

ENGR 298: Engineering Analysis and Decision Making – Time-Series Data and Numpy

Dr. Jason Forsyth
Department of Engineering
James Madison University

On Data

- Information required for an engineering application may come in a variety of forms: images, text, binary files.. Will explore various forms/types and the formats in which they are held
- *Sequential data*: a series of data points that have some ordered or temporal relationship. Examples: a series of temperature measurements over time
- *Spatial data*: a set of information that has distance/organizational relationships. Examples: an image or video

Data and File Format

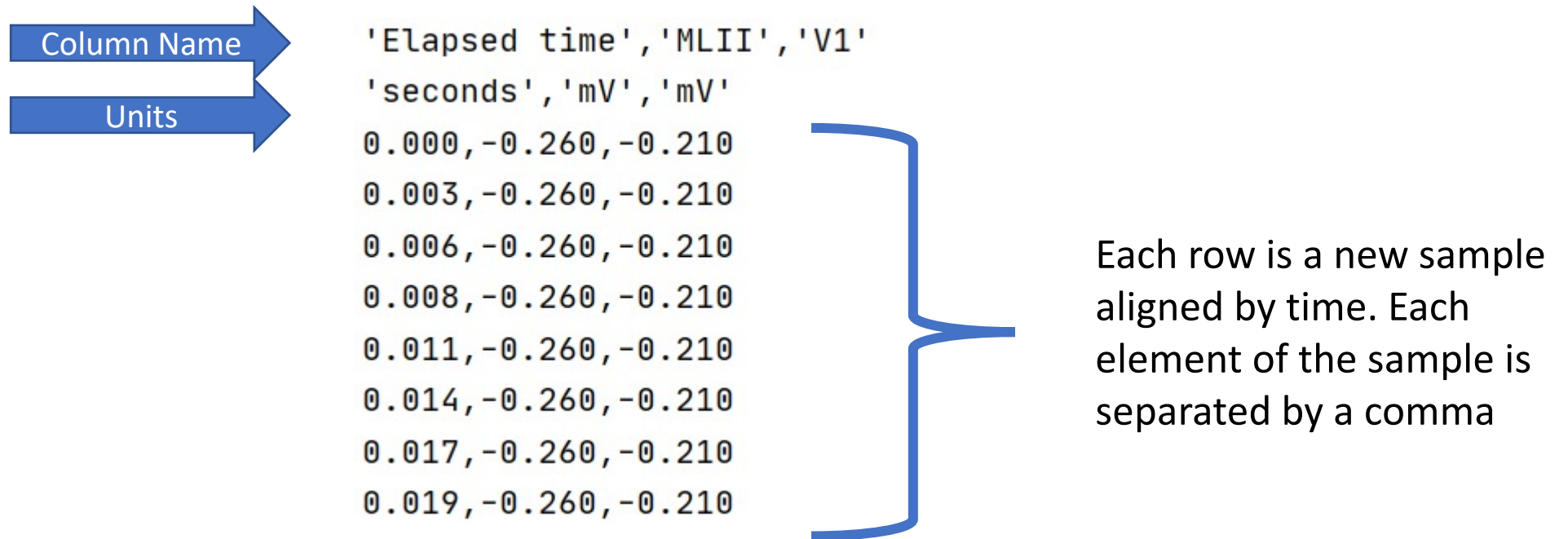
- Our information will generally be stored in various files on the computer (not streamed from cloud, API...etc)
- The **format** of a file, and its information **content**, are often linked based upon how easily content can be stored, represented, sorted..etc.
- A .txt file can easily store sequential information, but could be forced to encode an image (spatial information) if need be.

Common File Formats

- **.txt**: a text file. Likely just a set of ASCII character
- **.csv**: a common separated value file. Represents rows of a table where each row is on a new line and each element of the row is separated by a comma (,)
- **.tsv**: similar to CSV but separated by “tabs” or other whitespace characters
- **JSON**: a “human readable” relationship of values and attributes to store complex data structures
- **XML**: a “human” and machine readable file format to hold data structures
- **.bin**: a binary file. Likely an executable program or information stored in machine specific ‘binary’ representation. Not human readable.

