Skill-03-Numpy-Student

September 12, 2024

1 Skill Homework #3

1.1 # Modules, `Numpy`, and Reading & Writing Data Files

2 Modules (Libraries)

One of the most important concepts in good programming is to reuse code and avoid repetitions.

The idea is to write functions and classes with a well-defined purpose and scope, and reuse these instead of repeating similar code in different part of a program (modular programming). The result is usually that readability and maintainability of a program is greatly improved. What this means in practice is that our programs have fewer bugs, are easier to extend and debug/troubleshoot.

These packages of reusable code are called *modules* in python. Other languages refer to them as *libraries* or *include* files.

Python supports modular programming at different levels. Functions and classes are examples of tools for low-level modular programming. Python modules are a higher-level modular programming construct, where we can collect related variables, functions and classes in a module. A python module is defined in a python file (with file-ending `.py`), and it can be made accessible to other Python modules and programs using the `import` statement.

Consider the following example based on the previous homework: the file `mymodule.py` contains simple example implementations of a variable and a function. The curious line `%%file game.py` is a jupyter command (not a python command) that writes the code in the cell to the file `game.py`.

Run the cell below.

[1]:
```
%%file mymodule.py
"""
Example of a python module. Contains a variable called answer,
and a function called game.

The triple quotes are python's way of creating a multiple line string.
The convention for modules is that the should have a multiple line string
at the beginning of the module file that contains information on the
module.

19a - Converted to python 3
24a - Convert formatting to f-strings
"""
```

```python
import random as r

answer = r.randint(1,10)

def game():
    global answer
    done = False
    nGuesses = 0
    while not done:
        guess = int(input("Enter a guess from 1 to 10: "))
        nGuesses += 1
        if guess > answer:
            print("Your guess is too big")
        elif guess == answer:
            print("Correct!")
            done = True
        else:
            print("Your guess is too small")

        if done == True:
            print("Game over, it took you ", nGuesses, " guesses to get the␣
 ↪right answer")
            # generate a new answer
            answer = r.randint(1,10)
    return
```

```
Writing mymodule.py
```

2.1 Importing a Module and Getting Help

To use a module, you have to *import* it. The `import` statement has the syntax

```
import module
```

or

```
import module as name
```

where the second form allows you to abbreviate the name of the module. If you look at the code for `mymodule` above, you can see I imported the module `random` and gave it a shorter name of `r`.

2.2 Module `help`

If the author of a module has been kind, then they included an information string at the top of the module like I did above in `mymodule`. All modules distributed with python have these strings (called `docstrings`.) These are part of python's help system. You can type `help()` at a python prompt to get an interactive help. Type `quit` to quit.

In the cell below import `mymodule` and type `help(mymodule)`.

```
[ ]:
```

2.3 Accessing a Module

To access functions and variables in a module you use the syntax `module.function`, for example. Run the game in `mymodule` in the cell below by executing `mymodule.game()`.

```
[ ]:
```

2.3.1 Cheating

BTW, you could cheat in the game above by printing `mymodule.answer` before playing the game.

3 `numpy` the Mad-Good Math Module

Ok. You have come this far in the python skills homework without doing anything math-like. That is because most of the great numerical stuff is a module called `numpy`.

The `numpy` module is used in almost all numerical computation using Python. It is a package that provide high-performance vector, matrix and higher-dimensional data structures for Python. It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), performance is very good.

(By the way, it is modeled on MATLAB, so the usage is very similar to that commercial program. MATLAB is a very commonly used program in engineering companies.)

To use `numpy` you need to import the module, so run the following line in the cell below:

```
import numpy as np
```

BTW, this is the standard way `numpy` is imported.

```
[ ]:
```

arrays are at the heart of the numpy module.

In the `numpy` package the terminology used for vectors, matrices and higher-dimensional data sets is *array*.

3.1 Creating `numpy` arrays

There are a number of ways to initialize new numpy arrays, for example from * a Python list or tuples * using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc. * reading data from files

3.2 From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy array` function. For example

```
xPosition = np.array([0.0, 1, 2, 4])
```

In `numpy` arrays all the items have the same data type, and that type is determined by the longest data type in the list. Note that that one item was a `float` by using `0.0`. If I did not do that, the array would all be `int`s.

In the cell below, create the array `xPosition`, then print it out. Also print the type of `xPosition` by runnning the line `print(type(xPosition))`. (The `type()` function works on everything in python.

[ ]:
```
```

Note that all of the numbers had a trailing decimal point confirming that they are all `float`s.

3.3 From functions

Some commonly used `numpy` functions to create arrays are `linspace`, `arange`, `zeros`, and `ones`.

The function `linspace` has the usage `linspace(start, stop, nPoints)`. The last argument is optional; if you do not supply it, you get 50 elements in the array. The first element is `start`, and the last element is `last`. The default type is `float`. Create an array named `x` using `linspace(0,10)` and one named `y` using `linspace(0,10,51)`. Print them out.

Which one has nice even values in it? And why?

Hint: (Don't forget the `np.` before the function name.) **Make the code in the cell below work!**

[ ]:
```
x = np.linspace(
print(x)
y = np.linspace(
print(y)
```

4 Length, Size, and Shape

`Numpy` arrays are not limited to being linear; they can be two or even higher dimensional. For example, try

```
matrix = np.array([[0.0,1,2],[3,4,5]])
print(matrix)
```

makes a two-dimensional matrix that is three columns long and two rows high. Go ahead and run this in the cell below.

