CS319-Week06-notebook

February 19, 2025

1 CS319 Week 06 : Experimental Algorithm Analysis (in Python)

In this notebook we will * revise a little about working with lists in `Python`; * revise using 'numpy' nad `numpy` arrays; * do a little plotting with `matplotlib`. * I'll assume you know about f-strings

1.1 Storage types in Python

In C++ we use arrays to store collections of values of the same type.

In Python, there are several ways of doing this: * **Lists**: a list is a mutable (= changeable) ordered collection. Duplicates are allowed. * **Tuples**: a tuple is an immutable ordered collection. Duplicates are allowed. * **Sets**: a (Python) set is a mutable unordered collection without duplicates. * **Dictionaries**: a dictionary is an unordered but indexed mutable collection without duplicate indices.

But today we only care about `lists` and, later `numpy` arrays.

1.2 Lists

- **A `list` is a sequence of values**. But, whereas a string is a sequence of characters, a list can be a sequence of any type.
- The values in a list are called *items* or *elements*.
- The list starts with `[`, ends with `]`, and items are separated by commas.

We'll remind ourselves of… * What a list is * How to create one * Modifying items in a list * Indexing and slicing * How to traverse a list with a `for` loop * Some functions/methods for operating on a list

1.3 Making lists

1.3.1 Making a list with [and]

The simplest way to create a list is using square backets, `[` and `]`, with a comma between elements.

```
[1]: [0, 1, 2, 3, 4] # a list of integers
```

```
[1]: [0, 1, 2, 3, 4]
```

```
[2]: ['zero', 'one', 'two', 'three'] # a list of strings
```

```
[2]: ['zero', 'one', 'two', 'three']
```

Usually, we assign a variable name to the list.

[3]:
```
my_favourite_numbers = [0, 4, 1024]
```

[4]:
```
print(f"{my_favourite_numbers}")
```

```
[0, 4, 1024]
```

[5]:
```
print(f"The list has {len(my_favourite_numbers)} items")
```

```
The list has 3 items
```

1.4 More about lists in Python

The next parts are not so important for today: you can skip to the section on numpy, if you like

1.4.1 A list with different types of items

An item in a list can be just about anything. And items in a list can be of different types from each other. This list includes strings, an integer, a float, and a boolean.

[6]:
```
mixed_list = ["CS319", "Scientific Computing", 22, 78.5, True]
print(mixed_list)
```

```
['CS319', 'Scientific Computing', 22, 78.5, True]
```

[7]:
```
type(mixed_list)
```

[7]:
```
list
```

[8]:
```
type(mixed_list[2])
```

[8]:
```
int
```

You can even make a list with includes another list as an item. This is called *nesting*

[9]:
```
code         = "CS319" # string
instance     = "3BS2"  # string
num_students = 22      # int
ave_grade    = 87.5    # float
has_exam     = False;  # boolean
modules = ["CS319", "MA378", "CS211", "MA385"]  # list

new_list = [code, instance, num_students, ave_grade, has_exam, modules ]
print(new_list)
```

```
['CS319', '3BS2', 22, 87.5, False, ['CS319', 'MA378', 'CS211', 'MA385']]
```

```
[10]: new_list[5][0]
```

```
[10]: 'CS319'
```

1.4.2 len()

There are many operations that can be preformed on lists. One of the most important is counting the number of items in a list. In Python, the `len()` function can be applied to various types, including `lists`.

```
[11]: modules_in_3BS2 = ["CS300", "MP311", "MA322", "ST333", 'CS344']
      number_of_modules = len(modules_in_3BS2)
      print(f"You can choose from {number_of_modules} modules in 3BS2")
```

```
You can choose from 5 modules in 3BS2
```

1.4.3 Making a list with list()

One can also use the function `list()` to create a new list from an object, such as a `range` of numbers, or a `string`.

```
[12]: list_of_numbers = list(range(10))
      print(list_of_numbers)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[13]: list("hello")
```

```
[13]: ['h', 'e', 'l', 'l', 'o']
```

