

# Vectors, Matrices in Python

## What is NumPy?

NumPy, short for Numerical Python, is a package - it doesn't ship with basic Python, and you'll need to install to use it. Once installed, you need to import it inside your Python session (or script) using the following command:

```
>>> import numpy as np
```

This means "import the numpy functionality, store it in a variable called **np**". You could replace **np** with any other name, but **np** is the conventional name choice.

## Why NumPy?

What do you expect the following code snippet in basic Python does?

```
>>> [1,2,3] * 3
```

If you have experience with vector arithmetic, you may have expected that each element would be multiplied by 3, yielding [3, 6, 9]. The answer is, however:

```
[1,2,3,1,2,3,1,2,3]
```

Some would call this a poor design choice on Python's, but it could be overcome by designing a new data structure (called the '**ndarray**') with all the desired behaviours. This is what the NumPy authors did. They also provided a neat function for creating one, called **.array**:

```
>>> np.array([1,2,3]) * 3  
array([3, 6, 9])
```

Don't be too concerned with the `array()` syntax in the output - that just signifies that the result is an ndarray too. An ndarray interprets the multiplication operator as a '**vectorised**' operation, which is essential for scientific computing.

Ndarrays are much more efficient than regular Python lists. As a result, many other popular Python packages are based on NumPy, including pandas, SciPy, Keras, Matplotlib, and others.

## What is an ndarray?

Simply put, an ndarray is a container of data items that have the same type (e.g. numbers, text strings), i.e. it is '**homogenous**'.

An ndarray could have any number of dimensions, the simplest being a vector.

```
5 6 5 3 7 5
```

An ndarray could also be a **matrix**, which also could be created using the **.array** function, but this time passing a nested list:

```
>>> np.array([[5,6,5],  
[3,5,6]])  
array([[5, 6, 5],  
[3, 5, 6]])
```

```
5 6 5  
3 7 5
```

From the computer's point of view, a matrix is just another one-dimensional sequence, but by keeping meta-data about the matrix' shape, it remembers to treat it as a 2x3 structure. Such pieces of meta-data are called **attributes** and the most important are:

<b>.shape</b>	Tuple of dimension sizes, in this case <b>(2,3)</b> . Think of a tuple as a list, but created with round brackets instead of square <b>[]</b> .
<b>.ndim</b>	Number of array dimensions, in this case <b>2</b> .
<b>.size</b>	Number of elements, i.e. 6.

You can turn a one-dimensional array into a two-dimensional one using the **.reshape**:

```
>>> np.array([5,6,5,3,5,6]) \  
    .reshape((2,3))  
array([[5, 6, 5],  
[3, 5, 6]])
```

Ndarrays could have more than two dimensions, but this will not be covered here.

## Methods versus functions

With NumPy it is worth distinguishing between functions and **methods**. A method is a function that is associated with an object. NumPy also has functions simply belonging to the library, but these require you to explicitly pass an ndarray as argument. The following is a function:

```
>>> x = np.array([1,2,3,4])
```

The following is a method:

```
>>> x.reshape((2,2))
```

## Generating ndarrays

There are many NumPy functions available for conveniently generating common ndarrays:

### Arithmetic sequences

One common scenario is creating arithmetic sequences, with regularly spaced out numbers. This is done using **.arange**, which could have 2 arguments (start-value, excluded end-value) and optionally a third representing step-size:

1	3	5	7	9
---	---	---	---	---

```
>>> np.arange(1,10,2)  
array([1, 3, 5, 7, 9])
```

### Tiling arrays

You may want to repeat an ndarray using **.tile**, simply pass the ndarray and the times to repeat.

1	3	▶	1	3	1	3	1	3
---	---	---	---	---	---	---	---	---

```
>>> np.tile([1,3],3)
```

### Repeating arrays

You could repeat elements using **.repeat**.

