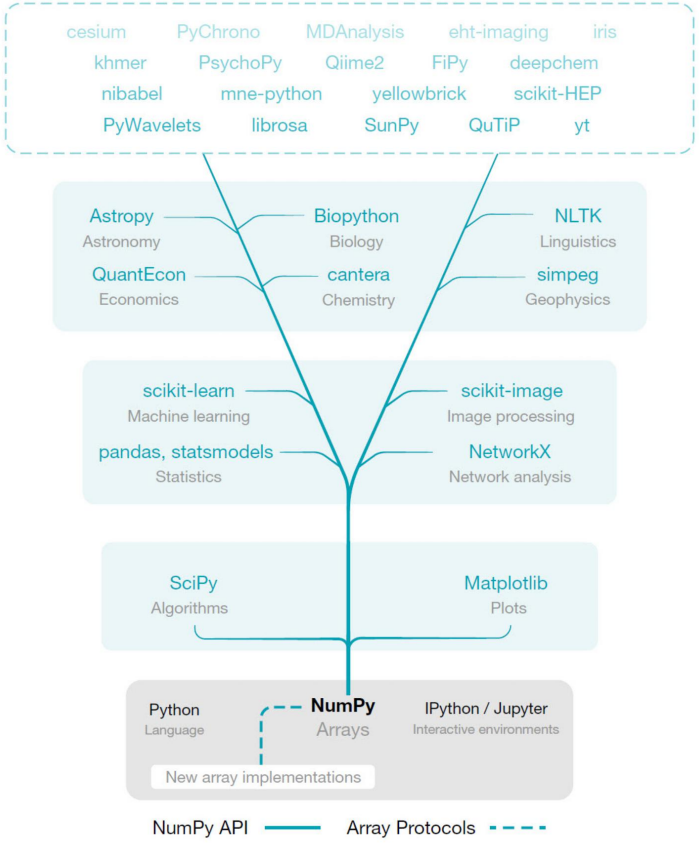
A gentle introduction of `numpy`

Disclaimer: These notes and examples are an adaptation of the references listed at the end. They are compiled to fit the scope of this course.

Introduction

Python, as a general-purpose programming language, is not explicitly designed for numerical computing, statistical modeling, or data science in general. This is where `numpy` comes into the picture. `numpy` (numerical python) is a library providing a multidimensional python array object along with a variety of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, introductory linear algebra, basic statistical operations, random simulation, and much more. `numpy` has a syntax that closely mimics mathematics. Because of its extraordinary speed, easy syntax, powerful capabilities, and seamless integration with python, the number of packages depending on `numpy` increases daily. According to https://libraries.io/, over 32,000 packages depend on `numpy` (Feb 11, 2022).

`numpy` is the foundation of almost every python library that handles numeric computations, such as `scipy`, `Matplotlib`, `pandas`, `keras`, and `scikit-learn`, among others. The following graphic is taken from ***Array programming with NumPy*; Nature, Vol 585, September 2020** beautifully displays `numpy`'s ecosystem. As you can see, most of the data science libraries are rooted in `numpy`.



This handout, we will focus on:

- `numpy` **'s n-dimensional array data structure** `ndarray` **:** `numpy` 's array is optimized for homogeneous numeric data that is accessed via integer indices.
- `ndarray` **'s most commonly used methods (functions) and attributes:** These functions implement efficient looping over `numpy` arrays. They can be used for:

- **Array manipulation & scientific computation:** Creating, reshaping, concatenating, and padding arrays; searching, sorting and counting data in arrays.
- **Computing elementary statistics:** such as the mean, median, variance, and standard deviation, among others.
- **Linear algebra:** Solving linear systems of equations, calculating various matrix properties such as the determinant, the norm, the inverse, and the eigenvalue.
- **Random number generation:** Generating (pseudo) random numbers from a variety of distributions.

Installing and importing `numpy`

Installation

`numpy` comes pre-installed with `anaconda` distribution; so there is no need for any installations if use `anaconda`. Sometimes, `anaconda` doesn't have the latest version of the packages; at the time of updating this handout (Jan, 30, 2023), the latest version of `numpy` was `1.24.1`; if you would like to install the newest version, try

```
pip install numpy==1.24.1
```

Note: Depending on your computer configuration and your user's privileges, you may need to try the following if you get a user privilege-related error

```
pip install --user numpy==1.24.1
```

Importing `numpy`

`numpy` documentation (about 2,000 pages) available at https://numpy.org/doc/ suggests importing `numpy` using

```
import numpy as np
```

Now, you can access all `numpy` capabilities using `np.`

`numpy`'s `ndarray` data structure

`numpy`'s fundamental n-dimensional data structure is called `ndarray`. Moving forward, we will refer to `ndarray` as *array*. `numpy` offers seven general mechanisms for creating arrays detailed at https://numpy.org/doc/stable/reference/routines.array-creation.html. This handout will cover four of the most commonly used mechanisms listed below:

- Conversion from other python structures (e.g., lists, tuples)
- Using `numpy`'s array creation objects (e.g., `arange`, `ones`, `zeros`, etc.)
- Use of special library functions such as `random`
- reading from a file

1-D and 2-D arrays

One-dimensional array is like one row or column in your Excel file with one or more elements

element_index_0	element_index_1	element_index_2	element_index_3	element_index_4

or

element_index_0
element_index_1
element_index_2
element_index_3
element_index_4

As you can see this is very similar to the structure of python's `list` class.

A two-dimensional (or multi-dimensional) array has more than one row or column. A two-dimensional array is like a typical Excel table.

element_index_0_0	element_index_0_1	element_index_0_2	element_index_0_3	element_index_0_4
element_index_1_0	element_index_1_1	element_index_1_2	element_index_1_3	element_index_1_4
element_index_2_0	element_index_2_1	element_index_2_2	element_index_2_3	element_index_2_4

element_index_0_0	element_index_0_1	element_index_0_2	element_index_0_3	element_index_0_4
element_index_3_0	element_index_3_1	element_index_3_2	element_index_3_3	element_index_3_4
element_index_4_0	element_index_4_1	element_index_4_2	element_index_4_3	element_index_4_4
element_index_5_0	element_index_5_1	element_index_5_2	element_index_5_3	element_index_5_4

Conversion from the other python data types

Numerical data arranged in an array-like structure in python can be converted to arrays through the `array()` function. The most obvious examples are `lists` and `tuples`.