Time-Series Data and CSVs

- Temporal information is commonly stored in CSVs. Easy for humans (and machines) to read. Can have header to name columns and each column is a different information stream.



The diagram illustrates the structure of a CSV file. On the left, two blue arrows point to the right. The top arrow is labeled 'Column Name' and points to the first column of the header. The bottom arrow is labeled 'Units' and points to the second and third columns of the header. The header consists of three rows: the first row contains column names, the second row contains units, and the third row contains numerical data. The data rows follow, each containing three numerical values separated by commas. A large blue bracket on the right side of the data rows points to a text box explaining that each row is a new sample aligned by time, and each element of the sample is separated by a comma.

Column Name	Units	
'Elapsed time'	'MLII'	'V1'
'seconds'	'mV'	'mV'
0.000	-0.260	-0.210
0.003	-0.260	-0.210
0.006	-0.260	-0.210
0.008	-0.260	-0.210
0.011	-0.260	-0.210
0.014	-0.260	-0.210
0.017	-0.260	-0.210
0.019	-0.260	-0.210

Each row is a new sample aligned by time. Each element of the sample is separated by a comma

Importing CSVs into Python

- CSV information can be imported using several packages (numpy or pandas) or the file can be manually parsed.
- Once imported, information can be easily represented as matrix (all information) or as vectors/arrays (for each column).
- Whether to utilize a package (numpy/pandas) or manually parse depends on how “regular” the file format is. If it is consistent (and the packages import correctly), then use them. Otherwise, can manually parse to ensure each row is treated correctly.
- Packages (numpy/pandas) will likely be faster than a manual/human implemented parser but may need to configure/tweak for each file type.

Opening a File in Python

- Files can easily be opened via `open()`. Must provide a valid path to file location. Returns a *file object*.
- `File.read()` will iterate through the file reading each character.
- `File.readline()` will read a single line and return as a string. Note: line endings (`\r\n`) are different between Windows and Mac/Linux systems. May cause trouble at times when passed between computers.

```
open(file, mode='r', buffering=- 1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

Open *file* and return a corresponding [file object](#). If the file cannot be opened, an `OSError` is raised. See [Reading and Writing Files](#) for more examples of how to use this function.

file is a [path-like object](#) giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed unless *closefd* is set to `False`.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), `'x'` for exclusive creation, and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform-dependent: `locale.getpreferredencoding(False)` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
<code>'r'</code>	open for reading (default)
<code>'w'</code>	open for writing, truncating the file first
<code>'x'</code>	open for exclusive creation, failing if the file already exists
<code>'a'</code>	open for writing, appending to the end of file if it exists
<code>'b'</code>	binary mode
<code>'t'</code>	text mode (default)
<code>'+'</code>	open for updating (reading and writing)

The default mode is `'r'` (open for reading text, a synonym of `'rt'`). Modes `'w+'` and `'w+b'` open and truncate the file. Modes `'r+'` and `'r+b'` open the file with no truncation.

Opening Files and Reading Lines

'Elapsed time','MLII','V1'

'seconds','mV','mV'

0.000,-0.260,-0.210

0.003,-0.260,-0.210

0.006,-0.260,-0.210

0.008,-0.260,-0.210

0.011,-0.260,-0.210

0.014,-0.260,-0.210

0.017,-0.260,-0.210

0.019,-0.260,-0.210

ekgdata/mitdb_201.csv

Approach #1: do a manual import for the file

path = 'ekg-data/mitdb_201.csv' path: 'ekg-data/mitdb_201.csv'

open the file

file = open(path) file: <_io.TextIOWrapper name='ekg-data/mitdb_201.csv'

read the first line to get the headers

header = file.readline() header: '\Elapsed time\','MLII\','V1'\n'

read the second line to get the units

units = file.readline() units: '\seconds\','mV\','mV'\n'

*After clearing out header information, can now
iterate/loop through file and parse each line*

Iterating through each line