1.4.4 Indexing

- Lists are indexed from zero
- the first element is `L[0]`, the second is `L[1]`, etc.
- The last is `L[-1]` which is the same as `L[len(L)-1]`.

```
[14]: modules_in_3BS2
```

```
[14]: ['CS300', 'MP311', 'MA322', 'ST333', 'CS344']
```

```
[15]: modules_in_3BS2[0]  # first item
```

```
[15]: 'CS300'
```

```
[16]: modules_in_3BS2[-1]  # last item
```

```
[16]: 'CS344'
```

1.4.5 Slicing

We create a sub-list from a string using the "`colon`" notation. The syntax is `L[a:b]` which is the same as `[L[a], L[a+1], ..., L[b-1]]`

```
[17]: print(modules_in_3BS2)
```

```
['CS300', 'MP311', 'MA322', 'ST333', 'CS344']
```

```
[18]: print(modules_in_3BS2[0:3])  # first 3
```

```
['CS300', 'MP311', 'MA322']
```

```
[19]: print(modules_in_3BS2[1:4])  # middle 3
```

```
['MP311', 'MA322', 'ST333']
```

```
[20]: print(modules_in_3BS2[2:5])  # last 3
```

```
['MA322', 'ST333', 'CS344']
```

We can also use the notation `a:b:s` which means `start at a, go to b in steps of s`.

```
[21]: print(modules_in_3BS2[0:5:2])  # every second one
```

```
['CS300', 'MA322', 'CS344']
```

If you leave out either `a` or `b` they are assumed to be the start and end of the list, if `s` is positive.

```
[22]: print(modules_in_3BS2[::2])  # every second one
```

```
['CS300', 'MA322', 'CS344']
```

1.4.6 More list functions

- Add a new item to the end

```
[23]: modules_in_3BS2 = modules_in_3BS2 + ["DS555"]
      print(f"The modules in 3BS2 are : {modules_in_3BS2}")
```

```
The modules in 3BS2 are : ['CS300', 'MP311', 'MA322', 'ST333', 'CS344', 'DS555']
```

- Similarly `L.extend(list2)` adds the items in `list2` to the end of `L`.
- `L.append("new item")` adds a single new item to a list.
- `L.insert(<position>, <item>)` adds a new `<item>` to the specified `<position>`. Anything to the right of that position is shuffled right.
- `L.remove(entry)` removes the specific entry from a list (error if it is no present)
- Use `del` to remove an entry by index.

1.5 Taversing a list with a `for` loop

`for` loops work well with lists. One form of the `for` statement is

```
for <var> in <list>:
    <body>
```

It consists of a **heading** and a **body** The heading, between the keyword `for` and the colon (`:`) introduces a **loop variable** `<var>` and refers to a list `<list>`. The `<body>` is a consistently indented sequence of statements.

When executed, a `for` statement results in the execution of its `<body>` of statements once for each item of the list (in order). The particular item for each iteration is stored as the value of the variable `<var>`.

[24]:
```
numbers = [10, 3, -1, -11, 12]
numbers
```

[24]:
```
[10, 3, -1, -11, 12]
```

[25]:
```
for item in numbers:
    print(item, end="--**--")
```

```
10--**--3--**---1--**---11--**--12--**--
```

Often we want to change items in the list. In the following example, we'll take a list of 3BS2 modules, and add a prefix of `2324-` to each code. For that we'll need to keep count of the items

[26]:
```
print(modules_in_3BS2)
```

```
['CS300', 'MP311', 'MA322', 'ST333', 'CS344', 'DS555']
```

[27]:
```
i = 0
for item in modules_in_3BS2:
    modules_in_3BS2[i]='2324-'+item
    i=i+1

print(modules_in_3BS2)
```

```
['2324-CS300', '2324-MP311', '2324-MA322', '2324-ST333', '2324-CS344',
'2324-DS555']
```

There are lots more we could cover on lists, Map/Filter/Reduce, the `in` operator, adding and multiplying lists, sorting, etc…

But they are not so relevant for CS319. So we'll skip (for now).

1.6 numpy

`numpy` (pronoucned "/numb-pee/" or "/numb-pie/") is the primary module for working with arrays of data in Python.

We use the word `array` in the same sense it is used in C++: it is a collection of items all of the same data type. Working with numpy arrays is MUCH faster than with lists.