> **creating 1-dimensional arrays from python lists (and tuples)**
>
> ```
> grades_list = [85, 90, 100, 95, 98, 92]  # python built-in list
> array_1 = np.array(grades_list)          # turning a list into a numpy array
> print(array_1)                           # array_1 is a vector
> print(type(array_1))
> ```
>
> yields
>
> ```
> [ 85  90 100  95  98  92]
> <class 'numpy.ndarray'>
> ```
>
> **using list comprehension**
>
> ```
> import numpy as np
>
> array_2 = np.array([i ** 2 for i in range(2, 10) if i % 2 == 0])
> print(array_2)
> ```
>
> yields
>
> ```
> [ 4 16 36 64]
> ```
>
> **STOP:** Try your best to avoid using `range` for creating `numpy` arrays as it is much slower than `arange` which will be covered in the next section.
>
> **creating a 2-dimensional array (matrix) from python lists (and tuples)**

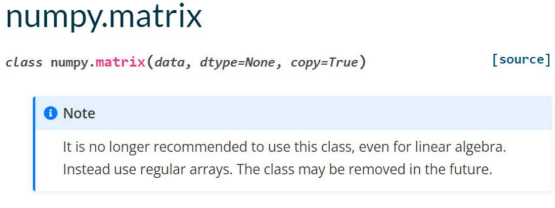
```
row_1 = [85, 90, 100]              # python built-in list
row_2 = [95, 98, 92]               # python built-in list
array_3 = np.array([row_1, row_2])  # turning lists into a 2-D numpy array (matrix)

print(array_3)                     # array_3 is a matrix
print(type(array_3))
```

yields a matrix with two rows and three columns as in

```
[[ 85  90 100]
 [ 95  98  92]]
<class 'numpy.ndarray'>
```

STOP: numpy has a `matrix` class for creating matrices (2-dimensional arrays). However, according to `numpy` 's official documentation (partial image below), its use is no longer recommended



Using `numpy` 's array creation functions

`numpy` has built-in functions for creating arrays from scratch. One of the most widely used functions is `arange()` function. This function could be used just like python's built-in `range()` function with one, two, or three arguments.

Using `arange`

```
range_1 = np.arange(8)         # integer numbers 0 to 7
range_2 = np.arange(3.1, 8.4)  # integer arguments not required; but recommended
range_3 = np.arange(3, 8, 2)   # integer numbers between 3 to 7 with step size of 2

print(f"range_1 is {range_1}")
print(f"range_2 is {range_2}")
print(f"range_3 is {range_3}")
```

> yields
>
> ```
> range_1 is  [0 1 2 3 4 5 6 7]
> range_2 is  [3.1 4.1 5.1 6.1 7.1 8.1]
> range_3 is  [3 5 7]
> ```
>
> **Observation 1:** Just like `range` function, `arange` can accept one, two, or three arguments. Either way, the lower limit is included and the upper limit is excluded.
>
> **Observation 2:** `arange` can also accept float arguments. This is one of the main differences between `range` and `arange`.

STOP: Avoid using `range()` to build `numpy` arrays; it is a few order of magnitude slower than `arange()`!

The following table summarizes some of the most widely used functions for array creation.

numpy function	Common syntax	Notes	Explanation
zeros	np.zeros(shape, dtype)	**dtype**: optional. default: **float**	array of 0s
ones	np.ones(shape, dtype)	**dtype**: optional. default: **float**	array of 1s
full	np.full(shape, fill_value, dtype)	**dtype**: optional. default: **float**	array of a given number **fill_value**
eye	np.eye(N, dtype)	**dtype**: optional. default: **float**	return a N-by-N 2-D array with **ones** on the diagonal and **zeros** elsewhere
arange	np.arange(start, stop, step, dtype)	**dtype**: optional. default: **int**	Return evenly spaced values within a given interval. Values are generated within the half-open interval [**start, stop**) meaning the interval including **start** but excluding **stop**. Note: when using a non-integer **step**, the results will often not be consistent. It is better to use **numpy.linspace()** for these cases.
linspace	np.linspace(start, stop, num, endpoint, retstep, dtype)	**num**: optional. default value = **50** **endpoint**: optional. default: **True** meaning **stop** included **retstep**: optional. default: **False**. if **True** returns (**sample**, **step**) where **step** is the	Returns **num** evenly spaced samples, calculated over the interval [**start**, **stop**]

numpy function	Common syntax	Notes	Explanation
		spacing between samples **dtype**: optional.	

Note 1: The above-provided syntax is the commonly used syntax; please consult with `numpy`'s documentation available at https://numpy.org/doc/stable/reference/routines.array-creation.html for the complete syntax of these functions.

Note 2: If there is no default `dtype`, the output data type is inferred from the other input arguments.

The following will demonstrate these functions in action:

> **Example**
>
> ```
> array_of_zeros = np.zeros(4, dtype = int)
> print(array_of_zeros)
> ```
>
> yields an array of 1 row and 4 columns with all elements equal to 0 (`int` type); as in
>
> ```
> [0 0 0 0]
> ```
>
> **Example**
>
> ```
> array_of_zeros = np.zeros((2, 3), dtype = int)
> print(array_of_zeros)
> ```
>
> yields a 2-dimensional array of 2 rows and 3 columns with all elements equal to 0 (`int` type); as in
>
> ```
> [[0 0 0]
> [0 0 0]]
> ```
>
> **Example**
>
> ```
> array_of_ones = np.ones((1, 2))
> print(array_of_ones)
> ```
>
> yields a 2-dimensional array of 1 row and 2 columns with all elements equal to 1 (default `float` type); as in
>
> ```
> [[1. 1.]]
> ```
>
> **Example**

```
array_of_ones = np.ones((3, 4))
print(array_of_ones)
```

yields a matrix of 3 rows and 4 columns with all elements equal to 1 (default `float` type); as in

```
[[1. 1. 1. 1.]
[1. 1. 1. 1.]
[1. 1. 1. 1.]]
```