```
# iterate through remainder of file
for line in file: line: '0.000,-0.260,-0.210\n'
    #split line based upon commas
    elements = line.split(",") elements: ['0.000', '-0.260', '-0.210\n']

    # manually assign time, ML2, and V2 values
    time = elements[0] time: '0.000'
    ml2 = elements[1] ml2: '-0.260'
    v1 = elements[2] v1: '-0.210\n'
```

```
'Elapsed time','MLII','V1'
'seconds','mV','mV'
0.000,-0.260,-0.210
0.003,-0.260,-0.210
0.006,-0.260,-0.210
0.008,-0.260,-0.210
0.011,-0.260,-0.210
0.014,-0.260,-0.210
0.017,-0.260,-0.210
0.019,-0.260,-0.210
```

line

For loop will iterate across file storing each line in the variable "line". Line is then split() based upon commas to generate an array called elements. Each element is then assigned its appropriate variable name

Iterating through each line

```
# iterate through remainder of file
for line in file: line: '0.000,-0.260,-0.210\n'
    #split line based upon commas
    elements = line.split(",") elements: ['0.000', '-0.260', '-0.210\n']

    # manually assign time, ML2, and V2 values
    time = elements[0] time: '0.000'
    ml2 = elements[1] ml2: '-0.260'
    v1 = elements[2] v1: '-0.210\n'
```



Challenge: is v1 actually an integer or is it a string?

Solution is to strip() string of whitespace characters

```
'Elapsed time','MLII','V1'
'seconds','mV','mV'
0.000,-0.260,-0.210
0.003,-0.260,-0.210
0.006,-0.260,-0.210
0.008,-0.260,-0.210
0.011,-0.260,-0.210
0.014,-0.260,-0.210
0.017,-0.260,-0.210
0.019,-0.260,-0.210
```

line

```
# iterate through remainder of file
```

```
for line in file: line: '0.000,-0.260,-0.210\n'
```

```
# split line based upon commas
```

```
elements = line.strip().split(",") elements: ['0.000', '-0.260', '-0.210']
```

```
# manually assign time, ML2, and V2 values
```

```
time = elements[0] time: '0.000'
```

```
ml2 = elements[1] ml2: '-0.260'
```

```
v1 = elements[2] v1: '-0.210'
```



If parsing a file, store the points in some list / data structure

```
# make a list to hold this information
points = list()

# iterate through remainder of file
for line in file:
    # split line based upon commas
    elements = line.strip().split(",")

    # manually assign time, ML2, and V2 values
    time = elements[0]
    ml2 = elements[1]
    v1 = elements[2]

    # add data points in list tuple
    points.append((time, ml2, v1))
```

The list points now contains 65,000 tuples, where each stores a (time, ml2, and v1) sequence.

Using numpy to load files...

- Since our data is all values/numbers, would be useful to automatically load into numpy matrix/arrays for calculation.
- Numpy will operate on entire file at once and parameters should be passed to `load()` function to ensure the is parsed properly.
- Assumes all lines follow CSV convention and are numerical values. Any lines that do not follow will cause error and should be skipped.

Loading Files and Skipping Lines

```
'Elapsed time','MLII','V1'  
'seconds','mV','mV'  
0.000,-0.260,-0.210  
0.003,-0.260,-0.210  
0.006,-0.260,-0.210  
0.008,-0.260,-0.210  
0.011,-0.260,-0.210  
0.014,-0.260,-0.210  
0.017,-0.260,-0.210  
0.019,-0.260,-0.210
```

- First row and second rows are not “numerical data”, should be skipped.
- Must also indicate to numpy what is the **delimiter** for the file (whitespace, tabs, commas...)

numpy.loadtxt

