Analytical Computing

Lecture 2: Array Handling

hogeschool Windesheim

Radboud University

Agenda

- Arrays
- Numpy
- Werken met dimensies
- Van Pandas naar NumPy
- NumPy in-depth
- Samenvatting

Planning

20-4-2021 **Git**

22-4-2021 **Array Handling**

T.b.d.

Arrays

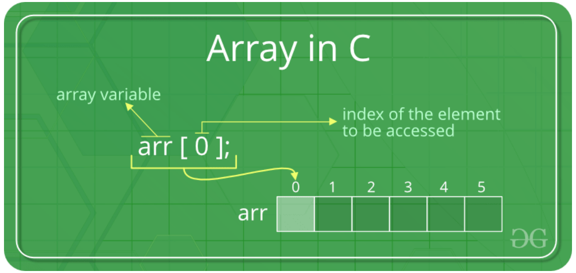
Wat is een array

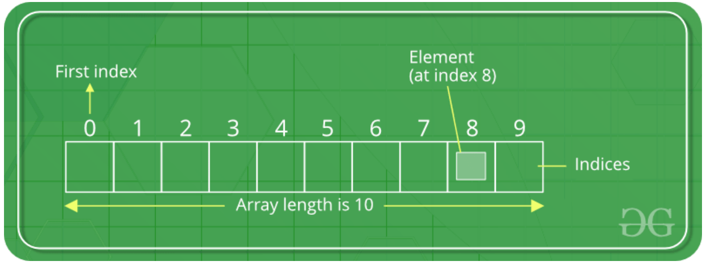
“A collection of items stored at contiguous memory locations”

- An array is a special type of variable
  - It can hold more than one value at a time
  - You can call it like any other variable
- Values in an array are stored in the memory of the computer

Wat is een array

- Define an array in Python as follows: `my_array = []`
  - `my_array` = variable name
  - `[]` = indicating that this variable is an array
- We can insert values in an array as follows: `my_array = ["Hello", "World"]`
  - Above array consists of 2 values, namely “Hello” and “World”
- Each value or “element” in an array gets assigned an index number
- Index **always** starts at “0”
- So: if I have 10 values in my array, its max index is 9
- If I want to return the 8th element in the array, I pick the value with index 9





* https://www.geeksforgeeks.org/python-arrays/

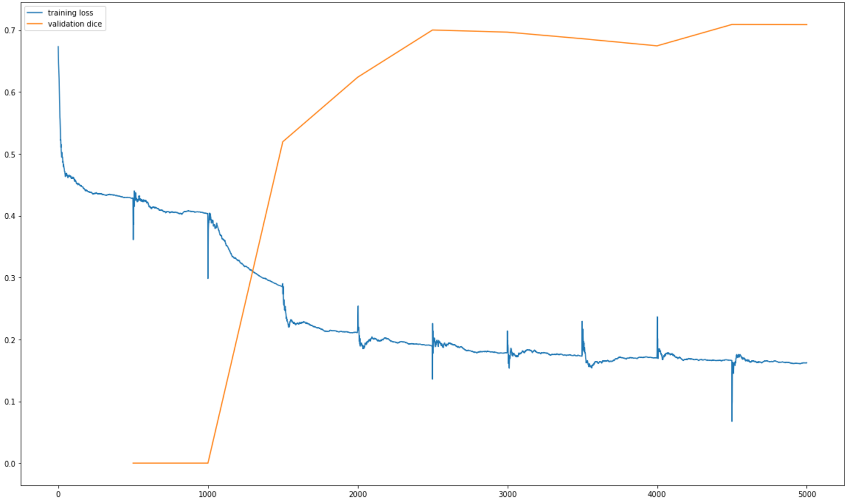
Waarom gebruiken we arrays?