Example

```
array_of_sevens = np.full((3,2), 7)
print(array_of_sevens)
```

yields

```
[[7 7]
[7 7]
[7 7]]
```

Example

```
eye_array = np.eye(4, dtype = float)
print(eye_array)
```

yields

```
[[1. 0. 0. 0.]
[0. 1. 0. 0.]
[0. 0. 1. 0.]
[0. 0. 0. 1.]]
```

Example

```
arange_1 = np.arange(1,6,2)
print(arange_1)
```

yields

```
[1 3 5]
```

Example

```
lin_space_1 = np.linspace(1, 10)
print(lin_space_1)
```

yields an array of 50 values evenly spaced between 1 and 10, including 10

```
[ 1.          1.18367347  1.36734694  1.55102041  1.73469388  1.91836735
 2.10204082  2.28571429  2.46938776  2.65306122  2.83673469  3.02040816
 3.20408163  3.3877551   3.57142857  3.75510204  3.93877551  4.12244898
 4.30612245  4.48979592  4.67346939  4.85714286  5.04081633  5.2244898
 5.40816327  5.59183673  5.7755102   5.95918367  6.14285714  6.32653061
 6.51020408  6.69387755  6.87755102  7.06122449  7.24489796  7.42857143
 7.6122449   7.79591837  7.97959184  8.16326531  8.34693878  8.53061224
 8.71428571  8.89795918  9.08163265  9.26530612  9.44897959  9.63265306
 9.81632653 10.        ]
```

Example

```
lin_space_2 = np.linspace(2, 5, 6)
print(lin_space_2)
```

yields an array of 6 values evenly spaced between 2 and 5, including 5

```
[2.  2.6 3.2 3.8 4.4 5. ]
```

Example

```
lin_space_3 = np.linspace(1, 3, 5, endpoint = False)
print(lin_space_3)
```

yields an array of 5 values evenly spaced between 1 and 3, excluding 3

```
[1.  1.4 1.8 2.2 2.6]
```

Example

```
lin_space_4 = np.linspace(2, 11, 5, endpoint = True, retstep = True)
print(lin_space_4)
```

yields a `tuple`. The first element of the tuple is the array of 5 evenly spaced values between 2 and 11 (including 11); the second element is the space between consecutive values in the array; as in

```
(array([ 2.  ,  4.25,  6.5 ,  8.75, 11.  ]), 2.25)
```

> `lin_space_4[0]` will return the `array` of `[ 2.  ,  4.25,  6.5 ,  8.75, 11.  ]` and `lin_space_4[1]` will return the step size of `2.25`. Alternatively, you can unpack the tuple using
>
> ```
> arr, step_size = lin_space_4
> print(f'array is {arr}')
> print(f'step size is {step_size}')
> ```
>
> yields
>
> ```
> array is  [ 2.    4.25  6.5   8.75 11.  ]
> step size is  2.25
> ```

Note 1: `np.linspace` is a convenient way of creating a list of numbers with an underlying pattern. As an example, for integer `a` and `b` (`b > a`), `np.linspace(a, b, b - a + 1)` will generate a list of integer numbers between `a` and `b`, including `a` and `b`. I am sure you have noticed that in this context: `np.linspace(a, b, b - a + 1)` is equivalent to `np.arange(a, b + 1)`

Note 2: Besides `np.linspace`, there is `np.logscale` for generating evenly-spaced numbers on a log scale. Please refer to `numpy`'s documentation for more details.

Array creation using `random` module functions of `numpy` library

Another common way of creating an array in `numpy` is using the `random` module. This module offers variety of functions that can be used for generating random numbers from different distributions. These functions are accessible via `np.random`. Here is a couple of quick examples

> ```
> rnd_1 = np.random.random(3)
> print(f'A vector of {len(rnd_1)} random numbers from u[0, 1]:\n{rnd_1}\n')
>
> rnd_2 = np.random.normal(loc = 10, scale = 2.5, size = (2,3))
> print(f'A 2x3 matrix of N(mu = 10, sigma = 2.5) random numbers:\n{rnd_2}\n')
> ```
>
> yields
>
> ```
> a vector of 3 random numbers from u[0, 1]:
> [0.1471569  0.029647   0.59389349]
>
> a 2x3 matrix of N(mu = 10, sigma = 2.5) random numbers:
> [[ 7.4303736   8.53820447 12.04148482]
>  [ 9.79513237  9.13808496 11.32072036]]
> ```

A complete list of these functions is available at https://numpy.org/doc/1.16/reference/routines.random.html. The following table summarizes the most widely used functions in this module.

numpy function	Most commonly used syntax	Notes	Explanation	Output type
random	random(size)	**size**: optional. Output array size default: **None**	Generates an array of (uniform) random number(s) between 0.0 and 1.0 with given shape **size**. **This is the preferred function for generating uniform random numbers.**	continuous
uniform	uniform(low, high, size)	**low**: optional. default value 0.0 **high**: optional. default value 1.0 **size**: optional. Output array size. default: **None**	Generates an array of (uniform) random number(s) between low and high with given shape size	continuous
randint	randint(low, high, size, dtype)	**low**: int **high**: optional int. If **high = None**, then **high = low + 1** **Note: highly recommended to specify values of low and high** **size**: optional. Output array size. **dtype**: optional. default: **int**	Return random integers from **low** (inclusive) to **high** (exclusive).	discrete
choice	np.random.choice(a, size=None, replace=True, p=None)	**a**: 1-D array of values or **int**. if 1-D array: random sample is generated from its elements. If an int: random sample is generated from **np.arange(a)** **size**: optional. Output array size **replace**: optional Boolean. default: **True**. if **True**: sampling with replacement. if **False**: sampling without replacement **p**: 1-D array. probabilities associated with each entry of **a**. If not given, uniform distribution over all entries in **a**	Generates a random sample from a given 1-D array	discrete

numpy function	Most commonly used syntax	Notes	Explanation	Output type
shuffle	np.random.shuffle(a)	**a**: 1-D array	Modifies the sequence a by shuffling its contents. **warning**: this is an in-place shuffling; meaning it modifies the original array **a**	discrete
randn	randn(d0, d1, …, dn)	**d0, d1, …, dn**: positive integers, optional. dimensions of the returned array	Return a sample (or samples) from the standard normal distribution (mean of zero and standard deviation of one)	continuous
normal	normal(loc, scale, size)	**loc**: distribution average **scale**: distribution standard deviation **size**: optional output shape	Return a sample (or samples) from the normal distribution with given mean and standard deviation	continuous
lognormal	lognormal(mean, sigma, size)	**mean**: Mean value of the underlying normal distribution. Default is **0** **sigma**: Standard deviation of the underlying normal distribution. **size**: optional output shape	Draw samples from a log-normal distribution.	continuous

