Advanced Data Processing and Visualization Techniques

Make your analyses smarter

some day in May
Schedule
<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 1st</td>
<td>Holiday</td>
<td>Jun 26th</td>
<td>NoSQL DBs</td>
</tr>
<tr>
<td>May 8th</td>
<td>Intro/Basics</td>
<td>Jul 3rd</td>
<td>ML intro</td>
</tr>
<tr>
<td>May 15th</td>
<td>Pythonics</td>
<td>Jul 10th</td>
<td>Visualization</td>
</tr>
<tr>
<td><strong>May 29nd</strong></td>
<td><strong>Vectorization</strong></td>
<td>Jul 17th</td>
<td>Interactive I</td>
</tr>
<tr>
<td>Jun 5th</td>
<td>Biological DBs</td>
<td>Jul 24th</td>
<td>Interactive II</td>
</tr>
<tr>
<td>Jun 19th</td>
<td>Pandas</td>
<td>Jul 31st</td>
<td>Recap</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Aug 7th</td>
<td>Exam</td>
</tr>
</tbody>
</table>

updated: no lecture on May 22nd and June 12th
- Updated due to the lecture period extension to August 7th
- If there is no lecture the exercise takes place one week later
- Exercises are published on Fridays and discussed Friday the week after
- Last sheet/exercise: Jul 3rd / Jul 10th (maybe adjusted)
- Exam (working date): Aug 7th, to be discussed
Vectorization
Let's create two lists containing 10,000,000 random numbers between -100 and 100:

```python
from random import randint
length = 10_000_000
numbers_1 = [
    randint(-100, 100) for _ in range(length)
]
numbers_2 = [
    randint(-100, 100) for _ in range(length)
]
```
First, we will create a new list `sums` s.t. `sums[i] == numbers_1[i] + numbers_2[i]`:

```python
def pairwise_sum(iterable1, iterable2):
    return [x + y for x, y in zip(iterable1, iterable2)]

sums = pairwise_sum(numbers_1, numbers_2)
```

We would like to measure how much time `pairwise_sum` takes, but we don’t want to modify the function.
Let’s wrap it in another function:

```python
import time

def time_decorator(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        rt = end - start
        print(f'{func.__name__}: {rt} seconds."
        return result
    return wrapper
```
And now... Magic happens (hopefully, at this point it doesn’t look like magic at all and you understand **what** and **why** is going on)!

```python
import time
timed_pairwise_sum = time_decorator(pairwise_sum)
_ = timed_pairwise_sum(numbers_1, numbers_2)
pairwise_sum: 1.0940041542053223 seconds.
```
Let’s have even more magic by using NumPy:

```python
import numpy as np
def np_pairwise_sum(np_array_1, np_array_2):
    return np_array_1 + np_array_2

np_numbers_1 = np.array(numbers_1)
np_numbers_2 = np.array(numbers_2)
timed_np_sum = time_decorator(
    np_pairwise_sum
)
_ = timed_np_sum(np_numbers_1, np_numbers_2)
np_pairwise_sum: 0.047905683517456055 seconds.
```
Second, we will take a look at dot product:

```python
@time_decorator
def dot_product(iterable1, iterable2):
    return sum(x * y for x, y in zip(iterable1, iterable2))

dot_product(numbers_1, numbers_2)

dot_product: 1.3942914009094238 seconds.

-8938246
```
What is going on on the previous slide with that @ symbol? That is Python’s syntactic sugar for applying a decorator, it reduces the amount of code we have to write, so this:

```python
def my_func():
    pass
my_func = my_decorator(my_func)
```

becomes this:

```python
@my_decorator
def my_func():
    pass
```
Now, we will calculate the dot product using NumPy:

```python
np_dot = time_decorator(np.dot)
np_dot(np_numbers_1, np_numbers_2)

<table>
<thead>
<tr>
<th>dot: 0.013365983963012695 seconds.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8938246</td>
</tr>
</tbody>
</table>
```
## Runtime Comparison

<table>
<thead>
<tr>
<th></th>
<th>Python</th>
<th>NumPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pairwise Addition</td>
<td>1.09</td>
<td>0.05</td>
</tr>
<tr>
<td>Dot Product</td>
<td>1.39</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**WHY?**
NumPy is a library implementing an N-dimensional array object and linear algebra (and other) capabilities. It is written in Python and C.
Memory:
- Persistent Storage - slow read/write speeds, high capacity (terabytes).
- RAM - faster read/write speeds, smaller capacity (gigabytes).
- L1/L2/L3 Cache - fastest read/write speeds, smallest capacity (kilobytes/megabytes).