```
numpy.loadtxt(fname, dtype=<class 'float'>, comments='#',  
delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False,  
ndmin=0, encoding='bytes', max_rows=None, *, like=None) \[source\]
```

Load data from a text file.

Each row in the text file must have the same number of values.

Parameters: *fname* : *file, str, pathlib.Path, list of str, generator*

File, filename, list, or generator to read. If the filename extension is *.gz* or *.bz2*, the file is first decompressed. Note that generators must return bytes or strings. The strings in a list or produced by a generator are treated as lines.

dtype : *data-type, optional*

Data-type of the resulting array; default: float. If this is a structured data-type, the resulting array will be 1-dimensional, and each row will be interpreted as an element of the array. In this case, the number of columns used must match the number of fields in the data-type.

comments : *str or sequence of str, optional*

The characters or list of characters used to indicate the start of a comment. None implies no comments. For backwards compatibility, byte strings will be decoded as 'latin1'. The default is '#'.

delimiter : *str, optional*

The string used to separate values. For backwards compatibility, byte strings will be decoded as 'latin1'. The default is whitespace.

Load CSV file via Numpy

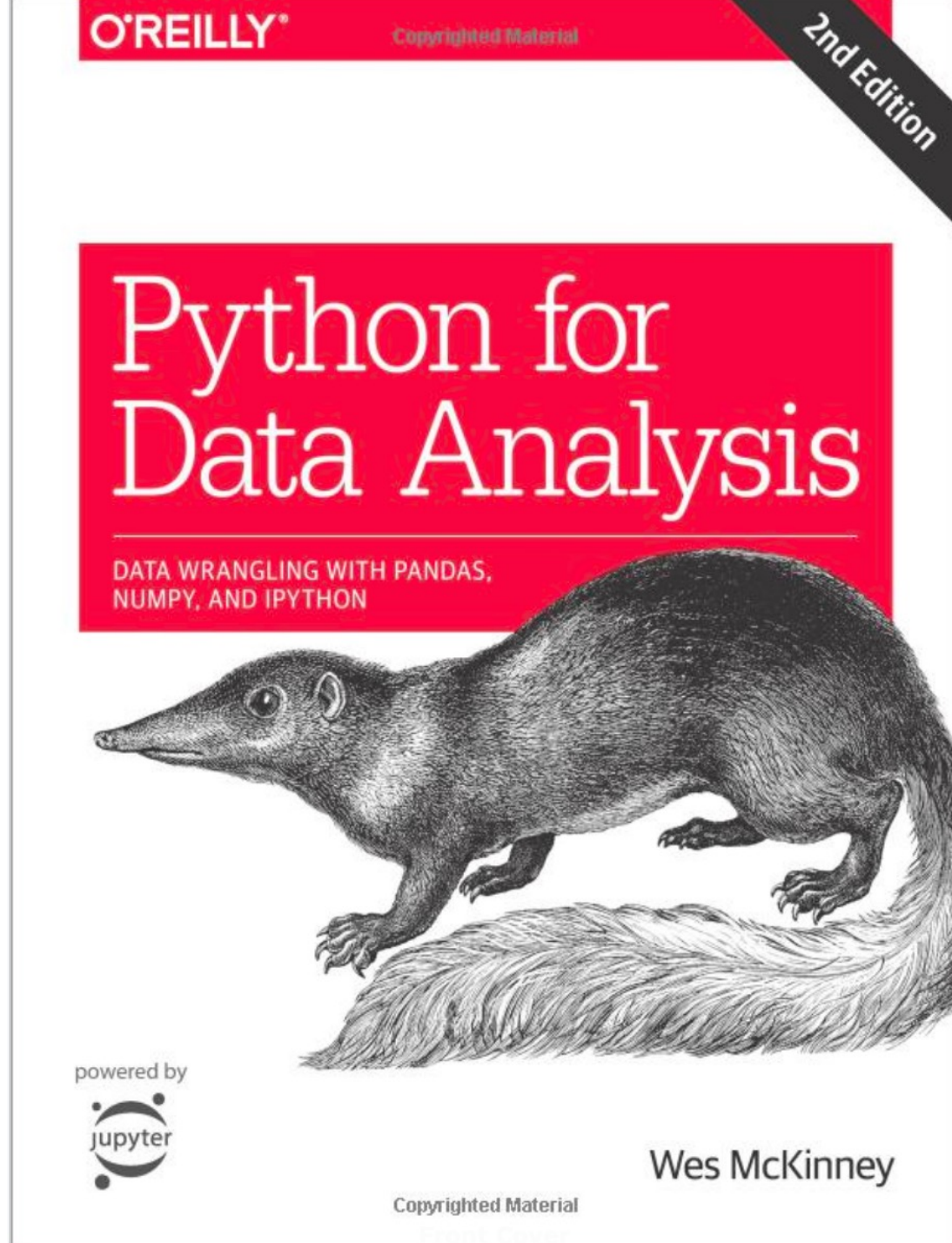
```
# Approach #2: import the CSV file using numpy
path = 'ekg-data/mitdb_201.csv'