Note 1: `numpy` offers several functions such as `random`, `random_sample`, `ranf`, `rand`, and `sample` for generating uniformly distributed random numbers between `0` and `1`. It is a common practice to use `np.random.random()` for this purpose. Please do your best to stay with `np.random.random()`.

Note 2: Sometimes, there is a need for generating reproducible random numbers. This can be done using `np.random.seed` function. As an example, the following snippet will generate same random numbers every time you run it

```
np.random.seed(1234)
np.random.random(3)
```

Now try to run the following several times and see if you get the same results!

```
np.random.seed(2022)
np.random.random(3)
```

Note 3: Please consider passing size argument to these functions. Some accept the size inside a set of `()`; whereas some don't accept the additional set of `()`. Here is an example:

> ```
> rnd_1 = np.random.random((2,3))
> rnd_2 = np.random.randn(2,3)
> ```
>
> will run with no problem; however,
>
> ```
> rnd_1 = np.random.random(2,3)
> rnd_2 = np.random.randn((2,3))
> ```
>
> will result in `TypeError`!

Note 4: New versions of `numpy` suggest using `default_rng()` class for initializing random number generation. Please refer to https://numpy.org/doc/stable/reference/random/generator.html for more details.

File operations with `numpy` arrays

So far we have seen several mechanisms of generating `numpy` arrays. This section will explore writing an array to a `csv` file and reading arrays from a `csv` file. We will learn much more about `csv` (comma-separated values) files later when we discuss `pandas` library. For the time being, it suffices to know that `csv` files may contain *numerical* and *text values* separated by a comma `,`; hence, the name `comma-separated values`. These files can be opened using common text editors such as *MS Word* and *Notepad* or spreadsheet applications such as *MS Excel*. `savetxt` and `loadtxt` functions enable us to work with files through `numpy`. Syntax of these functions are quite elaborate and can be found at https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html and https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html. In the following, we will see the most commonly used syntax and application.

Writing a `numpy` array to a `csv` file

`savetxt` function can be used to save a 1-D or 2-D array into a file.

> **Example: 1-D array**
>
> ```
> array_1 = np.arange(5,16, 2)
> np.savetxt( 'try_1.csv', array_1, fmt="%d", header = "my_array_1")
> ```

output: this saves the array of 6 elements in the file `try_1.csv`. You can choose other file extensions such as `.out` or `.txt`; however, as we will see later, using `csv` files is very common in data analytics. Any spreadsheet program such as MS Excel can open `.csv` files. If you open this file with a text editor such as MS Word, Notepad, or WordPad, you will see that there is one row for every row of the array and values in each row are separated with a comma `,`.

`fmt` **:** (optional) parameter determines the format of the file's values. The most common values for this parameter are `"%d"` for integer values, `"%f"` for floating point values.

`header` **:** (optional) argument determines the text that will be written at the beginning of the file.

Example: 2-D array

```
array_2 = np.random.random(size = (2,3))
np.savetxt('try_2.csv', array_2, delimiter = ',', fmt="%.4f", header = "A, B, C")
```

output: this saves the 2 by 3 array of 6 random numbers in the file `try_2.csv`.

`delimiter` parameter is the string or character that separates columns. It is highly recommended to use `,`.

`fmt` parameter determines float numbers with 4 decimal points. One can control the precision using `"%.[precision]f"` notation. For example `"%.3f"` stores values in floating point format with three decimal points.

`header` parameter determines three column headers for three columns of the data. Values of column headers are separated by a comma `,`.

Example: Multiple 1-D arrays of the same size in one file

```
array_3 = np.arange(22, 26)
array_4 = np.arange(4)
array_5 = np.arange(10, 18, 2)

np.savetxt("try_3.csv",(array_3, array_4, array_5),delimiter=",",fmt="%d",header="A, B, C,
D")
```

output: this saves 3 arrays of 4 numbers in the file `try_3.csv`.

`delimiter` parameter is the string or character that separates columns. It is highly recommended to use `,`.

`fmt` parameter determines integer numbers.

`header` determines column headers separated by a comma `,`.

Reading a saved `numpy` array from a file

Saved arrays in a file can be loaded into a `numpy` array using `loadtxt` function.

> **Example**
>
> ```
> array_A = np.loadtxt("try_1.csv", delimiter="," )
> ```
>
> reads the data from `try_1.csv` file and stores it into `array_A` array.
>
> You can also load `numpy` array using `genfromtxt` function. Examples are available at https://numpy.org/doc/stable/reference/generated/numpy.genfromtxt.html. `genfromtext` function can handle missing values as specified.

Homogeneous data in `ndarray` & structured arrays

Let's start with an example

> ```
> arr_1 = np.array([1, 2, 3.14])
> arr_2 = np.array([1, 2.5, '3.14'])
>
> print(f'arr_1 is \n{arr_1}\n')
> print(f'arr_2 is \n{arr_2}\n')
> ```
>
> yields
>
> ```
> arr_1 is
> [1.   2.   3.14]
>
> arr_2 is
> ['1' '2.5' '3.14']
> ```
>
> Surprised? You shouldn't be!

All the elements stored in `ndarray` are required to be of the same type; this implies that the `ndarray` is a block of homogeneous data. So

- if your `ndarray` contains a `string`, `numpy` will turn every element into a `string`.
- if your `ndarray` contains a `float` (and no `string`), `numpy` will turn every element into a `float`.