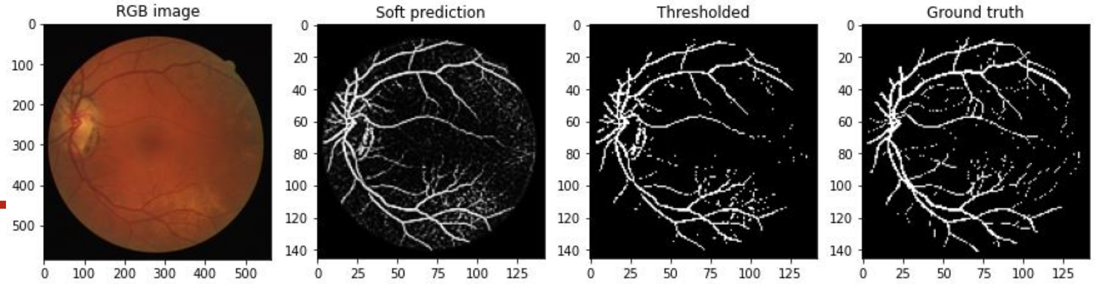
So…why are arrays important exactly?

- Practical benefits
  - They can be easily and efficiently stored in memory
  - They can store huge amounts of combined data without losing structure

- Personal experiences
  - Store statistics of machine learning model in an array and use it later to assess performance
  - Save class characteristics in a dictionary and store all classes together in an array. Then, we can use a simple loop to retrieve all the information
  - Create matrices (mathematical foundation) and use built-in functions to make infinite calculations

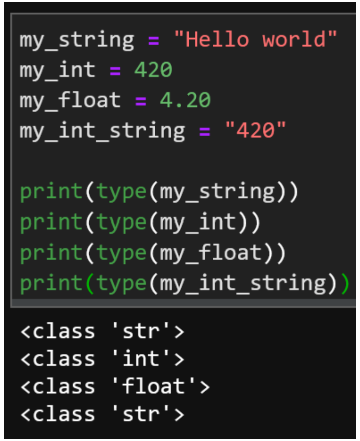
Waarom gebruiken we arrays?





Variable types

- Each variable you assign in your code is assigned a specific type
- For example:
  - A variable with two words “Hello world” is a **string** (of words)
  - A variable with the numbers 420 is an **int**(eger)
  - A variable with decimal numbers like 4.20 is a **float**
  - ...however, when we place quotes between “420”, it becomes a string



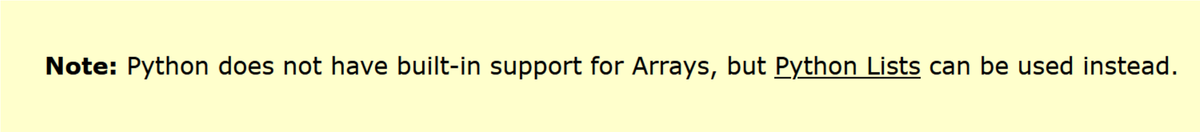
```
my_string = "Hello world"
my_int = 420
my_float = 4.20
my_int_string = "420"

print(type(my_string))
print(type(my_int))
print(type(my_float))
print(type(my_int_string))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'str'>
```

Variable types

- In languages like C#, you can directly declare the type of your variable
  - For example: `string my_string = ”Hello World”`
- However, in Python, you directly assign a value to a variable with an unknown type
  - `my_string = ”Hello World”`
  - Python will automatically recognize the variable type based on the inserted value, this is why *“Hello World”* is considered a string, *420* an integer, and *[ ]* an array
- We can force Python to use a certain variable type by declaring the type during variable assignment
  - For example: `my_int = 420` is automatically considered an integer
  - …but `my_int = str(420)` will be considered a string
  - This is powerful, but make sure that your variable **is allowed** to be assigned a different type. The string *“Hello World”*, for example, cannot be turned into an integer

Array vs list



Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

- We have learnt about the core characteristics of an array and how we can work with different variable types
- Fun fact: Python itself does not support arrays (in its definition)
- In stead of arrays, Python has out-of-the-box support for **lists**
- The difference is subtle:
  - **Lists** can accomodate a finite number of values of **different types**
  - **Arrays** can only store values of **the same type**

NumPy

Probleem met lists

- We could “cheat” an array declaration by only storing values of the same type in the variable
- …but with big lists, this would become obscure and we cannot guarantee that all values in our list are of the same type (*string*, *int*, *float*, etc.)