# load data in matrix from CSV file; skip first two rows
ekg_data = np.loadtxt(path, skiprows=2, delimiter=",")
```

	0	1	2
0	0.00000	-0.26000	-0.21000
1	0.00300	-0.26000	-0.21000
2	0.00600	-0.26000	-0.21000
3	0.00800	-0.26000	-0.21000
4	0.01100	-0.26000	-0.21000
5	0.01400	-0.26000	-0.21000
6	0.01700	-0.26000	-0.21000
7	0.01900	-0.26000	-0.21000
8	0.02200	-0.26000	-0.19500
9	0.02500	-0.25000	-0.18500
10	0.02800	-0.24000	-0.17500

- Parameters for `loadtxt()` tell numpy to skip the first two rows and that each line is delimited by commas
- Result is variable “ekg_data” which is a 65,000 x 3 matrix.
- Data can now be accessed through `[][]` notation.

NumPy



One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data. There are a number of reasons for this:

- NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy's library of algorithms written in the C language can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.
- NumPy operations perform complex computations on entire arrays without the need for Python for loops.

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 8 ms, total: 28 ms
Wall time: 26.5 ms
```

```
In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 408 ms, sys: 64 ms, total: 472 ms
Wall time: 473 ms
```

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a **shape**, a tuple indicating the size of each dimension, and a **dtype**, an object describing the *data type* of the array:

```
import numpy as np
```

```
# make a basic array
```

```
arr2d = np.array([[1,2,3],[4,5,6],[7,8,9]])  arr2d: [[1 2 3], [4 5 6], [7 8 9]]
```

```
# what's the array shape
```

```
shape = arr2d.shape  shape: (3, 3)
```

```
# what type of data is it holding
```

```
type = arr2d.dtype  type: int64
```

Accessing Data

Let's Recall Matrices...

An N by M matrix has N-rows and M-columns.

This format can be extended to arbitrary dimensions, but let's start with 2 for now.

The square matrix to the right is 3x3. All axes start at 0 (thank you this isn't MATLAB)

		axis 1		
		0	1	2
axis 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

$$\begin{bmatrix} 0,0 & \cdots & 0,m \\ \vdots & \ddots & \vdots \\ n,0 & \cdots & n-1,m-1 \end{bmatrix}$$

Accessing Data in a ND Array

- Data is accessed in “matrix” notation indicating the index/range for each dimension. `Array[row, column]`; or `Array[dimension, dimension, dimension...]` for higher dimensional data
- For two-dimensional data:
 - `Array[2,2]` return the (2,2) element
 - `Array[0]` return the 0th row and include all columns
- The colon `:` indicates “all” for some dimension
 - `Array[1,:]` return all columns for row 1
 - `Array[:,2]` return the column 2 for all rows

Accessing Data in ND Array

1	2	3
4	5	6
7	8	9

```
# make a basic array  
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
# access single elements (0,0) and (2,2)  
item = arr2d[0][0]  item: 1  
item2 = arr2d[2][2]  item2: 9
```

```
In[8]: arr2d[0,:]          • Get the first row  
Out[8]: array([1, 2, 3])  
In[9]: arr2d[1,:]          - Get the second row  
Out[9]: array([4, 5, 6])  
In[10]: arr2d[:,2]         - Get the right most column  
Out[10]: array([3, 6, 9])
```


Slicing In Array Access

1	2	3
4	5	6
7	8	9

```
In[19]: arr2d[:2]  
Out[19]:  
array([[1, 2, 3],  
       [4, 5, 6]])
```

