to the function. So, for a trivial example of such a function, we might have the classic form of equation to define a straight line relationship between a variable x and y:

[50]:
```
# The following "reset" directive is included to make some aspects of functions␣
 ↪clearer
%reset -f
import numpy as np
def linearrel(x,m,c):


    if np.ndim(m)>0 or np.ndim(c)>0:

        print('This function requires that m and c are scalars')
        return None

    # If we mistakenly input a list for x, we convert it to a float array
    x = np.asarray(x, dtype=float)

    # You should recognise this...
    y = m*x + c

    return y
```

[51]:
```
result=linearrel([1,2,3,4],1,2)
print('Result is:', result)
```

```
Result is: [3. 4. 5. 6.]
```

Variables `x`, `m`, `c` and `y` have *scope* (i.e. are recognised) only within the function, and will be undefined (or will retain any original state that was present before the calling of the function). We can also return multiple variables, by listing them with `return`, and also listing them on the left of the assignment (=) operator:

```
def myfunc(...):
    return a, b, c
```

and on calling:

```
first, second, third = myfunc(...)
```

the value of output variable `a` will be passed to the variable `first`, `b` to the variable `second` and `c` to `third`. The type of the variable as defined in `myfunc()` will be maintained as well.

We can also do things like this:

```
def myfunc(...):
    return (a, b, c)
```

and on calling:

```
result = myfunc(...)
```

the variable `result` will be a tuple containing three elements.

One more things to mention about functions:

You can nest `def` commands, so that you can have functions that are only seen by the outer function definition. Thus, in a notebook, if I have a `def myfunc()` that defines a function, then have something like `def mysubfunc()` indented as if nested under `myfunc()`, then `mysubfunc()` can be seen and called by `myfunc()`, but is not callable from outside of `myfunc()`.

Modules *Modules* are typically collections of functions that are stored in a .py file. So, if we have a *Python* file containing several function definitions (i.e. several `def` statements of functional form) then we can import those functions from that file using something like the `import` functions we have seen before.

So, let's say that the file `draw.py` has functions that are called `square()`, `circle()` and `rectangle()`; we can then do something like (these do not actually exist unless you define such a file!):

```
from draw import square, circle

# draw a square
square()

# draw a circle
circle()
```

We would not be able to call the `rectangle()` function without loading it with the `import` command, but we can also import all the functions in the fictional `draw` module using:

```
import draw

# draw a square
draw.square()

# draw a circle
draw.circle()
```

```
# draw a rectangle
draw.rectangle()
```

Packages To define packages, one usually creates a directory for them that typically sits below the root directory that runs the main *Python* code or notebook; the modules and functions that need to be called by the external and internal functions of that are then defined in an `_init_.py` function. An example of such a directory structure for our hypothetical `shapes` package might look something like this (this is a directory structure/list of files):

```
shapes/
         |---- _init_.py
         |---- circle.py
         |---- rectangle.py
         |---- square.py
```

and the file `_init_.py` can be empty, or can contain some code that is used to initialise the package on calling one of its modules or functions.

[52]:
```
from shapes import circle, square
```

```
Init of the shape package
```

[53]:
```
dir(circle) # Shows what functions and attributes exist
```

[53]:
```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'draw']
```

[54]:
```
dir(square)  # Shows what functions and attributes exist
```

[54]:
```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'draw']
```

[55]:
```
square.draw()
```

```
square
```

10. Classes

Object Oriented (OO) programming follows several paradigms, some of which are intended to make code more easily reusable (which I seriously question). But one of the more sensible ideas introduced by OO programming is that it is often convenient and appropriate to combine the data used during specific computations with the functions that are tailored to that data. Hence, we have the notion of an image as a *Python* object, and this is used by PIL, the *Python Image Library*; we will meet this library in the first practical. Here is a simple definition of a class:

[56]:
```
class HelloWorldFromPi:
    """A simple example class"""

    def __init__(self):
        # A "special" function that is automatically called when
        # an object with this class is created
        print('Thanks for creating me!')

    # Define some attributes of this class
    MyValue = 3.14159265359
    MyName = "Pi"

    def MyGreeting(self):

        # Note how the attributes are referred to: this avoids clashes with
        # other variables that might be defined outside the class
        MyGreeting = 'My name is ' + self.MyName + '. ' + 'Hello, World!'

        return MyGreeting
```

[57]:
```
# "Instantiate" an object with the class HelloWorldFromPi
x = HelloWorldFromPi() # Note the brackets
```

```
Thanks for creating me!
```

We can now "get at" the *attributes* of this object:

[58]:
```
x.MyValue
```

[58]:
```
3.14159265359
```

And we can invoke the methods attached to this object, like so:

[59]:
```
# The class-specific function (known as a method) returns a string; print it!
print(x.MyGreeting())
```

```
My name is Pi. Hello, World!
```

Finally, like we did with the module/package in **Section 9**, we can do the following:

[60]:
```
dir(x)
```

[60]:
```
['MyGreeting',
 'MyName',
 'MyValue',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

[61]:
```
type(x.MyValue)
```

[61]:
```
float
```

[62]:
```
type(x.MyGreeting)
```

[62]:
```
method
```

11. Concluding Remarks

Python is a remarkably rich language, and there are multiple ways of achieving certain results. There is no way that we can spend longer going into the intricacies of *Python*: that would be totally unrealistic. Instead, I expect you to return to these notes as you need to recall how to structure certain control statements, and to get examples of list comprehension, and `numpy` syntax.

Be aware that a key skill is to be able to figure out where to find the right information. The internet

is a fabulous place for this, but it can also contain examples that refer to old versions of *Python* libraries, or perhaps syntax that is obsolete (deprecated). Use caution and learn as best as possible to read the documentation and experiment with little code fragments to make sure you know what a command or function does. Also, make use of `type` and `dir` and `help` commands….

Acknowledgements: Many thanks to K Balaji who provided feedback on early versions of this NB, and GTAs from 2021/22 who are providing feedback as this is being run for the first time!

[ ]: