BCH2203 Python - 4. More data analysis

Ramses van Zon

2 February 2022

Some common issues from assignment 1

- The names of variables, functions and modules matters.

  If they are unclear, or even misleading, changes are you will use them incorrectly.

  https://levelup.gitconnected.com/a-guide-for-naming-things-in-programming-2dc2d74879f8

- Use for loops to count.

So not this:

```
i=1
while i<11:
   print(i)
   i=i+1
```

but this:

```
for i in range(1,11):
   print(i)
```

- Avoid computing the same things twice, or reading the same file twice.
- Limit the scope of the with statement.

So not:

```
with open("thisfile.txt") as f:
   lines = f.readlines()
   # more things to do with "lines"
   # but not with "f"
```

But rather:

```
with open("thisfile.txt") as f:
   lines = f.readlines()
# more things to do with "lines"
# but not with "f"
```

Working with Strings and Lists

Strings and lists are very similar

- A string can be viewed as a list of characters.
- While that is not entirely technically correct, they behave very similarly.

For example

Get an element from a list:

```
>>> mylist = [1,2,"ab"]
>>> print(mylist[1])
2
```

and from a string:

```
>>> mystring = "12ab"
>>> print(mylist[1])
2
```

Go over every element of a list:

```
>>> for element in mylist:
...     print(element)
...
1
2
ab
```

of a string:

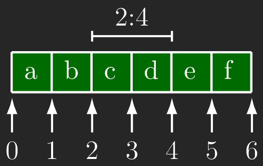
```
>>> for element in mystring:
...     print(element)
...
1
2
a
b
```

A major difference is that string elements cannot be assigned to, and always contains characters.

Slicing

- Given a string or list x, you can get a substring or sublist by using an **index range** instead of a single number between square brackets. This is called **slicing**.

```
>>> x = "abcdef world!"
>>> print(x[2:4])
cd
>>> print("Hello, " + x[7:13])
Hello, world!
>>> z = [1,2,3,4]
>>> print(z[1:3])
[2, 3]
```



- If no number before colon, starts from the beginning.

```
>>> print("Goodbye, " + x[:5])
Goodbye, abcde
```

- If you omit the number after the colon, the slice goes to the end:

```
>>> print("Good morning, " + x[7:])
Good morning, world!
```

- Negative indices indicate positions from the end of the string/list.

```
>>> print(x[:-1] + "?")
abcdef, world?
```

- Get every nth element by adding :n to the slice.

```
>>> print(x[::2])
ace ol!
```

Finding in Python Lists

- `lst.index(val)` returns the index of first occurance of `val` in the list. E.g.

```
>>> lst = [4, 3, 2, 4, 1, 6, 3, 4]
>>> lst.index(4)
0
```

- `lst.index(val, start)` returns the index of first occurance of `val` in the list starting from index `start`. E.g.

```
>>> lst.index(4,1)
3
```

- `lst.count(val)` give the number of occurances.

- To match things based on more than equality, you'll need if statements, possibly within list comprehensions. E.g, to get all even numbers:

```
>>> [x for x in lst if x%2 == 0]
[4, 2, 4, 6, 4]
```

- To get all the indices, you'd need more code, e.g.

```
>>> def indices(lst, val):
...     indices = []
...     lastindex = -1
...     Nindices = lst.count(val)
...     while len(indices) < Nindices:
...         nextindex = lst.index(val, lastindex+1)
...         indices.append(nextindex)
...         lastindex = nextindex
...     return indices
...
>>> indices(lst,4)
[0,3,7]
```

or use list comprehension...

```
>>> def indices(lst, val):
...   return [i for i,x in enumerate(lst) if x==val]
...
>>> indices(lst,4)
[0,3,7]
```

We can also use list comprehension for other cases, e.g. to find the indices of even numbers.

Finding in Strings

All previous techniques for finding things in lists work also for searching substring in strings.

But there are some additional techniques available as well.

- `s.find(substring)` is similar to `index`, but returns -1 of a substring is not found (instead of giving an error). E.g.

```
>>> s = "Hello, world!"
>>> s.find("world!")
7
>>> s.find("World!")
-1
```

- `s.startswith(substring)` checks if a string starts with `substring`.

```
>>> s.startswith('Hell')
True
```

- To find more complicated patterns, you can use regular expressions from the (built-in) `re` module.