Let's continue with a follow-up example:

```
arr_3 = np.array([False, True, True])
arr_4 = np.array([1, False, 3.14])
arr_5 = np.array([25, True, 3.14])
arr_6 = np.array(['25', True, 3.14])

print(f'arr_3 is \n{arr_3}\n')
print(f'arr_4 is \n{arr_4}\n')
print(f'arr_5 is \n{arr_5}\n')
print(f'arr_6 is \n{arr_6}\n')
```

yields

```
arr_3 is
[False  True  True]

arr_4 is
[1.   0.   3.14]

arr_5 is
[25.    1.    3.14]

arr_6 is
['25'    'True'    '3.14']
```

In order to maintain a homogeneous array:

`arr_3` is a homogeneous array of `bool`; so `numpy` keeps it as is.

in `arr_4`, `numpy` turns `False` to `0.` and in `arr_5` `True` turns to `1.`

in `arr_6` all elements turn to a `string` because there is a `string` in the array.

Final comment: For the most part, we will use `numpy` for handling numerical types (often `float`); however, you should be mindful of storing mixed data types in a `numpy` ndarray and try to avoid it at all costs. (`int` and `float` together in an array is fine!). **However...**

If you insist on having a mix of data types in a `numpy` ndarray, please refer to the `Structured arrays` part of `numpy`'s documentation at https://numpy.org/doc/stable/user/basics.rec.html. Although `numpy`'s `structured array` is a bit faster than `pandas` `DataFrame`, I strongly recommend you to use `DataFrame` for storing and working with mixed data types.

I will end this part with one example (based on `numpy`'s documentation); I am hoping that this example will convince you to stay away from `numpy`'s `Structured arrays` and develop affection and appreciation for `pandas`'s `DataFrame` that we will learn soon! Here is an example:

```
dogs = np.array([('Milo', 8, 85.5), ('Fluffy', 3, 16.4)],
         dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f8')])

print(f"my dogs are \n{dogs}\n")
print(f"first dog's info is \n{dogs[0]}\n")
print(f"{dogs[1]['name']}'s weight is \n{dogs[1]['weight']:.2f}\n")
```

yields

```
my dogs are
[('Milo', 8, 85.5) ('Fluffy', 3, 16.4)]

first dog's info is
('Milo', 8, 85.5)

Fluffy's weight is
16.40
```

in this example, we dictated data type of each element: `'U10'` for `'name'`, `'i4'` for `age`, `'f8'` for `'weight'`. `U10` means a `Unicode string with a maximum length of 10`, `i4` means an `integer` number that can be stored in 4 bytes (32 bits) of the memory, and `f8` means a `float` number that can be stored in 8 bytes (64 bits) of the memory.

Just for fun, try `U3` instead of `U10` and see what happens!

Optional (and unnecessary for this course) reference on more granular data types used in numpy The website https://betterprogramming.pub/a-comprehensive-guide-to-numpy-data-types-8f62cb57ea83 does a wonderful job explaining data types in `numpy`.

Take away: Use homogeneous data types in `numpy` arrays. For mixed data types use `pandas` `DataFrame` and move on!

`numpy` array attributes (properties)

You can learn more about structure and contents of an array by calling its attributes (properties). The following table summarizes the most commonly used array attributes:

Attribute	Description
ndim	Returns the number of array dimensions
shape	Returns a **tuple** consisting of array dimensions. It can also be used to resize the array.
size	Returns total number of elements in the array
dtype	Returns an array's element type
flat	Displays a one-dimensional view of the array

Example

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(f"arr ndim is  {arr.ndim}")
print(f"arr shape is {arr.shape}")
print(f"arr size is  {arr.size}")
print(f"arr dtype is {arr.dtype}")

print('arr values are: ', end = " ")
for item in arr.flat:
    print(item, end = " ")
```

yields

```
arr ndim is  2
arr shape is (3, 4)
arr size is  12
arr dtype is int32
arr values are:  1 2 3 4 5 6 7 8 9 10 11 12
```

explanation:

- `ndim` is the number of dimensions which is 2 (row is one dimension and column is another).
- `shape` is a tuple with the number of elements in each dimension (3 rows and 4 columns).
- `size` is the total number of entries in this array which is 12.

- `dtype` is the data type of the entries which is integer.
- `flat` collapses the 2-D array into a 1-D array.

Note about `int32` : majority of `numpy` is written in C language and uses C's data types. That is why you may see `int32` , `int64` , `float64` as data types in `numpy` . According to `numpy` documentation: >

> `numpy` supports a much greater variety of numerical types than Python. The primitive types supported are tied closely to those in C.

You can learn more about them at https://numpy.org/doc/stable/user/basics.types.html

Reshaping an array

In the above section, we learned that an array's shape is the number of elements in each dimension. *Reshaping* means changing the shape of an array. `numpy` offers several functions for reshaping an array. Each performs a particular task. The following table summarizes these functions:

function	Commonly used syntax	Description	Note
reshape	new_array = old_array.reshape(new_shape)	**new_array** is the **old_array** reshaped in the **new_shape**.	Doesn't change old_array
flatten	new_array = old_array.flatten()	**new_array** is a copy of the **old_array** collapsed into one dimension.	Doesn't change old_array
ravel	new_array = old_array.ravel()	Return a flattened array.	Doesn't change old_array
resize	array.resize(new_shape)	changes the array dimensions to **new_shape** dimensions. **new_shape** is either an integer or a tuple	Changes the original array dimensions

Note: In the above table, the `new_array` size must be the same as the `old_array` size.

Example

```
old_arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(f"old_arr is \n{old_arr}\n")
```

```
new_arr_1 = old_arr.reshape(2, 6)
print(f"(using reshape): new_arr_1 is \n{new_arr_1}\n")

new_arr_2 = old_arr.flatten()
print(f"(using flatten): new_arr_2 is \n{new_arr_2}\n")

new_arr_3 = old_arr.ravel()
print(f"(using ravel): new_arr_3 is \n{new_arr_3}\n")

old_arr.resize(6,2)
print(f"(using resize): resized old_arr is\n{old_arr}\n")
```

yields