- Solution: use a library which supports array declarations

NumPy

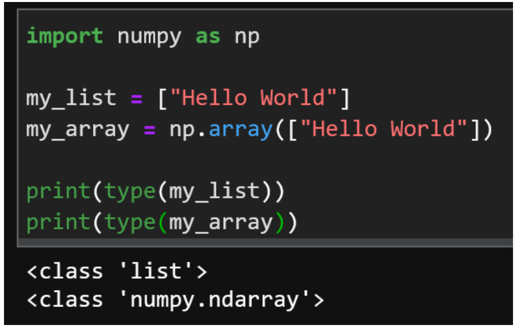
- Introducing…NumPy!
- Open-source
- Sole purpose of the NumPy library is to add Python support for large, multi-dimensional arrays and matrices
- Most recent version to date: 1.20
- Documentation: https://numpy.org/doc/
- #1 tool used by mathematicians, statisticians and analysts to execute multi-dimensional data analysis

NumPy gebruiken

- As always, after installing the NumPy library, we can import it into our current project using `import numpy as np`
  - Adding `as np` behind the import is not necessary, but is used in almost all occassions as it makes using the library in the future much easier (you can simply call `np.<function>` in stead of writing `numpy.<function>`)
- After importing NumPy, creating a NumPy array can be done as follows: `my_array = np.array([])`
  - `my_array` is the name of our variable
  - `np.array()` is the NumPy function in which we indicate that we want to create a new array
  - We use the brackets `[]` to create an **empty** array
    - We could also write `np.array([“Hello World”])` to create an array with a single “Hello World” value of the type string

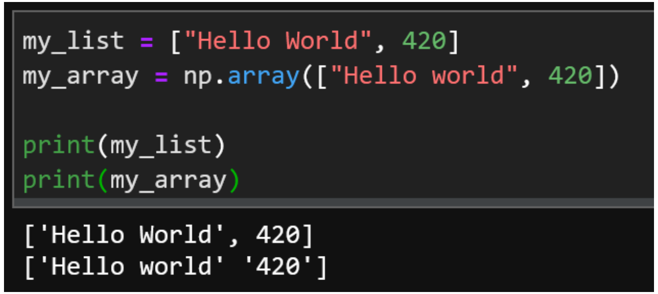
NumPy gebruiken

- Variables defined by the `np.array()` function will always belong to the type `<ndarray>`
  - Just like strings belong to `<str>`, integers to `<int>`, etc.
- …and as speculated before, all types in the array are the same
  - We can see that the `my_array` variable has changed our 420 value into a string. The list did not



```
import numpy as np

my_list = ["Hello World"]
my_array = np.array(["Hello World"])

print(type(my_list))
print(type(my_array))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```



```
my_list = ["Hello World", 420]
my_array = np.array(["Hello world", 420])

