COMP1730/COMP6730
Programming for Scientists

Introduction to NumPy arrays

Announcements

* Mid-Semester Survey closed yesterday
  – 170 responses (approx). **Thanks!**
  – Your feedback will be used to improve your experience
* Homework 5 (4%) is open and due on **01/10/2023, 11:55pm** (end of semester week 8)

Recap of 1st half and outline for 2nd half

So far:

* Functional decomposition
* Types and expressions
* Branching, `if else`
* Iteration, `while` & `for` loop
* Sequences, `list`, `tuple`, `str`
* Code quality
* Debugging & testing
* Data analysis & visualisation

What's next?

* NumPy arrays (**today**)
* Files, Input/Output
* Dictionaries and sets
* Exception handling
* Complexity, big-O notation
* Dynamic programming
* Computational Science
* Another advanced topic or 2

Many, if not most, concepts also apply to other programming languages, not just Python!

Lecture outline

* Minimal math background on vectors and arrays
* Differences among NumPy arrays and lists
* Working with 1-rank NumPy arrays
  – Creating 1-rank NumPy arrays
  – Indexing and slicing. Views versus copies.
  – Vectorized code/Vectorization
* Working with 2-rank NumPy arrays

Minimal math background on vectors

- Vectors are typically introduced in high school math courses, e.g., point (x, y) in the plane, point (x, y, z) in space
- In general, a vector v can be mathematically defined as a tuple with n numbers: $v = (v_0, v_1, \ldots, v_{n-1})$
- We can use lists to represent vectors; v_i is stored at `v[i]`
- However, in this lecture, we introduce a new sequence data type to represent mathematical vectors (and arrays, next slide) in the computer, the so-called **NumPy arrays**

Minimal math background on arrays

- **Arrays** are a generalization of vectors where we can have more than one index. Examples: $A_{i,j}$ (2 indices) and $B_{i,j,k}$ (3 indices)
- Example: table of numbers (called matrix by mathematicians); one index for the row, another for the column

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix} \qquad A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,3} \\ \vdots & \ddots & \vdots \\ A_{2,0} & \cdots & A_{2,3} \end{bmatrix}$$

- The number of indices in an array is the **rank** or **number of dimensions** of the array
- Example: vectors are rank-1 arrays (or one-dimensional arrays)
- The main reason behind using **NumPy arrays** instead of nested lists to represent mathematical arrays in the computer is **higher computational efficiency**

NumPy Arrays as opposed to lists (I)

$\boxed{\text{NumPy arrays} \neq \text{lists}}$

Main differences among NumPy arrays and lists:

- **A NumPy array can keep ONLY elements of the same type**, typically `int`, `float`, or `complex` numbers, whereas a list can mix objects of different types
- **NumPy arrays have a fixed size at creation**, unlike lists which can shrink and grow dynamically. Changing the size of a NumPy array will create a new array and delete the original
- **NumPy arrays can have arbitrary dimensions** (e.g., previous slide) whereas lists are one-dimensional (although, as seen in Lectures 11/12, they can be nested to, e.g., emulate rank-2 arrays)

NumPy Arrays as opposed to lists (II)

$\boxed{\text{NumPy arrays} \neq \text{lists}}$

Main differences among NumPy arrays and lists:

- **Code written using NumPy arrays might be vectorized**, that is, rewritten in terms of operations on entire arrays at once (without Python loops), **resulting in much faster code** versus lists
- **NumPy arrays are NOT built-in in Python**, but provided by an external library/module (called NumPy from Numerical Python)
  - Standard practice is to import it as `import numpy as np`
  - NumPy arrays are of type `np.ndarray`
  - Documentation available here (not part of Python documentation)
  - It has many different features, here we only cover the basics

Examples NumPy array creation (rank-1 arrays)

The first thing one does with a NumPy array is to create it

```
>> import numpy as np

>> l = [0.0, 0.5, 1.0, 1.5]
>> x = np.array(l) # Convert list into array of floats
>> x
array([0. , 0.5, 1. , 1.5])

>> n = 5 # Length of the two arrays created below

# Create rank-1 array of n floats and initialize it with zeros
>> x = np.zeros(n)
>> x
array([0., 0., 0., 0., 0.])

# Create rank-1 array with n equispaced floats over interval [0,1]
>> x = np.linspace(0.0, 1.0, n)
>> x
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```


Array indexing and slicing

- Arrays are **sequence** types
- Arrays indexing works in the same way as lists (Lecture 7)

```
>>> import numpy as np
>>> x = np.linspace(0.0, 1.0, 5)
>>> x
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
>> x[2]+x[-1]
1.5
```

- Slicing (Lecture 7) can also be used with arrays

```
>>> x[1:4] # (from:to; half-open)
array([0.25, 0.5 , 0.75])
```

- As opposed to lists, legal to assign single number to array slice

```
>>> x[1:3] = 10.0
>>> x
array([ 0.  , 10.  , 10.  ,  0.75,  1.  ])
```


NumPy array views versus copies (I)

- As with lists, array assignment does **not** copy its elements