There are two reasons why 1. It is easy to locate any element in memory. 2. Most of the data types and functions are actually implemented in C++, rather than Python.

1.6.1 Loading the module

To use `numpy` you must import the module. By convention, this is done as:

[28]:
```
import numpy as np
```

There are lots of ways of creating an `numpy` array. For example, to make an array from a list, use the `np.array()` function:

[29]:
```
N=np.array([4, 8, 16, 32, 64, 128, 256, 512])
type(N)
```

[29]:
```
numpy.ndarray
```

[30]:
```
T=np.array([8.9400e-03, 2.2367e-03, 5.5930e-04, 1.3983e-04, 3.4958e-05, 8.
 ↪7396e-06, 2.1849e-06, 5.4622e-07])
print(f"Contents of T: {T}")

print(f"T is of type{type(T)}")
```

```
Contents of T: [8.9400e-03 2.2367e-03 5.5930e-04 1.3983e-04 3.4958e-05
8.7396e-06
 2.1849e-06 5.4622e-07]
T is of type<class 'numpy.ndarray'>
```

Use the `dtype` method to check what the underlying type of the data is:

[31]:
```
print(f"Data in T is of type {T.dtype}")
print(f"Data in N is of type {N.dtype}")
```

```
Data in T is of type float64
Data in N is of type int64
```

1.7 Functions of numpy arrays

One of the many benefits of using `numpy` is that we can avoid writing explicit loops, since `numpy` comes with functions that map arrays to other arrays.

Example:

[32]:
```
print(N)
print(np.log2(N))
```

```
[  4   8  16  32  64 128 256 512]
[2. 3. 4. 5. 6. 7. 8. 9.]
```

Also, standard operators such as `+` and `*` can be used with these arrays:

[33]:
```
N+N
```

[33]:
```
array([   8,   16,   32,   64,  128,  256,  512, 1024])
```

[34]:
```
T*N*N # spoiler!!!!
```

[34]:
```
array([0.14304   , 0.1431488 , 0.1431808 , 0.14318592, 0.14318797,
       0.14318961, 0.14318961, 0.1431883 ])
```

1.8 Analysing the Quadrature Data

We'll need the `matplotlib` library, as well as `numpy`

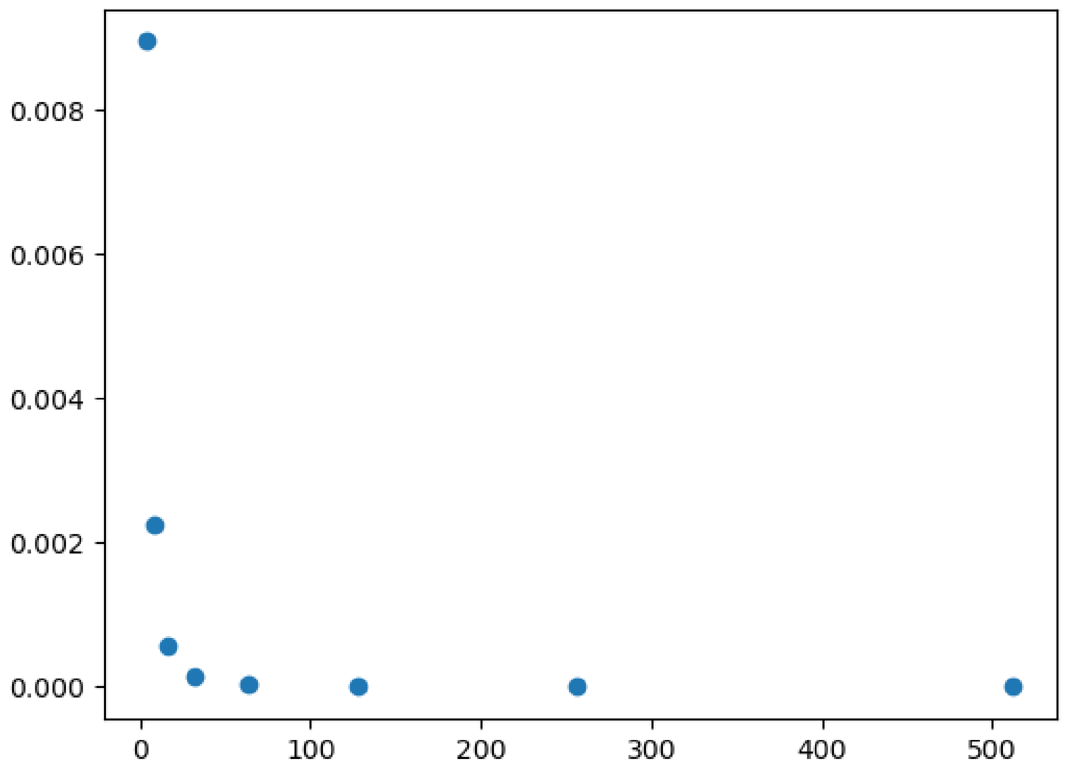
[35]:
```
import numpy as np
import matplotlib.pyplot as plt
```