```
old_arr is
[[ 1  2  3  4]
[ 5  6  7  8]
[ 9 10 11 12]]

(using reshape): new_arr_1 is
[[ 1  2  3  4  5  6]
[ 7  8  9 10 11 12]]

(using flatten): new_arr_2 is
[ 1  2  3  4  5  6  7  8  9 10 11 12]

(using ravel): new_arr_3 is
[ 1  2  3  4  5  6  7  8  9 10 11 12]

(using resize): resized old_arr is
[[ 1  2]
[ 3  4]
[ 5  6]
[ 7  8]
[ 9 10]
[11 12]]
```

Note `np.ravel` and `np.flatten` may look the same. Still, they are quite different! `np.flatten` returns a copy of the original array and by changing one array the other one wouldn't change. `np.ravel`, on the other hand, returns a `view` of the array. If you modify the array returned by ravel, it will also modify the entries in the original array. The following example shows the difference:

```
import numpy as np
arr = np.arange(6).reshape(2, 3)
print(f'original arr is \n{arr}\n')

arr_ravel = arr.ravel()
arr_flatten = arr.flatten()

arr[1,2] = 5000  # modify arr
print(f'modified arr is \n{arr}\n')
print(f'arr_ravel after modifying original arr is \n{arr_ravel}\n')
print(f'arr_flatten after modifying original arr is \n{arr_flatten}\n')
```

yields

```
original arr is
[[0 1 2]
[3 4 5]]

modified arr is
[[   0    1    2]
[   3    4 5000]]

arr_ravel after modifying original arr is
[   0    1    2    3    4 5000]

arr_flatten after modifying original arr is
[0 1 2 3 4 5]
```

Transposing an array

Transposing an array is a special form of changing the array shape. Transposing is flipping the array so that rows become the columns and columns become the rows. `.T` attribute returns a transposed *view* (shallow copy) of the original array; in other words, `.T` doesn't change the original array.

Example

```
import numpy as np

my_array = np.arange(6).reshape(2, 3)
print("my_array is")
print(my_array)
```

```
print()                    # print an empty line for improved output readability
my_array_trans = my_array.T
print("my_array transpose is")
print(my_array_trans)
```

yields

```
my_array is
[[0 1 2]
 [3 4 5]]

my_array transpose is
[[0 3]
 [1 4]
 [2 5]]
```

Stacking (combining) arrays

`hstack` and `vstack` functions can join multiple arrays of the same dimension horizontally or vertically, respectively. Here is an example

Example

```
import numpy as np

arr_1 = np.array([[10, 20, 30], [40, 50, 60]])
arr_2 = np.array([[70, 80, 90], [100, 110, 120]])
arr_3 = np.array([[130, 140, 150], [160, 170, 180]])

print("vertical stacking of arr_1 and arr_2 is ")
print(np.vstack((arr_1, arr_2)))
print()

print("horizontal stacking of arr_1 and arr_2 is ")
print(np.hstack((arr_1, arr_2)))
print()
```

yields

```
vertical stacking of arr_1 and arr_2 is
[[ 10  20  30]
 [ 40  50  60]
```

```
 [ 70  80  90]
 [100 110 120]]

horizontal stacking of arr_1 and arr_2 is
[[ 10  20  30  70  80  90]
 [ 40  50  60 100 110 120]]
```

Note 1: Both `hstack` and `vstack` functions accept arrays inside parentheses `()`.

Note 2: You can stack more than two arrays simultaneously. Try `np.vstack((arr_1,arr_2,arr_3))`

Accessing array elements

So far, our discussions were around creating arrays and changing their shape. In this section, we will learn how we can access array elements.

How **not to** access `numpy` array elements

Accessing array elements using `for` loop is always an option. Here is an example:

Example

```
arr_1 = np.arange(12).reshape(3, 4)
for row in arr_1:
    for item in row:
        print(item, end=" ")
```

yields

```
0 1 2 3 4 5 6 7 8 9 10 11
```

However, `numpy` has numerous fast and concise array manipulation methods. So avoid using `for` loop at all costs; unless advised otherwise. In the remaining, we will learn the basics of these methods.

Accessing 1-D array elements

You can access an array element by referring to its (zero-based) index number, meaning that the first element has index 0, and the second has index 1, etc. Here is an example:

Example

```
my_array = np.arange(2, 5)
print(my_array[1])
```

yields

```
3
```

Accessing 2-D array elements

2-D array elements are accessible via two (zero-based) indices; separated by a comma `,`; the first index is for the row and the second one is for the column.

Example

```
arr = np.arange(0, 10).reshape(2, 5)

print(f"arr is \n{arr}\n")
print(f"element is 2nd row and 4th column is {arr[1, 3]}")
print(f"element is the last row and last column is {arr[-1, -1]}")
```

yields

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

element is 2nd row and 4th column is  8
element is the last row and the last column is  9
```

Note 1: For higher dimension arrays, accessing array elements is similar.

Note 2: Just like indexing for lists and tuples, you can also use negative indices with `numpy` arrays.

Slicing `numpy` arrays

Very similar to lists and tuples, one can slice `numpy` arrays; here is an example

Example

```
arr = np.arange(20).reshape(4, 5)

print(f"original array arr is \n{arr}\n")

print(f"second row is \n{arr[1]}\n")

print(f"element in 3rd row and 4th column is \n{arr[2, 3]}\n")

print(f"first three rows are \n{arr[:3]}\n")

# equivalent to arr[0:4, 1]
print(f"2nd column is \n{arr[:, 1]}\n")

# equivalent to arr[1:3, : ]
print(f"2nd and 3rd rows are \n{arr[1:3]}\n")

print(f"2nd and 4th rows are \n{arr[[1, 3]]}\n")

print(f"intersection of 2nd and 3rd rows with 3rd and 4th columns is \n{arr[1:3, 2:4]}\n")

print(f"intersection of first and 3rd rows with 2nd column is \n{arr[[0, 2], 1]}\n")

print(f"intersection of first three rows with column 2 and 4 is \n{arr[:3, [1, 3]]}\n")
```

yields

```
original array arr is
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]

second row is
[5 6 7 8 9]

element in 3rd row and 4th column is
13

first three rows are
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