```
>>> import numpy as np
>>> x=np.linspace(0.0, 1.0, 5)
>>> y=x
>>> y[-1]=1000.0
>>> x
array([0.0e+00, 2.5e-01, 5.0e-01, 7.5e-01, 1.0e+03])
```

- **However**, as opposed to lists, array slicing does **not** return a copy but a "view" to original array

```
>>> y=x[1:4]
>>> y[-1]=1000.0
>>> x
array([0.0e+00, 2.5e-01, 5.0e-01, 1.0e+03, 1.0e+03])
```


NumPy array views versus copies (II)

- The `np.copy` function returns a copy of the array

```
>>> import numpy as np
>>> x=np.linspace(0.0, 1.0, 5)
>>> y=np.copy(x)

>>> y[-1]=1000.0
>>> x
array([0.  , 0.25, 0.5 , 0.75, 1.  ])

>>> y=np.copy(x[1:4])
>>> y[-1]=1000.0
>>> x
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Vectorized code

- With NumPy, it is possible to **work with entire arrays at once** versus using Python loops to process one element at a time
- Code written using this feature is called **vectorized code**
- **Example**: we want to evaluate the mathematical function

$$f(x) = \sin(x)e^{-x}$$

at 10^6 equispaced points in the interval $[0, 2\pi]$, and store the result in a NumPy 1-rank array

Option 1 - Non-vectorized code

Evaluates $f(x) = \sin(x)e^{-x}$ at 10^6 equispaced points within $[0, 2\pi]$

```
import math
import numpy as np
n = 1_000_000
x = np.linspace(0, 2*math.pi, n)
y = np.zeros(n)
for i in range(0,len(x)):
    y[i] = math.sin(x[i])*math.exp(-x[i])
```

Remarks:

- Python `for` loop to evaluate $f(x)$ one element at a time
- Code pretty similar to the one that one would write with lists
- We can use `math.sin` and `math.exp` from the `math` module as we are evaluating them with one array element at a time

Option 2 - Vectorized code

Evaluates $f(x) = \sin(x)e^{-x}$ at 10^6 equispaced points within $[0, 2\pi]$

```
import math
import numpy as np
n = 1_000_000
x = np.linspace(0, 2*math.pi, n)
y = np.zeros(n)
y = np.sin(x)*np.exp(-x)
```

Remarks:

- No Python `for` loop (i.e., vectorized code)
- **Much faster**, as it is very efficiently handled by NumPy under the hood
- Shorter, more readable code, closer to math notation
- **IMPORTANT(!):** we cannot use `math.sin` and `math.exp` on entire arrays, we **must** use NumPy versions `np.sin` and `np.exp` instead

Non-vectorized VS vectorized code (performance)

```
In [1]: import math
In [2]: import numpy as np
In [3]: n = 1_000_000
In [4]: x = np.linspace(0.0, 2.0*math.pi, n)
In [5]: y = np.zeros(n)

In [6]: %timeit for i in range(0,len(x)): \
   ...:     y[i] = math.sin(x[i])*math.exp(-x[i])
147 ms ...

In [7]: %timeit y = np.sin(x)*np.exp(-x)
11.8 ms ...
```

- Code run on laptop with Intel i7-1265U microprocessor
- Measurements taken with `%timeit` magic command in IPython
- 147 VS 11.8 millisecs. (**vectorized code ≈12.5 times faster!**)

Vectorized functions

- A function `f(x)` written for a single number `x` usually also works for an array of numbers `x`

```
>> import numpy as np
>> def f(x):
...     return x**3 + np.sin(x)*np.exp(-3*x)

>> x = 1.0
>> y = f(x)
>> y
1.0418943734502046

>> x = np.linspace(0.0, 1.0, 5) # array([0., 0.25, 0.5, 0.75, 1.])
>> y = f(x)
>> y
array([0.  , 0.13249036, 0.2319743 , 0.4937192 , 1.04189437])
```

- **Exercise**. Write a “scalar” version of `f(x)` (i.e., that works with one element of `x` at a time) and compare its performance versus vectorized function above with arrays of increasing length

Vectorized in-place arithmetics (I)

Consider these two mathematically equivalent statements:

```
a = a + b
a += b
```

In practice, much more subtle:

- `a=a+b` is computed in two steps as (**extra array** `w` **needed**):
  Step 1: `w=a+b`
  Step 2: `a=w`
- The variable `a` is reassigned to a new array `w` in Step 2
- **However**, `a+=b` is computed as `a[i]+=b[i]` for each `i`, and thus **no extra array is needed**
- `a+=b` is an **in-place addition**: it changes each element in `a` rather than letting the name `a` refer to a new array (result of `a+b`)

Vectorized in-place arithmetics (II)