[ ]:
```
```

You can access elements by having two indexes between the brackets `[` and `]`. Print out elements `matrix[0,1]` and `matrix[1,0]` in the cell below.

NOTE: if not all of the output shows up, click to the right of the cell execution number, `[8]`, for example

[ ]:
```
```

5 Element-Wise Operations

Arrays (1D and multidimensional) do not behave like matrices when they are a part of mathematical operations. Instead, the operations happen element by element. To help you see this run the following code:

```
time = np.linspace(0,5,6)
print("time = ", time)
print("2 * time = ", 2 * time)
xPos1 = 1 + 2 * time
print("xPos1 = ", xPos1)
xPos2 = 2 * time**2 # remember ** is raise to a power.
print("xPos2 = ", xPos2)
```

Run this in the cell below.

[ ]:
```

```

Make sure you understand the results before you move on.

5.1 Operating on Two (or More) Arrays

Next, two arrays can be operated on. for example, from the previous array you can calculate, for example

```
newX = xPos1 + xPos2
xProd = xPos1 * xPos2
```

Print each arry out.

One caution: for these operations to work, both operand arrays must have exactly the same shape. Try these below, print out the result for each, and you will see the result has the same shape as each operand.

[ ]:
```

```

5.2 Other Operations

Almost any other operation you can image will work like this, including subtraction, division, remainder, even comparison operations. These can be strung together. As an example in the cell below print out the polynomial `1 + 2 * xPos1 + 3 * xPos **2`

[ ]:
```

```

6 Higher Math Functions

The `numpy` module also provides functions that operate on arrays and return array. (You can write these functions, too, with some care.) All of the functions you know and love, like `sqrt`, `sin`, `cos`, `exp`, `atan`, and many more are in the `numpy` module. to use these you have to use the `np.fcn` syntax. For example `print np.sin(time)` will calculate the `sin` of the variable `time` you created earlier.

Go ahead and try the `sin` example. Also print out $\sin(time)^2 + \cos(time)^2$ as well. (What should you get for this?) You should be able to guess the answer for that.

The number π is also defined ar np.pi. Print it out, too.

[ ]:

```

```

7 Statistical Functions

Once you have taken some data, the first business is to answer some basic statistical questions about the data. The most common are what is the min, max, and average. You also often want to know what is the standard deviation of the average is. `numpy` has functions to do these basic statistical calculations. The maximum and minimum functions are `np.max` and `np.min`; the average and standard deviation functions are `np.average` and `np.std`. The standard deviation of the average is standard deviation of the data divided by the number of points. (Remember the `len` function?)

In the cell below print out the max, min, average, standard deviation of the average and standard deviation of the data `newX`.

[ ]:

```

```

7.1 Formating Strings

Python has three ways of printing formatted numbers. I will only show the the newer "F-strings" method. This also works with micropython.

Another topic that can get tedious is how to format output, print, and strings. The simplest structure of a format string is something like `f"{np.pi:.4f` where the first `f` means this is an *f-string*. Next, inside the f-string is a pair of brackets. The bracket should contain a float constant or variable. Here I use `np.pi`, the value of . To use an optional format you need a color, `:` followed by a format, here `.3f` to format a `float`. Format strings also take one or more precision numbers. Here are example of some commonly used format strings: * `.2f` - `float` with two decimal places * `6d` - `int` or `long` printed in 6 spaces. * `.3e` - `float` as an exponential with three decimal places, like "3.142e0" for pi. * `10s` - output a string stretched to 10 character spaces.

7.2 Using Formatting Strings

Here are some examples:

```
msg = "Hello"
amp1 = 3.141592653
amp2 = -2.71828
nPts = 47
```

format string usage	output
f"You said {msg}"	"You said Hello"
f"Pi equals {amp1} but rounds to {amp1:.3f}"	Pi equals 3.141593 but rounds to 3.142
f"One million is {1000000:.1e}"	"One million is 1e+06"

format string usage	output
f"A small number is {1.0/1000000:.1e}"	"A small number is 1.0e-06"
f"There are {nPts} points with a min of {amp2:.2f}"	"There are 47 points with a min of -2.72"

Print each of these example.

[ ]:

```

```

Now go back and print the statistics above with a message and the numbers formatted to one decimal place after the decimal point. For example the max should look something like `Min = 1.0`.

Print the `min`, `max`, `average`, `sigma` and `sigma(avg)`.

[ ]:

```

```

8 Random Numbers

The `numpy` module also provides all of the random number generators you can imagine. They are in a submodule named `np.random`, so executing `help(np.random)` will give you most of the information you need about them. Random numbers are very useful in modeling experimental data.

Here are some of the functions used the most:

```
np.seed(5) # starts the random number generator the same each time
np.random.random() # returns a float in the range [0,1)
np.random.random(10) # returns 10 floats in the range [0,1)
np.random.uniform(10,15) # random numbers uniformly from [10,15)
np.random.uniform(10,15,20) # returns an array of 20 of the above
np.random.randint(2,5) # returns a random including 2 but NOT 5
np.random.randint(2,5,20) # returns an array of 10 of the above
np.random.normal() # return one number from the normal distribution
                   # with mean = 0 and sigma = 1
np.random.normal(101,3.2) # one number with mean=101
                          # and signa = 3.2
np.random.normal(101,3.2, 30) # an array of 30 of the above
np.random.shuffle(myArray) # randomly shuffles myArray in place
```

In the cell below print out at least five different examples from the functions given above.

[ ]:

```

```