print(my_list)
print(my_array)
```

```
['Hello World', 420]
['Hello world' '420']
```

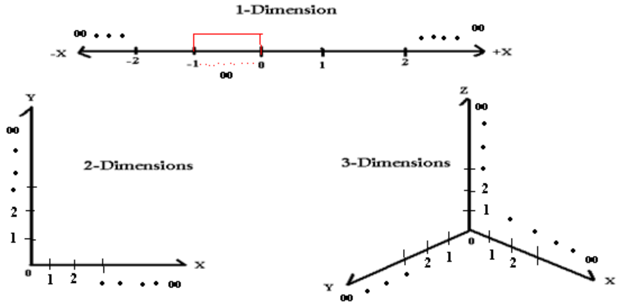
Werken met dimensies

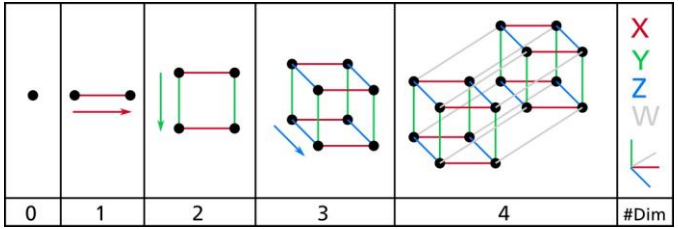
Werken met dimensies

- Working with dimensions is the most powerful tool of any mathematician or analyst
- The default Python lists only accepts one dimension, e.g. a single “line” of values
- …however, numpy allows many more (finite) dimensions

Werken met dimensies

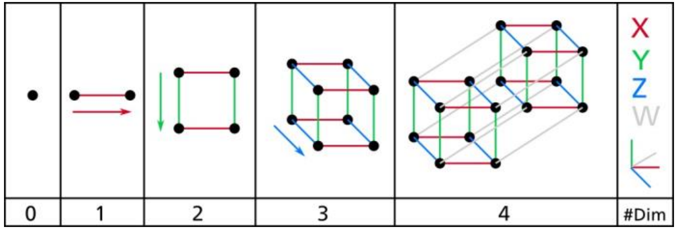
- A 1-dimensional array can be seen as a line which can span infinitely large
- A 2-dimensional array can be visualised as your ‘regular’ graph with an x and y axis
- You can view a 3-dimensional array as the 3D cube you used to draw as a child, basically spanning in 3 dimensions (x, y and z)
- You can go on and on and on

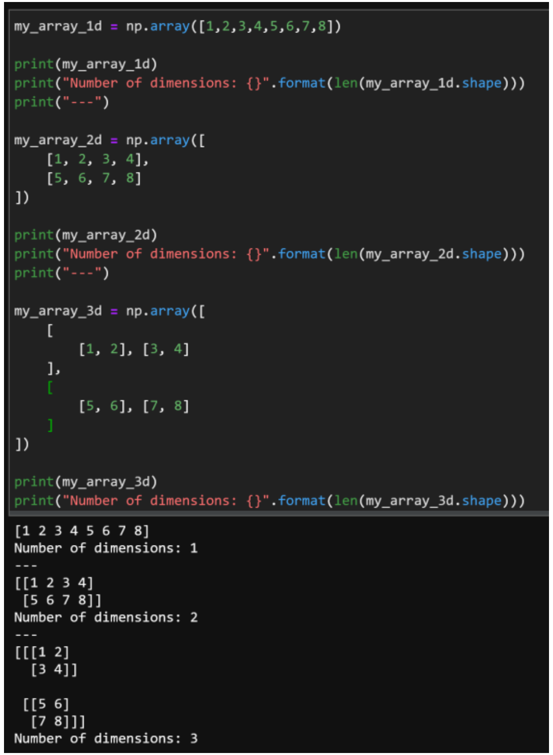




Werken met dimensies

- We cannot really visualize more than 3 dimensions
  - However, we can always create a finite number of dimensions in NumPy!
- In NumPy, each bracket [ ] indicates a new dimension in your array
- Each 2-dimensional array of values



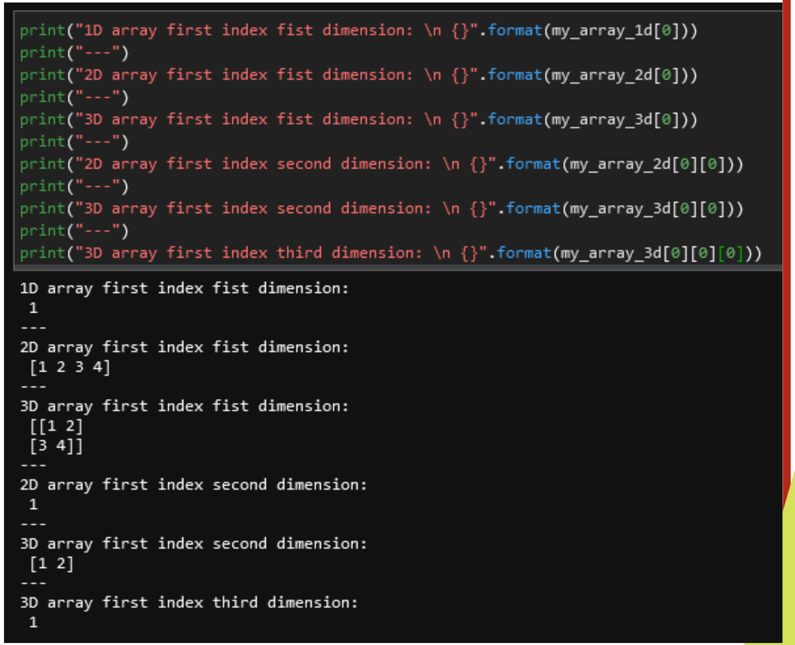