- When writing functions that modify NumPy arrays passed as arguments, use in-place arithmetics (otherwise, the changes won't to be visible outside the function)

```
>> def update_array_wrong(a,b):
...     a=a+b

>> def update_array_in_place(a,b):
...     a+=b

>> x=np.array([1.0,2.0,3.0])
>> y=np.array([10.0,20.0,30.0])

>> update_array_wrong(x,y)
>> x
array([1., 2., 3.])

>> update_array_in_place(x,y)
>> x
array([11., 22., 33.])
```


Rank-2 arrays recap (math)

A table of numbers (called matrix by mathematicians) such as:

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix}$$

can be represented as a 2-rank array A_{ij} with (row identifier) $i = 0, 1, 2$ and (column identifier) $j = 0, 1, 2, 3$

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,3} \\ \vdots & \ddots & \vdots \\ A_{2,0} & \cdots & A_{2,3} \end{bmatrix}$$

Rank-2 arrays with NumPy

Example: create and fill a rank-2 NumPy array using indexing

```
import numpy as np
A=np.zeros((3,4)) # Create a 2-rank array with 3 rows, 4 columns
A[0,0]=0
A[0,1]=12
A[0,2]=-1
A[0,3]=-5
A[1,0]=-1
...
A[2,3]=-2

# One can also write (as for nested lists):
A[2][3]=-2
```

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix}$$

Creating rank-2 NumPy arrays from nested lists

- One can also use a nested list to create a rank-2 NumPy array
- Each element in the nested list represents a different row in the rank-2 array

```
>> import numpy as np
>> A = np.array([[0.0,12.0,-1.0,5.0],
...              [-1.0,-1.0,-1.0,0.0],[11.0,5.0,5.0,-2.0]])
>> A
array([[ 0., 12., -1.,  5.],
      [-1., -1., -1.,  0.],
      [11.,  5.,  5., -2.]])
```

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix}$$

Shape of NumPy array

- The shape of a NumPy array is a tuple with the number of elements in each dimension
- The length of this tuple is the rank of the array
- One can access the shape of a NumPy array using the `shape` attribute

```
>> import numpy as np
>> A = np.array([[0.0,12.0,-1.0,5.0],
...              [-1.0,-1.0,-1.0,0.0],[11.0,5.0,5.0,-2.0]])
>> A.shape
(3,4)
>> len(A.shape)
2                  # A is a rank-2 array!
```

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix}$$

Looping over rank-2 array entries

- One can loop over the entries of a rank-2 array using a nested loop (e.g., outer loop over the rows, inner loop over the columns)

```
>> import numpy as np
>> A = np.array([[0.0,12.0,-1.0,5.0],
...              [-1.0,-1.0,-1.0,0.0],[11.0,5.0,5.0,-2.0]])

>> for i in range(0,A.shape[0]):
...    for j in range(0,A.shape[1]):
...        print("A["+str(i)+","+str(j)+"]=",A[i,j])
A[0,0]= 0.0
A[0,1]= 12.0
A[0,2]= -1.0
A[0,3]= 5.0
A[1,0]= -1.0
...
A[2,3]= -2.0
```

2-rank NumPy array slicing

- One can also use slicing with rank-2 arrays
- `A[i,:]` is row `i` (same as `A[i]`)
- `A[:,j]` is column `j`
- `:` can also be *from*`:`*to* (equivalent to *from*`:`)

```
>> import numpy as np
>> A = np.array([[0.0,12.0,-1.0,5.0],
 ...             [-1.0,-1.0,-1.0,0.0],[11.0,5.0,5.0,-2.0]])

>> A[1,:] # Row 1; equivalent A[1,0:] and A[1,0:A.shape[1]]
array([-1., -1., -1.,  0.])

>> A[:,3] # Column 3; equivalent A[0:,3] and A[0:A.shape[1],3]
array([5.0., 0.,  -2.])
```

- **Exercise:** is `A[1:3,1:3]` legal? what does it return?

More on vectorized code (broadcasting)

- NumPy allows one to write vectorized operations among a single number and an array
- E.g., if `A` rank-2 NumPy array, and `a` number, one can write `a*A`

```
>> import numpy as np
>> A = np.array([[0.0,12.0,-1.0,5.0],
...               [-1.0,-1.0,-1.0,0.0],[11.0,5.0,5.0,-2.0]])
>> a=2.0
>> a*A
array([[ 0., 24., -2., 10.],
       [-2., -2., -2.,  0.],
       [22., 10., 10., -4.]])
```

- The result is an array with the same shape as `A` where each element is multiplied by `a`
- Actually this is just an example of a more general feature called **broadcasting** that allows one to perform operations among “compatible” arrays of different shapes

Programming problem idea

Write a vectorized version of homework 2 where the neural network is evaluated on a batch of inputs at once (instead of one input at a time)

Take home messages

- NumPy arrays and lists are types with different features
- NumPy arrays less flexible but much faster than lists (if used wisely)
- Vectorization is the process of turning a non-vectorized algorithm with Python loops accessing single array elements into a vectorized version without Python loops
- Vectorization can make scientific programs working with a large number of numerical data much faster