```
2nd column is
[ 1  6 11 16]

2nd and 3rd rows are
[[ 5  6  7  8  9]
 [10 11 12 13 14]]

2nd and 4th rows are
[[ 5  6  7  8  9]
 [15 16 17 18 19]]

intersection of 2nd and 3rd rows with 3rd and 4th columns is
[[ 7  8]
 [12 13]]

intersection of first and 3rd rows with 2nd column is
[ 1 11]

intersection of first three rows with column 2 and 4 is
[[ 1  3]
 [ 6  8]
 [11 13]]
```

Boolean indexing in `numpy`

One of the powerful capabilities of `numpy` is its boolean selection. Consider

Example

```
arr = np.array([[60, 70, 80], [30, 40, 50]])
print(arr[arr >= 45])
```

yields a one-dimensional array of values that satisfy the condition of ≥ 45 as in

```
[60 70 80 50]
```

Numerical operations on `numpy` arrays

Once you have created the `numpy` arrays, you can do basic operations. `numpy` offers a vast set of tools for array operations. This section will explore the basic operations.

Element-wise operations

All arithmetic operations `+`, `-`, `*`, `**`, `/`, `//`, `%` between an array and a scalar are element-wise. Element-wise operations are applied to every element of an array.

Arithmetic between an array and a scalar

> **Example**
>
> ```
> arr = np.arange(6).reshape(2, 3)
>
> print(f"arr is \n{arr}\n")
>
> print(f"arr plus scalar 10 \n{arr + 10}\n")
>
> print(f"scalar 4 subtracted from arr is \n{arr - 4}\n")
>
> print(f"arr multiplied by scalar 3 \n{arr * 3}\n")
> ```
>
> yields
>
> ```
> arr is
> [[0 1 2]
>  [3 4 5]]
>
> arr plus scalar 10
> [[10 11 12]
>  [13 14 15]]
>
> scalar 4 subtracted from arr is
> [[-4 -3 -2]
>  [-1  0  1]]
>
> arr multipied by scalar 3
> [[ 0  3  6]
>  [ 9 12 15]]
> ```
>
> as you can see original array `arr` has not changed. However...

> **using *augmented assignments* such as `+=`, `-=`, `*=`, `**=`, `/=`, `//=`, `%=` will modify the original array `arr`.** Here is an example:
>
> **Example**
>
> ```
> arr = np.arange(8).reshape(4, 2)
>
> print(f"original array is \n{arr}\n")
>
> arr += 100
> print(f"arr after augmented assignment += is \n{arr}\n")
> ```
>
> yields
>
> ```
> original array is
> [[0 1]
>  [2 3]
>  [4 5]
>  [6 7]]
>
> original array after augmented assignment += is
> [[0 1]
>  [2 3]
>  [4 5]
>  [6 7]]
>
> arr after augmented assignment += is
> [[100 101]
>  [102 103]
>  [104 105]
>  [106 107]]
> ```

Arithmetic between two `numpy` arrays

All arithmetic operations `+`, `-`, `*`, `**`, `/`, `//`, `%` between two arrays are element-wise. Element-wise operations are applied to every element of an array.

> **Example**

```
arr_1 = np.arange(4).reshape(2, 2)
arr_2 = np.arange(5, 9).reshape(2, 2)

print(f"arr_1 is \n{arr_1}\n")

print(f"arr_2 is \n{arr_2}\n")

print(f"arr_1 + arr_2 is \n{arr_1 + arr_2}\n")

print(f"arr_1 - arr_2 is \n{arr_1 - arr_2}\n")

print(f"arr_1 * arr_2 is \n{arr_1 * arr_2}\n")
```

yields

```
arr_1 is
[[0 1]
 [2 3]]

arr_2 is
[[5 6]
 [7 8]]

arr_1 + arr_2 is
[[ 5  7]
 [ 9 11]]

arr_1 - arr_2 is
[[-5 -5]
 [-5 -5]]

arr_1 * arr_2 is
[[ 0  6]
 [14 24]]
```

Note: augmented assignments such as `+=`, `-=`, `*=`, `**=`, `/=`, `//=`, `%=` can be applied between two arrays as well and the impact will be element-wise.

Basic reductions in `numpy`

You can summarize/describe an array using `sum()`, `min()`, `max()`, `median`, `mean()`, `std()`, `var()`. These could be done in a *quick & dirty* way, such as

> **Example**
>
> ```
> grades = np.random.randint(70,100,20)
> print(grades.max())
> ```

You can use these functions in a more tailored manner as well. Here is a sneak peek:

You can use `np.sum(grades)` or `np.min(grades)`. In this manner, you can provide many more arguments to these functions for reducing (summarizing) your data.

Note about `np.var()` function: `np.var(array, ddof = 0)` provides population variance; whereas, `np.var(grades,ddof = 1)` provides the sample variance. The same options exist for `np.std()` function.

To calculate reduction functions for a row or a column, we need to learn about the concept of `axis` first.

Concept of axis in `numpy`: (`axis = 0` vs. `axis = 1`)

The concept of `axis` is key in working with `numpy` arrays and later with `DataFrame` structures of `pandas`. `pandas` follows `numpy`'s use of the word `axis`. The usage is explained in `numpy`'s glossary of terms:

> Axes are defined for arrays with more than one dimension. A 2-dimensional array has two corresponding axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1).

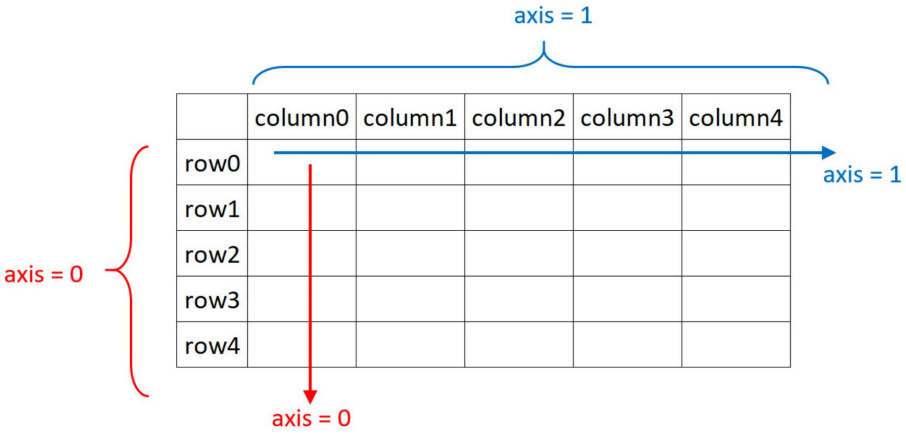
It might help to remember

- 0 = down
- 1 = across

In other words,

- Use `axis = 0` to apply a method down each column to each row.
- Use `axis = 1` to apply a method across each row to each column.

The following image depicts `axis = 0` versus `axis = 1`



`numpy` reductions for each row/column

`numpy` can `reduce` or summarize an array along its rows or columns using the `axis` argument. `axis = 0` will calculate along the columns and `axis = 1` will calculate along the rows. The same idea can be applied in higher dimensions as well. Here is an example:

> **Example**
>
> ```
> arr = np.arange(6).reshape(2, 3)
>
> print(f"original arr is \n{arr}\n")
>
> print(f"sum along each column is \n{np.sum(arr, axis = 0)}\n")
>
> print(f"sum along each row is \n{np.sum(arr, axis = 1)}\n")
> ```
>
> yields
>
> ```
> original arr is
> [[0 1 2]
> [3 4 5]]
>
> sum along each column is
> [3 5 7]
> ```