```
my_array_1d = np.array([1,2,3,4,5,6,7,8])

print(my_array_1d)
print("Number of dimensions: {}".format(len(my_array_1d.shape)))
print("---")

my_array_2d = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8]
])

print(my_array_2d)
print("Number of dimensions: {}".format(len(my_array_2d.shape)))
print("---")

my_array_3d = np.array([
    [
        [1, 2], [3, 4]
    ],
    [
        [5, 6], [7, 8]
    ]
])

print(my_array_3d)
print("Number of dimensions: {}".format(len(my_array_3d.shape)))
```

```
[1 2 3 4 5 6 7 8]
Number of dimensions: 1
---
[[1 2 3 4]
 [5 6 7 8]]
Number of dimensions: 2
---
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
Number of dimensions: 3
```

Toegang tot dimensies

- As mentioned before, each bracket `[]` indicates a new dimension in your array
- We can use this theory to print the contents of our array(s) dimension-wise
  - If the first bracket `[]` indicates the first dimension, it means that we can return this first dimension using `variable_name[]`
  - The number that is specified between the brackets `[]` will decide which index of the array is returned (for this example we pick “0” as the first index, but we can also choose “1” or whatever other number until the max index of our array is reached)
- Every bracket we add after our variable will show us the next available dimension, so `variable_name[0]` shows the **first index of the first dimension**, `variable_name[0][0][0]` the **first index of the first index of the second dimension**, etc.

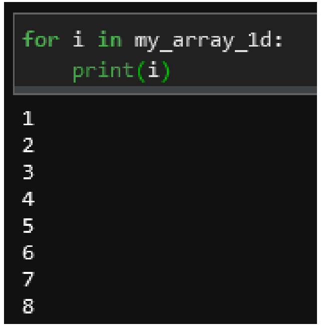


```
print("1D array first index fist dimension: \n {}".format(my_array_1d[0]))
print("---")
print("2D array first index fist dimension: \n {}".format(my_array_2d[0]))
print("---")
print("3D array first index fist dimension: \n {}".format(my_array_3d[0]))
print("---")
print("2D array first index second dimension: \n {}".format(my_array_2d[0][0]))
print("---")
print("3D array first index second dimension: \n {}".format(my_array_3d[0][0]))
print("---")
print("3D array first index third dimension: \n {}".format(my_array_3d[0][0][0]))
```

```
1D array first index fist dimension:
 1
---
2D array first index fist dimension:
 [1 2 3 4]
---
3D array first index fist dimension:
 [[1 2]
 [3 4]]
---
2D array first index second dimension:
 1
---
3D array first index second dimension:
 [1 2]
---
3D array first index third dimension:
 1
```

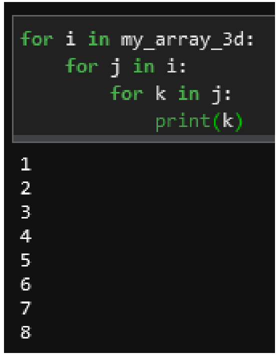
Toegang tot dimensies

- We can also visualize our dimensions in a loop
- Example:
  - Print contents of our 1D array in a single for-loop
  - Print contents of our 3D array in 3 nested for-loops
- Why does this work?
  - Our array has **three dimensions** containing **two values in each dimension**
  - In short, the shape of our array is `(2,2,2)`
  - Each time we start a `for`, we simply *move* to the *next* dimension



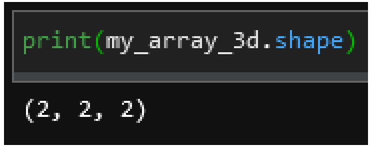
```
for i in my_array_1d:
    print(i)
```

```
1
2
3
4
5
6
7
8
```



```
for i in my_array_3d:
    for j in i:
        for k in j:
            print(k)
```