We copy some data computed by `00CheckConvergence.cpp` * `N` is the set of numbers of intervals used in the calculations * `T` is the set of values of `E_N` for the Trapezium Rule.

[36]:
```
N=np.array([4, 8, 16, 32, 64, 128, 256, 512])
T=np.array([8.940076e-03, 2.236764e-03, 5.593001e-04, 1.398319e-04, 3.
  ↪495839e-05, 8.739624e-06, 2.184908e-06, 5.462270e-07])
```

Plot the data, using `matplotlib`

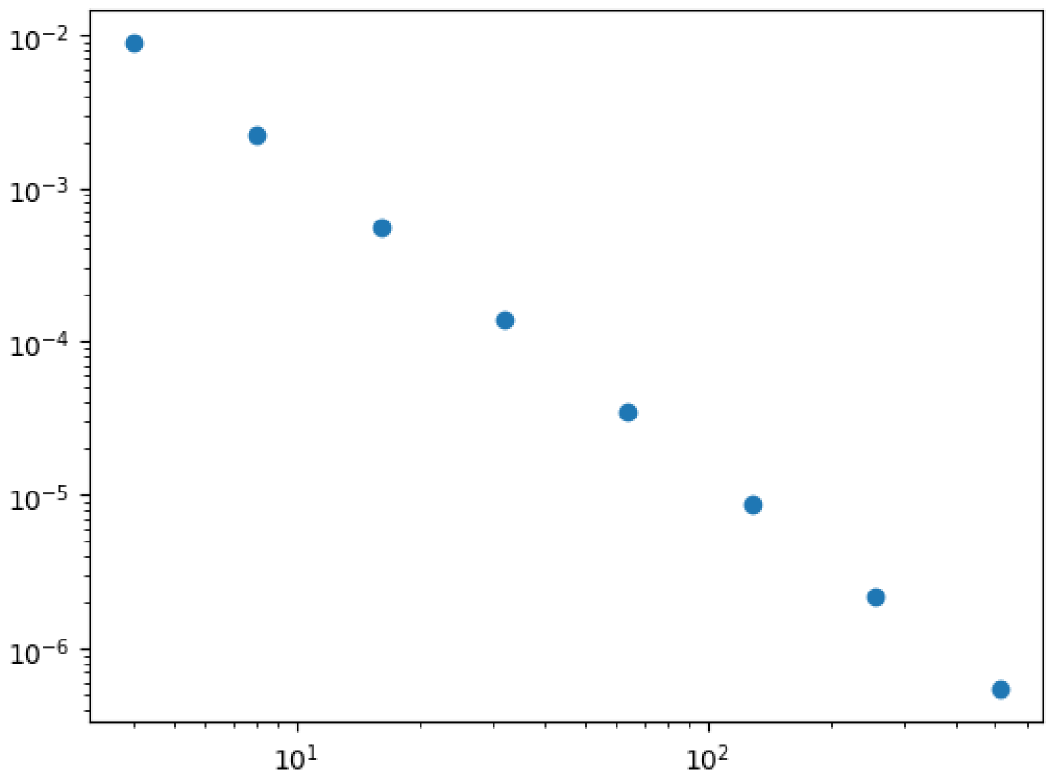
[37]:
```
plt.plot(N, T, 'o')
plt.show()
```



That last figure is not very informative... * on the horizontal axes, the data are too spread out * on the vertical axis the data is too compressed near $T = 0$.