```
sum along each row is
[ 3 12]
```

Broadcasting

We have already seen that operations between two `numpy` arrays (equal size) operate element-wise. Here is an example:

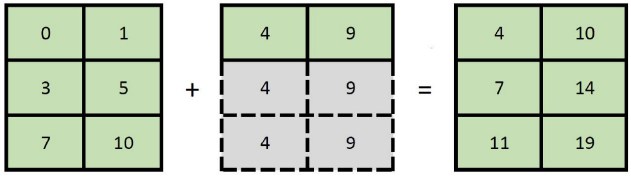
> **Example**
>
> ```
> arr_1 = np.arange(1, 6)
> arr_2 = np.arange(10, 15)
> print(arr_1 + arr_2)
> ```
>
> yields
>
> ```
> [11 13 15 17 19]
> ```

But:

what about unequally-sized arrays?

This is where the very important concept of **broadcasting** comes into play. According to `numpy`'s documentation:

> The term broadcasting describes how `numpy` treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "broadcast" (stretch) across the larger array so that they have compatible shapes. Arrays do not need to have the same number of dimensions.
>
> **Example**
>
> ```
> arr_1 = np.array([[0, 1],[3, 5],[7, 10]])
> arr_2 = np.array([4, 9])
> arr_3 = arr_1 + arr_2
>
> print(arr_3)
> ```
>
> `arr_1` and `arr_2` are differently sized and differently-shaped arrays. When you try to add them, `numpy` broadcasts (stretches) the smaller array, in this case `arr_2`, to the extent that its size becomes the same as `arr_1` and then performs the element-wise operations. This is shown in the following image:



this operation yields

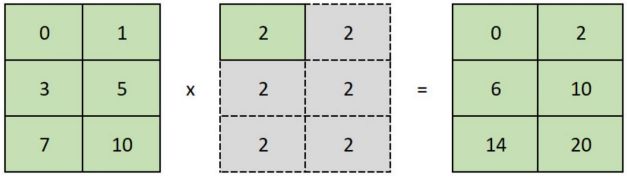
```
array([[ 4, 10],
       [ 7, 14],
       [11, 19]])
```

we have been using broadcasting all along. Here is a familiar example

Example

```
arr_4 = np.array([[0, 1],[3, 5],[7, 10]])
arr_5 = arr_4 * 2
print(arr_5)
```

`arr_4` is in a different shape and size than scalar `2`; `numpy` first broadcasts scalar `2` to the size of `arr_4` and then performs the arithmetic operation. This is depicted in the following image:



this operation yields

```
array([[ 0, 2],
       [ 6, 10],
       [14, 20]])
```

References

This document is an adaption from the following references:

1- *Python online documentation*; available at https://docs.python.org/3/

2- *A Byte of Python*; available at https://Python.swaroopch.com/

3- *Introduction to programming in Python*; available at https://introcs.cs.princeton.edu/Python/home

4- *Introduction to Computation and Programming Using Python: With Application to Understanding Data*, John Guttag, The MIT Press, 2nd edition, 2016

5- *Introduction to Python for Computer Science and Data Science*, Paul J. Deitel, Harvey Deitel, Pearson, 1st edition, 2019

6- *Data Mining for Business Analytics*, Galit Shmueli, Peter C. Bruce, Peter Gedeck, Nitin R. Patel, Wiley, 2020

7- *Python for Data Analysis, Data Wrangling with Pandas, NumPy, and IPython*; Wes McKinney, O'Reilly Media; 2nd edition, 2017

8- *Python Data Science Handbook: Essential Tools for Working with Data*, Jake VanderPlas, O'Reilly Media; 1st edition, 2016

9- *Data Visualization in Python with Pandas and Matplotlib*, David Landup, Independently published, 2021

10- *Introduction to Programming Using Java*; available at http://math.hws.edu/javanotes/

11- *Python for Programmers with introductory AI case studies*, Paul Deitel, Harvey Deitel, Pearson, 1st edition, 2019

12- *Effective Pandas: Patterns for Data Manipulation (Treading on Python)*, Matt Harrison, Independently published, 2021

13- https://betterprogramming.pub/

14- https://www.learnpython.org/

15- https://realpython.com/

16- *Numerical Python*, Robert Johansson, 2nd edition, 2019, Apress

17- *Array programming with NumPy*; Nature, Vol 585, September 2020

18- https://numpy.org/

19- https://betterprogramming.pub/a-comprehensive-guide-to-numpy-data-types-8f62cb57ea83