```
1
2
3
4
5
6
7
8
```



```
print(my_array_3d.shape)
```

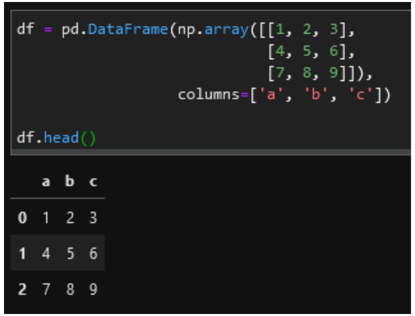
```
(2, 2, 2)
```

Van Pandas naar NumPy

Van Pandas naar NumPy

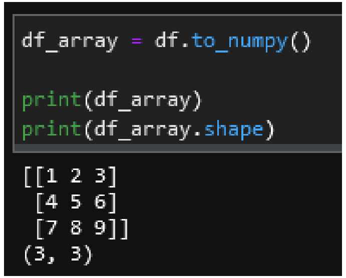
- Pandas is a great tool to visualize and mutate your data set
- …however, there are drawbacks
  - mathematical tools (also used for statistics) are sometimes limited by the Pandas library
  - Pandas dataframes take up lots of space in your memory
- Luckily, in Pandas, you can convert your dataframe into a NumPy array
  - `<df_name>.to_numpy()`

- **Question: how many dimensions does our `df_array` array have? How many values are present in each dimension?**



```
df = pd.DataFrame(np.array([[1, 2, 3],
                            [4, 5, 6],
                            [7, 8, 9]]),
                  columns=['a', 'b', 'c'])

df.head()
```

```
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

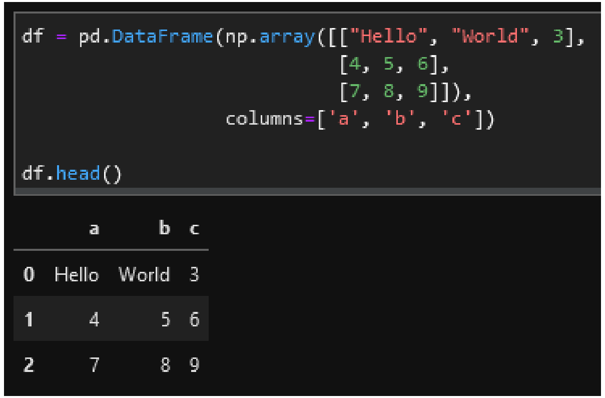


```
df_array = df.to_numpy()

print(df_array)
print(df_array.shape)
```

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

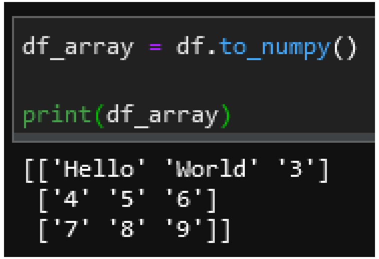
Van Pandas naar NumPy

- Pandas dataframes can have multiple variable types in each column
- …but, we also know that an array only returns values from the same type
- So, what happens when we replace some integers in our `df` variable with strings?
    - Short answer: all values (elements) in the array will be converted to a string
    - Long answer: the strings “Hello” and “World” cannot be converted into an integer (after all, they are not numbers), thus the ‘best’ option for the Pandas dataframe to be accepted as a NumPy array is to turn all values in the array into strings
- Same rule applies to floats, booleans, etc.



```
df = pd.DataFrame(np.array([["Hello", "World", 3],
                            [4, 5, 6],
                            [7, 8, 9]]),
                  columns=['a', 'b', 'c'])

df.head()
```

	a	b	c
0	Hello	World	3
1	4	5	6
2	7	8	9



```
df_array = df.to_numpy()

print(df_array)
```

```
[['Hello' 'World' '3']
 ['4' '5' '6']
 ['7' '8' '9']]
```

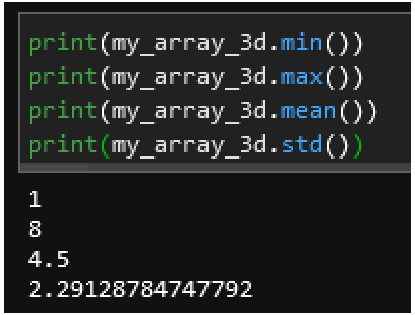
NumPy in-depth

NumPy in-depth

- We went over the basic principles of NumPy and viewed its strengths and weaknesses
- …but it doesn’t stop there
- The NumPy library is *huge*, it can be guaranteed that you won’t use every function throughout your career
- …however, knowledge of some important functions are critical to get you started