1	3	▶	1	1	1	3	3	3
---	---	---	---	---	---	---	---	---

```
>>> np.repeat([1,3],3)
```

## Concatenating arrays

To combine ndarrays, use **.concatenate**.

```
>>> np.concatenate((x,y))
```

1	2	3	4
x		y	
▼			
1	2	3	4

1	3	5
2	4	6
x		
7	9	11
8	10	12
y		

1	3	5
2	4	6
7	9	11
8	10	12

For matrices, it takes an argument called **axis**, which represents the dimension (0 for row-wise, 1 for column-wise).

```
>>> np.concatenate((x,y),axis=1)
```

You'll see this axis argument re-appear in other NumPy functions.

## Method Chaining

Usually we will process data in many steps - selecting, filtering, grouping, etc. NumPy and Pandas methods are designed to be chained together, causing them to execute in sequence. This is called "**method chaining**".

In the example below, we have an ndarray **x** that we repeat twice, fold, sum column-wise, and then take the mean. The results are stored in "intermediate variables" **x2**, **x3**, and **x4**.

```
>>> x1 = np.tile(x, 2)  
>>> x2 = x1.reshape((2,4))  
>>> x3 = x2.sum(axis=1)  
>>> x3.mean()
```

Both NumPy and Pandas objects have built-in methods that essentially use the object itself as an input and outputs the modified object. This means that you can chain them all together using the dot operator, and get the same result (the **\** is used to make it span multiple

```
>>> np.tile(x, 2) \  
    .reshape((2,4)) \  
    .sum(axis=1) \  
    .mean()
```

# DataFrames in Python

To a first approximation, a **DataFrame** - which ships with the **Pandas** library - is simply a **data table**. If structured well, each row represents a **sample** (for example, a person or a country) and each column represents a **scientific variable** (a name, a size) describing the sample.

In addition to the data itself, the DataFrame has several attributes with meta-data. There most important are `columns`, which simply put is a list containing the column labels, and `index`, which contains row labels.

		.columns		
		A	B	C
.index	0	1	5	3
	1	2	6	4
	2	3	7	5

`.columns` and `.index` are not actually simple lists, but rather are automatically converted into `Index` objects. There are many specialised types of `Index` objects. When creating a DataFrame, the default is a `RangeIndex`, which is defined by a start, stop, and step-value, and effectively stored 0..N.

```
>>> df.columns
Index(['A', 'B', 'C'], dtype='object')
>>> df.index
RangeIndex(start=0, stop=3, step=1)
```

You can access these the way you would any list:

```
>>> df.columns[1]
'B'
>>> 'B' in df.columns
True
```

Although you can overwrite `.columns` and `.index`, you cannot modify individual elements, because `Index` objects are immutable.

```
>>> df.index = ['x', 'y', 'z']
>>> df.index[0] = 'xx'
TypeError: Index does not support mutable operations
```

## Series

Each column in a DataFrame is an array structure known as a `Series`. Whenever you access a single column from a DataFrame, the result is usually a `Series`. They could be thought of as an `ndarray` but with an `.index` attribute that allows you to name elements.

a	b	c	d	e	f	g	h

As ultimately an `ndarray`, a `Series` can only contain a single datatype, but a DataFrame can contain a `Series` of multiple types.

## Creating a dataframe

In most cases, you will be reading in data from an external file (say, a `.csv`, `.txt`, or `.json`), a database, or via a web URL. But there may be times you wish to create a DataFrame from scratch however. You do this via the `.DataFrame`:

The simplest method is to create a Python dictionary (a built-in key-value structure created using curly brackets).

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

```
>>> pd.DataFrame({'A':a, 'B':b, 'C':c},
                  columns = ['A', 'B', 'C'])
   A  B  C
0  'x'  1  4
1  'y'  2  5
2  'z'  3  6
```


The dictionary structure ignores order, so to enforce column order you need to pass `.columns` as an argument. You also have the option to pass a 2D `ndarray` or list as the data:

```
>>> pd.DataFrame(
    [[1,5,3],[2,6,4],[3,7,5]],
    columns = ['A', 'B', 'C'])
```

## Selecting columns

One of the most common operations is to access a single column.