We resolve this by using logarithmic axes:

[38]:
```
plt.loglog(N, T, 'o')
plt.show()
```



Looks like a straight line!

As discussed in class, $E_N \approx CN^{-q}$. Then, if we set * `Y = log(T)`, * `X = log(N)`, and * `K = log(C)`,
we get $Y \approx K - qT$. We have Y and X, so we want to estimate K and q, which are the slope and Y-intercept of the line.

We'll use the (depreciated) `np.polyfit()` function to compute the coefficients of the line that best fits (in a least squares sense) the points (X, Y).

If we set `A=polyfit(x,y,n)` then `A` is a np.array with the coefficients of the polynomial of degree n that best approximates the points (x, y). That is $y \approx A[0]x^n + A[1]x^{n-1} + \cdots + A[-1]$.

[39]:
```
X = np.log(N); Y = np.log(T)
Fit = np.polyfit( X, Y,1)
q = -Fit[0];
K = Fit[1];
print(f'We get K={K} and q={q}');
```

```
We get K=-1.9443306386200654 and q=1.9998490653205538
```

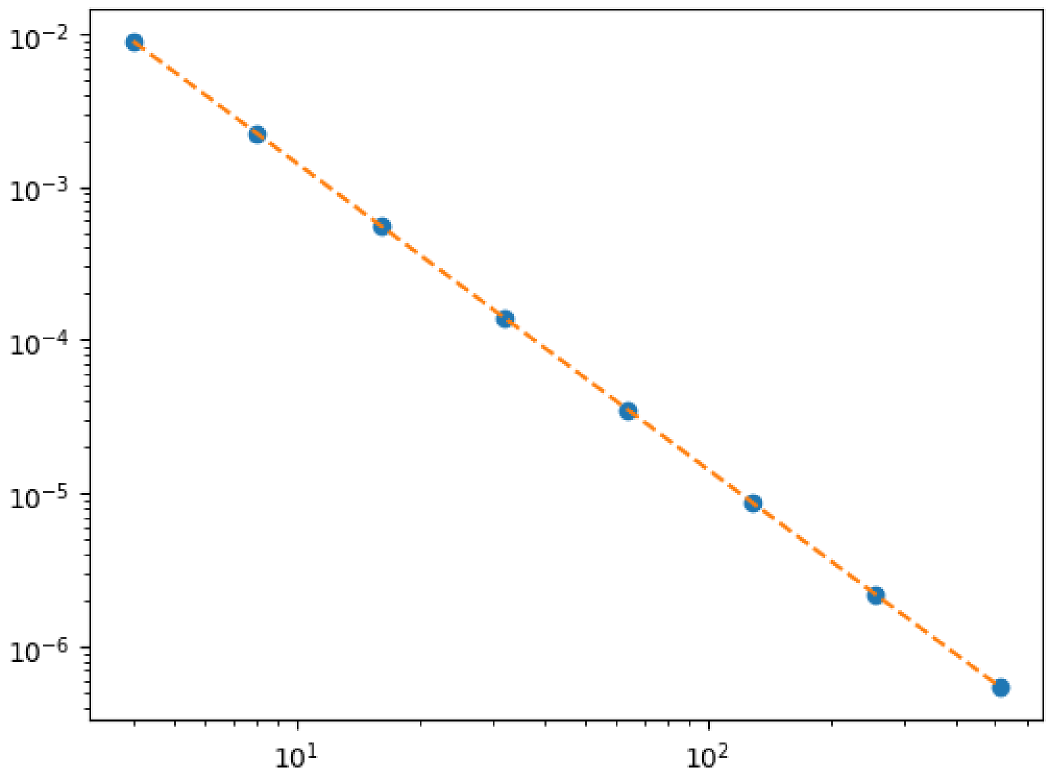
We can now recover the value of C

[40]:
```
C = np.exp(K)
print(f'C={C : .3f} and q={q : .3f}');
```

```
C= 0.143 and q= 2.000
```

Let's plot to check:

[41]:
```
plt.loglog(N, T, 'o', N, C*N**(-q), '--')
plt.show()
```



10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{1}
10^{2}