Standaard array kenmerken

- `<ndarray>.min()`
  - Returns minimum value out of all values present in array
  - In the string array example from before, this function will actually check all string values which could potentially be an integer and return the miniumum value, which was 3
- `<ndarray>.max()`
  - Returns minimum value out of all values present in array
- `<ndarray>.mean()`
  - Returns mean (center) value from the array
- `<ndarray>.std()`
  - Returns array standard deviation (the mean ‘offset’ between the mean of the array and all the other array values)

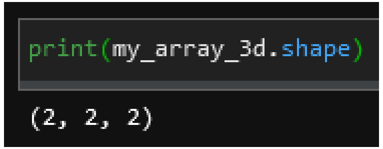


```
print(my_array_3d.min())
print(my_array_3d.max())
print(my_array_3d.mean())
print(my_array_3d.std())
```

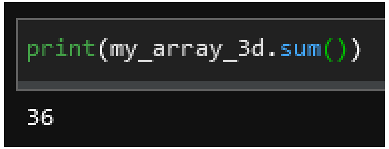
```
1
8
4.5
2.29128784747792
```

Standaard array kenmerken

- `<ndarray>.sum()`
  - Returns the sum of the (integer or float) values in the array
- `<ndarray>.shape`
  - Returns a list of array dimensional information



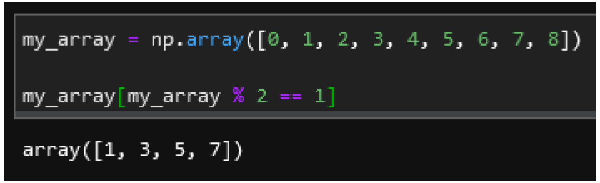
```
print(my_array_3d.shape)
(2, 2, 2)
```



```
print(my_array_3d.sum())
36
```

Waardes verwijderen

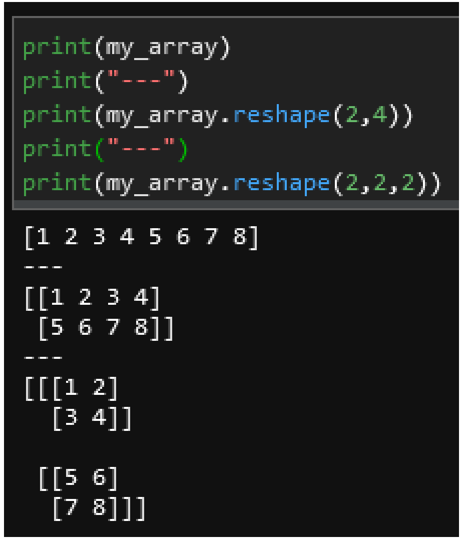
- Suppose you want to remove all even numbers from your array
- We define the brackets `[]` after the variable to call the first dimension of the array
- In stead of returning, for example, index “0”, “1”, etc. from the array, we now want to filter out any even numbers
- We do this by writing `my_array % 2 == 1` inside the brackets
    - `my_array` is used to indicate that we want to use the entire array (not just a single index)
    - The percentage symbol `%`, in mathematics, is called the “modulo”, and boils down to *“what is left if we keep subtracting <value> from our current index”*
    - In our case, the *<value>* is 2, meaning that we filter our `my_array` array by all numbers for which the value minus 2 is not 0 (thus 1)
    - As a result, all that is left are the odd numbers



```
my_array = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])

my_array[my_array % 2 == 1]

array([1, 3, 5, 7])
```