A	B	C
1	5	3
2	6	4
3	7	5



B
5
6
7

df

### Using attribute access

You could access the column as though the column were an attribute, using dot notation and without using quotation marks. This can only be used to return a Series.

```
>>> df.B
```

### Using indexing operator

You may pass the column name as a string to the square brackets. This will return a Series.

```
>>> df['B']
0    5
1    6
2    7
```

If you pass the column-name into a list, it will return a DataFrame instead of a Series.

```
>>> df[['B']]
   B
0  5
1  6
2  7
```

Note that this list-method also can be used to access multiple columns.

### Label-based access

Using the `.loc` attribute, you could combine column-selection with row-selection. This allows you to use slicing.

```
>>> df.loc[:, "A": "C"]
```

### Positional-based access


You could use the `.iloc` attribute to do the same row/column indexing, but with position index.

```
>>> df.iloc[:, 1]
```

## Filtering columns

You can also dynamically select columns based on a condition involving the column names.

A	B	C
1	5	3
2	6	4
3	7	5



B	C
5	3
6	4
7	5

df

### Column label substring

Suppose the column names are "id", "firstname" and "surname". We want to access columns whose name contains the substring "name". For this we could use the like-argument.

```
>>> df.filter(like="name")
```

### Column label patterns


Although it is beyond the scope of this book, you can also filter column names by regular expressions using the `regex` argument.

```
>>> df.filter(regex=r"\d")
```

## Filtering rows

Your options when it comes to filtering rows are similar to those of `ndarrays` and `columns`:

	A	B	C
True	1	5	3
False	2	4	4
True	3	5	5



A	B	C
1	5	3
3	5	5

mask df

### By Boolean indexing

You could pass a condition involving one or several columns:

```
>>> df[df.B==5]
```

This is the same as indexing by mask:

```
>>> df[np.array(True,False,True)]
```

### Positional-based access

You could use the `.iloc` attribute to do access rows purely based on index.

```
>>> df.iloc[[0,2], :]
```

# Indexing in Python

When values are stored in array data structures, such as lists, `ndarrays`, `Series`, or `DataFrames`, we need a way for accessing and modifying specific values. This is called “indexing”, because the numerical address of each element is called an index.

In Python, indexing starts with 0. This means that the first element has index 0, the second element has index 1, and the last element in an array of length `N` has index `N - 1`.



There are three different uses for indexing that are worth distinguishing among:

## Access

We could use it to simply access values - have a peek at what is stored therein:

```
>>> x[1]
5
```

## Assign

We could also use indexing to pry open a particular element in an array and overwrite the value with a new one:

```
>>> x[1] = 8
>>> x
array([2, 8, 0, 4, 1])
```

If you assign a single value (scalar) to multiple elements, that value will be propagated (broadcasted) to all those elements:

```
>>> x[[0,1,2]] = 8
>>> x
array([8, 8, 8, 4, 1])
```

## Modifying

We could also access the value, modify it, and then put it back, for example multiplying by 2:

```
>>> x[1] = x[1] * 2
>>> x
array([2, 16, 0, 4, 1])
```

## Multiple array elements

To specify multiple elements in an `ndarray`, you can put the indices inside a list. You can use this to duplicate and re-order elements. This method is known as “fancy indexing”.

```
>>> x[[1,4,1]]
array([16, 1, 16])
```

## Slicing

In Python, you can also “slice” an array structure by specifying a start-index and an end-index, separated by a colon. The start-value is included, but the end-value is excluded.

```
>>> x
array([2, 16, 0, 4, 1])
>>> x[1:4]
array([16, 0, 4])
```

If you omit the start-index, it will default to 0. If you omit the end-index, it will default to `N-1`.

```
>>> x[:]
array([2, 16, 0, 4, 1])
>>> x[:2]
array([2, 16])
>>> x[2:]
array([0, 4, 1])
```

When an index is negative, it is counted from the end of the array. Below, in an array with five elements, `-2` refers to the index `5-2` (i.e. 3).

```
>>> x[:-2]
array([2, 16, 0])
>>> x[-2:]
array([4, 1])
```