From the matrix, grab the first two rows

```
In[23]: arr2d[1,:2]  
Out[23]: array([4, 5])
```

From row 1, grab the first two columns

```
In[24]: arr2d[:, :1]  
Out[24]:  
array([[1],  
       [4],  
       [7]])
```

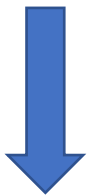
From all rows, grab the first column

Be Careful with indexing

```
In[19]: arr2d[:2]
```

```
Out[19]:
```

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



```
In[21]: arr2d[:2][0]
```

```
Out[21]: array([1, 2, 3])
```

From the resulting 2x3 array, grab the 0th row



```
In[22]: arr2d[:2,0]
```

```
Out[22]: array([1, 4])
```

From the first two rows, grab the 0th column

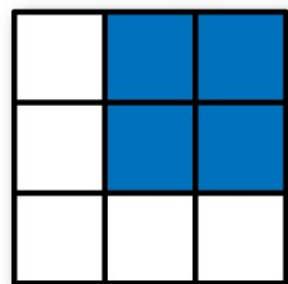
1	2	3
4	5	6
7	8	9

Fancy Indexing

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

```
In [120]: arr[[4, 3, 0, 6]]  
Out[120]:  
array([[4., 4., 4., 4.],  
       [3., 3., 3., 3.],  
       [0., 0., 0., 0.],  
       [6., 6., 6., 6.]])
```

Grab the 4th, 3rd, 0th, and 6th rows

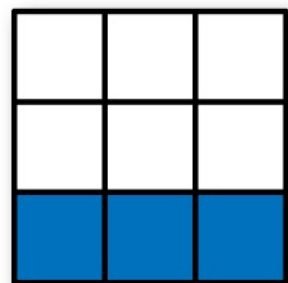


Expression

`arr[:2, 1:]`

Shape

`(2, 2)`



`arr[2]`

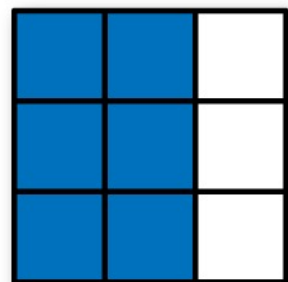
`(3,)`

`arr[2, :]`

`(3,)`

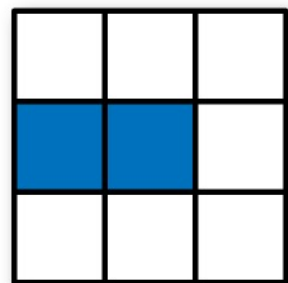
`arr[2:, :]`

`(1, 3)`



`arr[:, :2]`

`(3, 2)`



`arr[1, :2]`

`(2,)`

`arr[1:2, :2]`

`(1, 2)`

Arithmetic operations

Or the entire reason that Numpy exists

Basic functions...

```
In[47]: a=np.random.randint(10,size=5)
```

```
In[48]: a
```

```
Out[48]: array([1, 3, 3, 1, 7])
```

```
In[49]: b=np.random.randint(10,size=5)
```

```
In[50]: b
```

```
Out[50]: array([2, 5, 7, 9, 7])
```

```
In[51]: c=a+b
```

```
In[52]: c
```

```
Out[52]: array([ 3,  8, 10, 10, 14])
```

Piece-wise addition

```
In[54]: a**2
```

```
Out[54]: array([ 1,  9,  9,  1, 49])
```

Element-wise power operation

```
In[55]: a**2 + b**2
```

```
Out[55]: array([ 5, 34, 58, 82, 98])
```

Chained power and addition operations

```
In[56]: sum(a**2 + b**2)
```

```
Out[56]: 277
```

Chained power and addition operations

Vector Operations

```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
```

```
In [156]: xs, ys = np.meshgrid(points, points)    xs and ys are now 2x2
```

```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)     $Z = xs^2 + ys^2$     Result z is 2x2
```

```
In [159]: z
```

```
Out[159]:
```

```
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
```