Replacing

- For lists, replacing a single element can be done using indexing:

```
>>> lst = [1,2,3]
>>> lst[1] = 42
>>> print(lst)
[1, 42, 3]
```

- This does not work for strings, they are **immutable**. Instead, you'll need to create a new string.

```
>>> s = "abc"
>>> s2 = s[:1] + "B" + s[2:]
>>> print(s, s2)
abc aBc
```

- Python strings have a method to find and replace substrings:

```
>>> seq = "AUGAUCUCGUAA"
>>> print(seq.replace("UAA","*"))
AUGAUCUCG*
>>> print(seq.replace("UAA","*").replace("AUG","Met").replace("UCG","Ser").replace("AUC","Ile"))
MetIleSer*
```

- There are more 'find-and-replace' type methods. Check docs.python.org/3/library/stdtypes.html#string-methods
- To replace more complex patterns, again, regular expressions from the `re` module could be used.

From strings to lists and back again

- Sometimes, one may have a large string that one wants to divide up. E.g. we may want to split up a sentence into words.

  We can use the string `split` method for that:

```
>>> s = 'Hello, beautiful world!'
>>> words = s.split()
>>> print(words)
['Hello,', 'beautiful', 'world!']
```

  Note that the result is a list of strings.

- We can use a different separator character instead of a space:

```
>>> notwords = s.split('o')
>>> print(notwords)
['Hell', ', beautiful w', 'rld!']
```

- Given a list of strings, we can concatenate them with `join`.

  join is a method of the separator to insert, e.g.

```
>>> expressive = 'oooo'.join(notwords)
>>> print(expressive)
Helloooo, beautiful woooorld!
```

Working with Arrays

Numerical arrays

For numerical work, the python-native lists aren't the ideal data type.

Lists can do funny things that you don't expect, if you're not careful.

- Lists are just a collection of items, of any type.
- If you do mathematical operations on a list, you won't get what you expect.
- These are not the ideal data type for scientific computing.
- Arrays are a much better choice, but are not a native Python data type.

```
>>> a = [1, 2, 3, 4]
>>> a
[1, 2, 3, 4]
>>>
>>> b = [3, 5, 5, 6]
>>> b
[3, 5, 5, 6]
>>>
>>> 2 * a
[1, 2, 3, 4, 1, 2, 3, 4]
>>>
>>> a + b
[1, 2, 3, 4, 3, 5, 5, 6]
>>>
```

Lists vs. Arrays

Lists: optimized for flexibility

- Can hold any type
- Can grow
- Are one-dimensional
- Do not have out-of-the-box element-wise operations

Arrays: optimized for speed

- Single type
- Fixed size
- Multi-dimensional
- Have optimized element-wise operations

Arrays are what we want to use: NumPy

Almost everything numerical that you want to do starts with . . .

NumPy has been around for a while, but a review article about it just appeared in *Nature*:

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature **585**, 357–362 (2020).

https://www.nature.com/articles/s41586-020-2649-2

https://numpy.org

NumPy Basics

- Before using it we need to import it in Python:

```
>>> import numpy as np
```

- We can then create an array of any shape and type, e.g.

```
>>> a = np.ndarray([3,4], dtype=int)
```

  The content of this 3x4 integer array is undefined.

- NumPy has functions to create arrays of various types and forms: `zeros`, `ones`, `linspace`, *etc.*

- `linspace` creates an array of 50 equally spaced values between two limits. The number of values can be changed using an optional third arguments.

Examples:

```
>>> import numpy as np
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
>>> np.ones(5, dtype = int)
array([ 1,  1,  1,  1,  1])
>>> np.zeros([2,2])
array([[ 0., 0.],
       [ 0., 0.]])
>>> np.arange(5)
array([ 0,  1,  2,  3,  4])
>>>
>>> np.linspace(1,5)
array([ 1.,   1.08163265,
        1.16326531,  1.24489796,


        4.67346939,  4.75510204,
        4.83673469,  4.91836735,  5. ])
>>> np.linspace(1, 5, 6)
array([ 1., 1.8, 2.6, 3.4, 4.2, 5.])
```