CPU:
- clock speed - # of cycles per second,
- IPC - instructions per cycle.
Single Instruction, Multiple Data: a CPU is simultaneously provided with multiple values and performs an operation on all of them at once (SSE3).

This is also known as vectorization.

```python
def pairwise_sum(iterable1, iterable2):
    stride = 4
    result = []
    for i in range(0, len(iterable1), stride):
        result.append(
            iterable1[i:i+stride] +
            iterable2[i:i+stride]
        )

DOES NOT WORK IN PYTHON ^__^
In general, memory units perform better when they read continuous chunks of data instead of separate pieces.

Containers in Python hold references to the data they hold (instead of containing the data itself, they contain locations in memory where the data is stored).

That gives us heterogeneity, but that also means when data is moved from RAM to CPU cache, each piece is moved individually (so we don’t have several values all copied to the CPU cache and ready to be processed).

That means no vectorization.
NumPy arrays are homogeneous and store data in sequential chunks of memory*. That means several values can be moved from RAM to the CPU cache at once.

NumPy also supports vectorized operations on the data.

That means we
- don’t need to explicitly loop over elements;
- get results of our computations faster.
We can create NumPy arrays using Python lists or tuples:

```python
arr_1 = np.array([1, 2, 3])
arr_2 = np.array((4, 5, 6))
arr_1 + arr_2
```

```python
array([5, 7, 9])
```

We have mentioned that NumPy arrays contain elements of the same data type. What is the data type in this particular example?

```python
arr_1.dtype
```

```python
dtype('int64')
```
We don’t set the data type explicitly, so NumPy uses the default `int` datatype. Let’s specify a data type:

```python
arr_1 = np.array([255, 255, 255], dtype=np.uint8)
arr_2 = np.array([1, 1, 1], dtype=np.uint8)
arr_1 + arr_2
```

```python
array([0, 0, 0], dtype=uint8)
```

Everything comes with a price, and so do benefits of data types occupying fixed space in memory. We just got an overflow error!
We can check values of a data type using `np.iinfo`:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arr_1</code></td>
<td>0</td>
<td>255</td>
<td><code>uint8</code></td>
</tr>
<tr>
<td><code>np.int16</code></td>
<td>-32768</td>
<td>32767</td>
<td><code>int16</code></td>
</tr>
<tr>
<td><code>np.uint32</code></td>
<td>0</td>
<td>4294967295</td>
<td><code>uint32</code></td>
</tr>
<tr>
<td><code>np.int64</code></td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
<td><code>int64</code></td>
</tr>
</tbody>
</table>
We also have Boolean and floating-point values:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr_1 = np.array([1.0, 2.0, 3.0])</td>
<td>dtype('float64')</td>
</tr>
<tr>
<td>arr_2 = np.array([True, False, True])</td>
<td>dtype('bool')</td>
</tr>
<tr>
<td>arr_1 + arr_2</td>
<td>array([[ 2.,  2.,  4.]])</td>
</tr>
</tbody>
</table>
You will mostly use NumPy with numeric data. That said, it is also possible to use NumPy with strings:

```python
np.array(['a', 'b', 'c'])
array(['a', 'b', 'c'], dtype='<U1')

np.array(['aa', 'bb', 'cc'])
array(['aa', 'bb', 'cc'], dtype='<U2')

np.array(['a', 'bb', 'ccc'])
array(['a', 'bb', 'ccc'], dtype='<U3')
```
NumPy uses the maximum length of strings present in an array as the upper limit on all values. That can lead to a following situation:

```python
arr_s = np.array([['martin', 'oliver']])
arr_s[0] = 'andreas'
arr_s[1] = 'dmitrii'
arr_s

array([['andrea', 'dmitri'],
       dtype='<U6')
```
We can store strings of arbitrary length using `np.object`. In fact, we can store arbitrary objects in NumPy arrays using `np.object`:

```python
arr_s = np.array([['martin', 'oliver'],
                  dtype=np.object)
arr_s[0] = 'andreas'
arr_s[1] = 'dmitrii'
arr_s
```

```
array([['andreas', 'dmitrii'],
       dtype=object)
```

When we use `np.object`, we create a NumPy array storing *references* to Python objects.
We can change the data type via the `astype` method:

```python
arr_1 = np.array([1.1, 2.2, 3.3])
arr_2 = arr_1.astype(np.uint8)
arr_2
```

```
array([1, 2, 3], dtype=uint8)
```

```python
arr_s = np.array(['10', '20', '30'])
arr_s.astype(np.uint8)
```

```
array([10, 20, 30], dtype=uint8)
```
NumPy provides several functions for array creation:

<table>
<thead>
<tr>
<th>Function</th>
<th>Array Created</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>np.zeros((1, 2))</code></td>
<td><code>[[ 0., 0.]]</code></td>
</tr>
<tr>
<td><code>np.ones((2, 3))</code></td>
<td><code>[[ 1., 1., 1.], [ 1., 1., 1.]]</code></td>
</tr>
<tr>
<td><code>np.eye(3)</code></td>
<td><code>[[ 1., 0., 0.], [ 0., 1., 0.], [ 0., 0., 1.]]</code></td>
</tr>
</tbody>
</table>
Creating NumPy Arrays II

```python
np.full((2, 3), 4)
array([[4, 4, 4],
       [4, 4, 4]])

arr_r = np.random.random((2, 2))
arr_r
array([[ 0.73728063,  0.45252309],
       [ 0.8386122 ,  0.97436006]])

(100 * arr_r).astype(np.uint8)
array([[73, 45],
       [83, 97]], dtype=uint8)
```
We can change shape of an array using the `reshape` method:

```python
arr_1d = np.arange(8)
arr_1d
```

```python
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```python
arr_1d.reshape(2, 4)
```

```python
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```python
arr_1d.reshape(4, 2)
```

```python
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```
**NumPy Array: Reshaping**

```python
arr_3d = arr_1d.reshape(2, 2, 2)
arr_3d

array(
    [[[0, 1],
      [2, 3]],
     [[4, 5],
      [6, 7]]])

arr_3d[0, 1, 1]
3

arr_3d[1, 0, 0]
4
```
To transpose an array, use the `T` method or the `numpy.transpose` function:

```python
arr = np.random.randint(10, 100, size=(2,3))
arr

array([[17, 98, 85],
       [65, 24, 29]])
```

```python
np.transpose(arr)
```

```python
array([[17, 65],
       [98, 24],
       [85, 29]])
```

```python
arr.T
```

```python
array([[17, 65],
       [98, 24],
       [85, 29]])
```
We can access elements of NumPy arrays by tuple of indices or via slicing:

```python
arr_2d = np.arange(1, 10).reshape((3, 3))
arr_2d[1, 1]
5

arr_2d[:2, 1:]
array([[2, 3],
       [5, 6]])

arr_2d[1:, 2]
array([6, 9])
```
We can also access array elements using Boolean indexing:

```python
numbers = np.arange(1, 10).reshape(3, 3)
booleans = (numbers % 2 == 0)
booleans
```

```
array([[False,  True, False],
       [ True, False,  True],
       [False,  True, False]], dtype=bool)
```

```python
numbers[booleans]
```

```
array([[2, 4, 6, 8]])
```
We pass an array of indices to access multiple elements at once:

```
numbers
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

numbers[[0, 2, 1], [1, 2, 0]]
array([[2, 9, 4]])

[numbers[0, 1], numbers[2, 2], numbers[1, 0]]
[2, 9, 4]
```
We can combine simple indices with fancy indexing:

```python
numbers[2, [2, 0]]
array([9, 7])
```