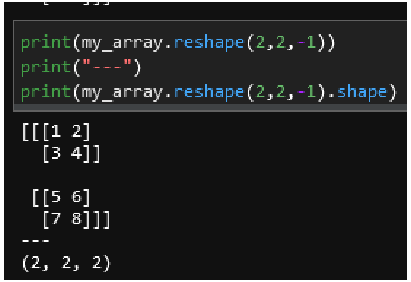
Arrays hervormen

- Suppose you have a 1D array, but you want to turn it into a 2D (or even 3D) array
- Remember the result of the `<ndarray>.shape` function? We can use this information to reshape our array
- `<ndarray>.reshape()`
- Tip: don’t know how big your dimensions should be? Using the value “-1” will automatically determine the size of **one** of your dimensions



```
print(my_array)
print("---")
print(my_array.reshape(2,4))
print("---")
print(my_array.reshape(2,2,2))
```

```
[1 2 3 4 5 6 7 8]
---
[[1 2 3 4]
 [5 6 7 8]]
---
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

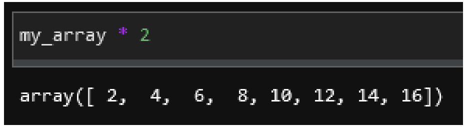


```
print(my_array.reshape(2,2,-1))
print("---")
print(my_array.reshape(2,2,-1).shape)
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
---
(2, 2, 2)
```

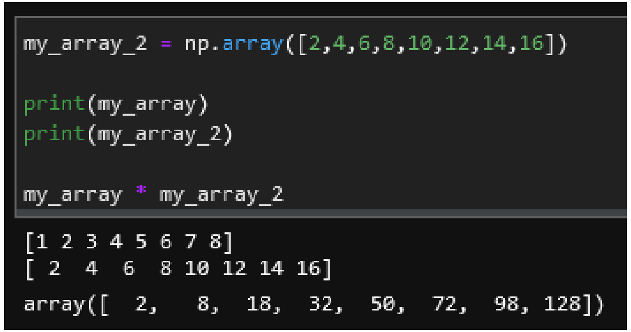
Arrays vermenigvuldigen

- You can simply multiple all of the (integer or float) contents of your array by multiplying it with a constant value
- You can also multiply multiple (1D) arrays with each other
    - Arrays have to be of the same length
    - Values with the same index will be multiplied with eachother



```
my_array * 2
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16])
```

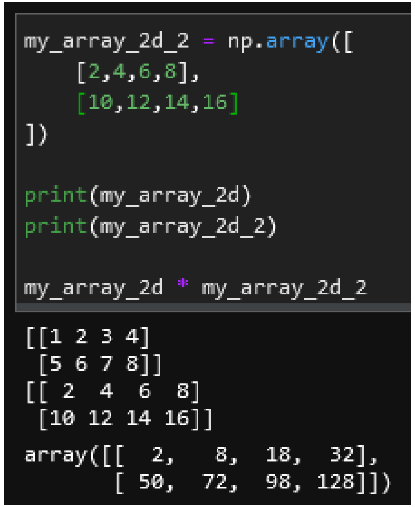


```
my_array_2 = np.array([2,4,6,8,10,12,14,16])

print(my_array)
print(my_array_2)

my_array * my_array_2
```

```
[1 2 3 4 5 6 7 8]
[ 2  4  6  8 10 12 14 16]
array([  2,   8,  18,  32,  50,  72,  98, 128])
```

Arrays vermenigvuldigen

- You can do the same with 2D, 3D, etc. arrays
  - Make sure they have the same dimensions and the same number of values per dimension!
  - For 2D arrays this is called **matrix multiplication**



```
my_array_2d_2 = np.array([
    [2,4,6,8],
    [10,12,14,16]
])

print(my_array_2d)
print(my_array_2d_2)

my_array_2d * my_array_2d_2
```

```
[[1 2 3 4]
 [5 6 7 8]]
[[ 2  4  6  8]
 [10 12 14 16]]
array([[  2,   8,  18,  32],
       [ 50,  72,  98, 128]])
```

Samenvatting

Samenvatting

- Make sure you downloaded NumPy using the Pip package
- NumPy is a powerpull tool for creating and mutating arrays
- Current NumPy documentations: https://numpy.org/doc/1.20/
- Fun site to practice NumPy: https://www.machinelearningplus.com/python/101-numpy-exercises-python/

Vragen?

hogeschool Windesheim

Radboud University

IN·DEI·NOMINE·FELICITER