Specifying data types

```
>>> x = np.float32(7.4e-3)
>>> a = np.array([[1,2,3],[4,5,6]],dtype=np.float32)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]], dtype=float32)
>>>
>>> b = np.ndarray((2,3),dtype=np.float16)
>>> b
array([[ -1.51875000e+01,   5.11169434e-02,              nan],
       [  0.00000000e+00,  -3.12500000e+01,   4.35709953e-05]], dtype=float16)
```

- Integers:
  `int8 int16 int32 int64 uint8 uint16 uint32 uint64`
  Number indicates number of bits.
- Floats of half, single and double precision:
  `float16 float32 float64`
- Complex numbers in single and double precision:
  `complex64 complex128`

Accessing array elements

Elements of arrays are accessed using square brackets.

- Like most languages, the first index is the row, the second is the column.
- Indexing starts at 0.
- You cannot assign values outside the index range (unlike e.g. in R).

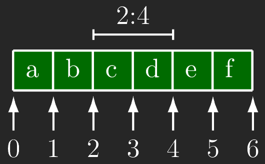
```
>>> np.zeros([2, 3])
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
>>> a = np.zeros([2,3])
>>>
>>> a[1,2] = 1
>>> a[0,1] = 2
>>> a
array([[  0.,   2.,   0.],
       [  0.,   0.,   1.]])
>>>
>>> a[2,1] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 2 is out of bounds for axis
  0 with size 2
>>>
```

Slicing arrays

You can select a subset of an NumPy array using slicing, just as with a list.

- An index range looks like “a:b”, e.g. “2:4”. So `a[2:4]` selects those elements of an array a.
- Read “2:4” as “from the beginning of the element at index 2, to the beginning of that at index 4”.
- Or read it as: index 2 is the first you get, index 4 is the first you do not get.
- Negative indexing is supported.
- If a third index is specified, it refers to the step size (“1:10:2”, for example).
- If no index is specifed, either “beginning” or “end” is assumed.

```
>>> a = np.array([1,2,3,4,5,6,7])
>>> print(a[2])
3
>>> print(a[2:4])
[3,4]
>>> print(a[::2])
[1,3,5,7]
```



Slicing arrays, continued

Elements in an array can also be selected using a boolean array. Boolean arrays can be created using a conditional expression.

```
>>>
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a > 2
array([False, False, False, True, True], dtype=bool)
>>> a[a > 2]
array([3, 4])
>>> a[(a % 2) == 0]
array([0, 2, 4])
>>>
```

The “%” symbol is the modulus operator.

Array arithmetic

Looping over arrays

- In Python, loops over arrays are performed over the first index.
- To go over all elements of a multidimensional array a without using nested loops, use `a.ravel()` or `a.flat`

  (or `a.flatten()` if you need a copy).

```
>>> a = np.array([1,2,3])
>>> for e in a:
...     print("element:", e)
element: 1
element: 2
element: 3
>>>
>>> a = np.array([[1,2],[3,4]])
>>> for e in a:
...     print("element:", e)
element: [1 2]
element: [3 4]
>>>
>>> for e in a.ravel():
...     print("element:", e)
element: 1
element: 2
element: 3
element: 4
>>>
```

Shape and reshape

- NumPy allows you to modify the shape of an array once it already exists.
- You can only change the shape to one which contains the same number of elements.
- Also, note that `reshape` creates a new view of the array data, and doesn't change the shape of the original array.

```
>>> a = np.arange(8)
>>> a.shape
(8,)
>>>
>>> a.reshape([2,4])
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> a.reshape([2,4]).shape
(2, 4)
>>>
>>> a.reshape([2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of
  size 8 into shape (2,3)
>>>
```

Vector-vector & vector-scalar arithmetic

1-D arrays are often called 'vectors'.

- When vectors are added, subtracted, multiplied, divided, etc. you get element-by-element operations.
- This is much faster than executing a python loop or list comprehension.
- When vectors are add, subtracted, multiplied, divide, . . . , by a scalar, you also get element-wise operations.

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>>
>>> b = np.arange(4.) + 3
>>> b
array([ 3.,  4.,  5.,  6.])
>>>
>>> c = 2
>>> c
2
>>>
>>> a * b
array([  0.,   4.,  10.,  18.])
>>> a * c
array([0, 2, 4, 6])
>>> b * c
array([  6.,   8.,  10.,  12.])
```

Peculiar matrix-vector multiplication

A 2-D array is sometimes called a 'matrix'.

- Matrix-scalar multiplication gives **element-wise** multiplication.