We can also combine slicing with fancy indexing:

```python
numbers[1:, [1, 0]]
array([[5, 4],
      [8, 7]])
```
We can use the `len` function, but it will return number of elements in the first dimension:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(np.arange(1, 10))</code></td>
<td>9</td>
</tr>
<tr>
<td><code>len(np.arange(1, 10).reshape(3, 3))</code></td>
<td>3</td>
</tr>
</tbody>
</table>

If we want to get numbers of elements across all dimensions, we need to use the `shape` attribute:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>np.arange(1, 10).reshape(3, 3).shape</code></td>
<td>(3, 3)</td>
</tr>
</tbody>
</table>
We can use basic mathematical functions as operators or functions:

```python
arr_1 = np.array([1, 2, 3])
arr_2 = np.array([4, 5, 6])
np.add(arr_1, arr_2)
array([5, 7, 9])

arr_2 - arr_1
array([3, 3, 3])

arr_1 * arr_2
array([ 4, 10, 18])

np.divide(arr_2, arr_1)
array([ 4., 2.5, 2.])
```
In the example above multiplication means pairwise multiplication. If we want to compute the dot product, we need to use the \texttt{dot} function or method:

\begin{verbatim}
np.dot(arr_1, arr_2)
\end{verbatim}
\begin{verbatim}
32
\end{verbatim}

\begin{verbatim}
arr_1.dot(arr_2)
\end{verbatim}
\begin{verbatim}
32
\end{verbatim}
To compute a sum or product, use the corresponding methods (or functions):

```python
arr_2d = np.arange(1, 10).reshape(3, 3)
np.sum(arr_2d)
45

arr_2d.prod()
362880
```
We can also compute sum and product over a given axis:

```py
arr_2d.sum(axis=0)
array([12, 15, 18])
```

```py
np.prod(arr_2d, axis=1)
array([  6, 120, 504])
```
We can find mean and median value over the entire array:

```python
arr = (100 * np.random.random((3, 3))).astype(np.uint8)
arr
```

```
array([[88, 74, 60],
       [36, 77, 95],
       [99, 49,  3]], dtype=uint8)
```

```python
np.mean(arr)
```

```
64.555555555555557
```

```python
np.median(arr)
```

```
74.0
```
We can also do it over a specific axis:

\[
\text{np.mean(arr, axis=0)}
\]
\[
\begin{array}{c}
74.33333333, 66.66666667, 52.66666667 \\
\end{array}
\]

\[
\text{np.median(arr, axis=0)}
\]
\[
\begin{array}{c}
88., 74., 60. \\
\end{array}
\]
Math on Arrays

We can find min and max elements:

```
np.min(arr)
3
```

```
np.min(arr, axis=0)
array([[36, 49,  3],
       [88, 95, 99]], dtype=uint8)
```

```
np.max(arr, axis=1)
array([[88, 95, 99]], dtype=uint8)
```
We can also find indices of min and max elements:

```python
np.argmax(arr, axis=0)
array([2, 1, 1])
```

```python
np.argmin(arr, axis=1)
array([2, 0, 2])
```
NumPy provides a wealth of mathematical functions: trigonometric functions, hyperbolic functions, rounding functions, exponents and logarithms, and many others.

Remember to check the documentation!
We can sort an array using `np.sort`:

```python
np.sort(np.array([2, 3, 1, 5, 4]))
array([1, 2, 3, 4, 5])
```

```python
np.sort(
    np.array([[2, 5, 6], [1, 4, 3]]),
    axis=0
)
array([[1, 4, 3],
       [2, 5, 6]])
```
We can get indices of elements satisfying a specific condition with `np.where`:

```python
indices = np.where(arr > 60)
arr[indices]
```

```
array([88, 74, 77, 95, 99], dtype=uint8)
```
We can perform pairwise operations on arrays of different shapes*:

```python
long_arr = np.array([[1, 2], [3, 4], [5, 6]])
short_arr = np.array([10, 100])
long_arr * short_arr
```

```
array([[ 10, 200],
       [ 30, 400],
       [ 50, 600]])
```
Broadcasting allows us to use a smaller array several times together with a larger array as long according to the following rules:

- if the arrays do not have same number of dimensions, prepend 1 to the shape of the smaller one until they do;
- arrays are compatible in a dimension if they have the same size in a given dimension OR if a smaller array has size 1;
- a smaller array acts as if it was copied along those dimensions where its size is 1.
In this example dimensions are not compatible and broadcasting doesn’t work:

```python
long_arr = np.array([[1, 2, 3], [4, 5, 6]])
short_arr = np.array([10, 100])
long_arr * short_arr
```

```
ValueError Traceback (most recent call last)
<ipython-input-144-3698e23e512a> in <module>()
 1 long_arr = np.array([[1, 2, 3], [4, 5, 6]])
 2 short_arr = np.array([10, 100])
----> 3 long_arr * short_arr

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```
When we slice an array, we do not create a copy, we create a *view*:

```python
arr = np.arange(1, 10).reshape(3, 3)
sub_arr = arr[1:, :2]
sub_arr
```

```
array([[4, 5],
       [7, 8]])
```

```python
sub_arr[0, 1] = 100
arr
```

```
array([[ 1,  2,  3],
       [ 4, 100,  6],
       [ 7,  8,  9]])
```
A view is a new array object that refers to the same data:

```python
arr_view = arr.view(); arr_view
```

```
array([[ 1,  2,  3],
       [ 4, 100,  6],
       [ 7,  8,  9]])
```

```
arr_view is arr
```

```
False
```

```python
arr[2, 2] = 999; arr_view
```

```
array([[ 1,  2,  3],
       [ 4, 100,  6],
       [ 7,  8, 999]])
```
Reshaping a view doesn’t affect the underlying data:

```python
arr = np.arange(1, 7).reshape(2, 3)
arr_view = arr.view().reshape(3, 2); arr_view

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

```python
arr_view[1, 1] = 40; arr_view

array([[ 1,  2],
       [ 3, 40],
       [ 5,  6]])
```

```python
arr

array([[ 1,  2,  3],
       [40,  5,  6]])
```
The copy method returns a new array containing a copy of data:

```python
arr = np.arange(1, 7).reshape(2, 3)
arr_copy = arr.copy()
arr_copy[1, 0] = 9; arr_copy

array([[1, 2, 3],
       [9, 5, 6]])
```

```python
arr

array([[1, 2, 3],
       [4, 5, 6]])
```
NumPy started its life in the namespace of SciPy.

SciPy is a Python library providing modules for optimization, linear algebra, integration, interpolation, etc. etc. etc. It is written in Python, Fortran, C, and C++.

Matplotlib is a plotting library for the SciPy ecosystem. If you use NumPy or Pandas - you are going to need to know Matplotlib to plot your results.
CuPy is an implementation of n-dimensional array on CUDA. It supports a subset of NumPy’s arrays. The authors claim that some operations were sped up more than 100 times, however, CuPy speeds up operations on arrays with more than 10 million values.

```python
import cupy as cp
x = cp.arange(6).reshape(2, 3).astype('f')
x
```

```
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
```

```python
x.sum(axis=1)
```

```
array([ 3., 12.], dtype=float32)
```
Think of Dask like Spark for the SciPy ecosystem. Dask provides ability to run your code on a cluster of machines while using a NumPy-compatible API.

```python
import dask.array as da
x = da.random.random(size=(10000, 10000), chunks=(1000, 1000))
x + x.T - x.mean(axis=0)
```
Decorators allow us to change behaviour of a decorated function. Decorators modify *definitions* (every time we invoke a function, we get the modified behaviour).

We can manipulate passed arguments or a returned value. We can even return an entirely unrelated function! This is way more than just executing code before and after a decorated function is executed.
Vectorization refers to converting an algorithm from operating on a single value at a time to operating on several values at a time (SIMD). We need to have several values ready in the CPU cache for that. Python does not provide vectorization capabilities.
NumPy n-dimensional arrays store data in contiguous chunks of memory and possess optimizations to perform vectorized operations.