```
>>> a = np.array([[1,2,3],[2,3,4]])
>>> b = np.array([1, 2, 3])
>>> a * b
array([[ 1,  4,  9],
       [ 2,  6, 12]])
```

Python matrix-vector multiplication:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_1 & a_{12} \cdot b_2 & a_{13} \cdot b_3 \\ a_{21} \cdot b_1 & a_{22} \cdot b_2 & a_{23} \cdot b_3 \end{bmatrix}$$

Note: This is very different from matrix-vector multiplication in linear algebra!

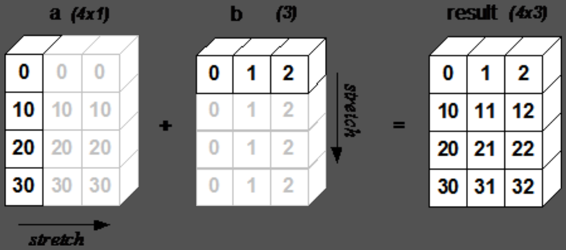
Vector broadcasting

This peculiar multiplication is result of element-wise operations plus **broadcasting**.

Python will perform vector broadcasting if you perform a matrix-vector operation:

- Python repeatedly applies the vector to the matrix.
- The length of the vector must equal the last dimension of the matrix.

```
>>> a = np.array([[0],[10],[20],[30]])
>>>
>>> b = np.array([0,1,2])
>>>
>>> a + b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```



Summing array elements

- Let's create a 2x4 matrix:

```
>>> m = np.array([[0,1,2,3],[3,1,1,2]])
>>> print(m)
[[0 1 2 3]
 [3 1 1 2]]
```

- We can sum all the elements of an array with `.sum()`

```
>>> print(m.sum())
13
```

- But we can also just sum the columns with `axis=0`:

```
>>> columnsums = m.sum(axis=0)
>>> print(columnsums)
[ 3 2 3 5 ]
```

- Or the rows:

```
>>> rowsums = m.sum(axis=1)
>>> print(rowsums)
[ 6 7 ]
```

- For integer arrays, we can count the occurances of each value, with `bincount` e.g

```
>>> counts = np.bincount(m.flat)
>>> print(counts)
[1 3 2 2]
>>>
>>> columnsumcounts = np.bincount(columnsums)
>>> print(columnsumcounts)
[0 0 1 2 0 1]
>>>
>>>
>>> rowsumcounts = np.bincount(rowsums)
>>> print(rowsumcounts)
[0 0 0 0 0 0 1 1]
```

- For floating point arrays, there's `np.histogram` to make histograms.

Assignment 2

Assignment 2: Survey analysis with numpy and function.

Here's a reference solution for assignment 1:

```python
#!/usr/bin/env python3
# Read the survey file
filename = "2022-01-19_survey-results.txt"
with open(filename) as f:
    entries = f.readlines()
# Determine the size of the data
nquestions = len(entries[0])-1
nentries = len(entries)
# Count yes's per question
nyesperquestion = [0 for i in range(nquestions)]
for entry in entries:
    for i in range(nquestions):
        if entry[i]=='Y': nyesperquestion[i] += 1
# Count entries with specific number of 'Y's
nentriesyescount = [0 for i in range(nquestions+1)]
for entry in entries:
    nentriesyescount[entry.count("Y")] += 1
# Output to screen
for i in range(nquestions):
    percent = int(100*nyesperquestion[i]/nentries)
    print ("Question",i,"Y:",percent,"%")
for i in range(nquestions+1):
    print ("Entries with",i,"Ys",nentriesyescount[i])
```

Split this into three functions:

- One function should read the data and store it in a 2-dimensional numpy array containing 0's and 1's (such that N becomes 0 and Y becomes 1).
- A second function should take such a numpy array as an argument and compute, for each of the questions, the percentage of 'N' and 'Y' answers.
- A third function should take such a numpy array and print the number of surveys with zero 'Y' answers, then the number of surveys with one 'Y' answer, with two 'Y' answers, . . . , etc.
- The script should then call these functions to achieve the same result as before.

Use numpy functions where you can.
Try this out on the data file from assignment 1.
Use best practices.