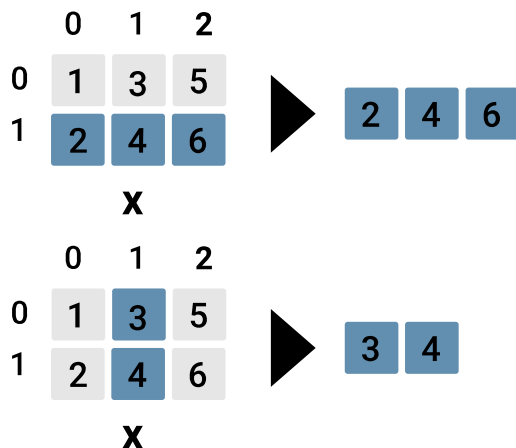
You can also specify a step-size, which requires you to add a colon: `[start:stop:step]`. This is by default 1, and when negative steps through the array backwards.

```
>>> x[::2]
array([2, 0, 1])
>>> x[::-2]
array([1, 0, 2])
```

Slicing only works within `[]` - you cannot use it to generate an array. The slicing operator creates a slice object

## Entire rows or columns

To access a row, you could only pass the row-index: a single number will be interpreted as row-index. To access a column, place the slicing operator in place of the row index.

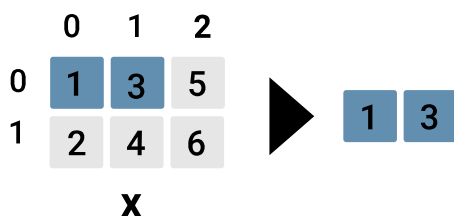


```
>>> x[1]      #x[1,:] works too
array([2,4,6])
>>> x[:,1]
array([3,4])
```

You can think of the slice operator as representing “all indices”, which ensures that the entire row/column is returned.

## Row/column combination

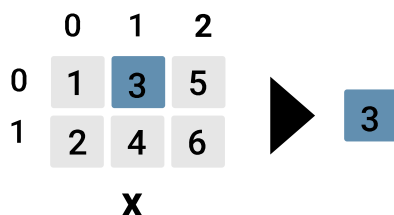
You can slice by both rows and columns at the same time:



```
>>> x[0,[0:1]]
array([1,3])
```

## Elements by single index

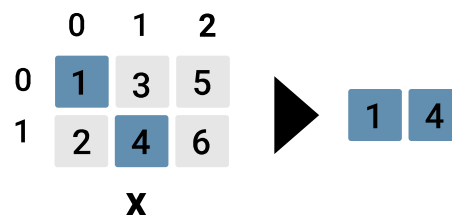
To specify a single element from a matrix, you can either pass the indices as normal, or in two steps, each with their own square brackets:



```
>>> x[0][1]      #x[0,1]
3
```

## Multiple values by index

If you have the indices of multiple elements stored in an array, make sure, each element's indices are in a column:



```
>>> indices
array([[0,1],
       [0,1]])
>>> x[list(indices)]
array([1,4])
```

## Boolean indexing

### Elements by condition

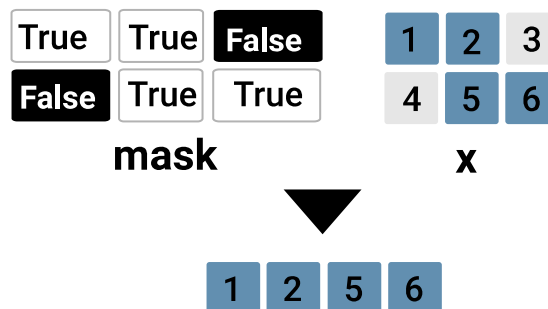
A logical condition involving an ndarray creates an ndarray of Boolean values that could be used as an index to access the values whose corresponding value is True.



```
>>> x%2 == 0
array([False, True, False, True, False])
>>> x[x%2 == 0]
array([2,4])
```

### Elements by mask

You can also create an explicit mask without a condition and use that for indexing:



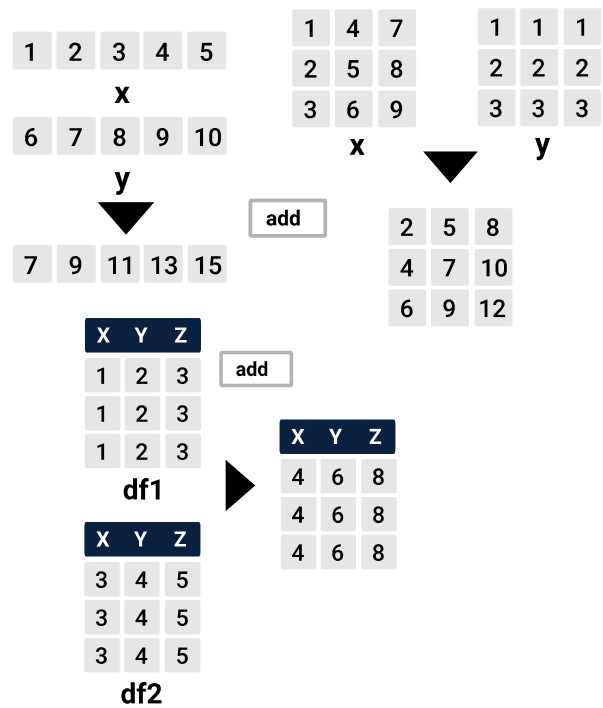
```
>>> mask
array([[True, True, False],
       [False, True, True]])
>>> x[mask]
array([1,2,5,6])
```



# Arithmetic

## Element-wise arithmetic

The most common operation accepts one or more values and transform them into another. When values are elements in a structure, such as a vector, each element can be thought of as its own parallel operation. This is known as **element-wise arithmetic**. Below we see element-wise addition for vectors, matrices and dataframes.



This can be done using the basic **binary operators** like + or \*:

```
>>> np.arange(1,7) +
np.arange(6,11)
array([7, 9, 11, 13, 15])
```

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modulus (i.e. remainder)
//	Integer division

These arithmetic operations could also be done using a set of functions inside NumPy known as universal functions or **ufuncs**, which are carefully optimised for efficiency.

Ufuncs can be binary or unary:

### Unary ufuncs

abs	Returns absolute value
sqrt	Returns square root
square	Returns square
ceil	Returns ceiling
floor	Returns floor
round	Rounds decimals

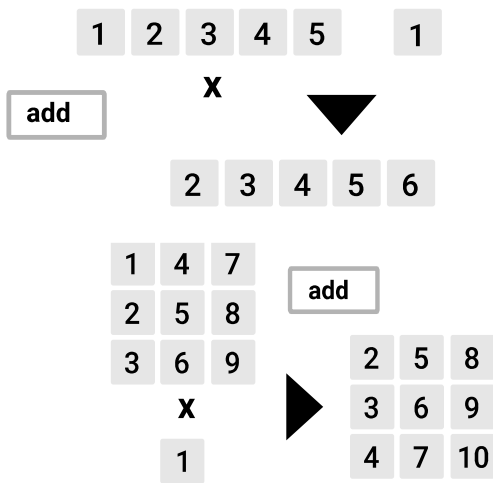


### Binary ufuncs

add	Returns absolute value
subtract	Returns square root
multiply	Returns square
divide	Returns ceiling
floor_divide	Returns floor
power	Exponentiation
mod	Modulus

### Structures of different shape

If you have a binary operation involving two structures where one is a scalar, then that scalar will be repeated ("**broadcast**") for each element in the larger structure:



If the smaller ndarray isn't of size 1, it will still be repeated for as many times as it can, assuming that at least one of the dimensions of the smaller ndarray has size 1. If one is a (2,3) ndarray and one is (1,3) (in other words, a single row vector), the row will be repeated.

## Pair-wise arithmetic

If you have two vectors and wish to pair up every element in one vector, with element in the other, then that is known as pair-wise.

	1	2	3	4	5	x
2	2	4	6	8	10	
4	4	8	12	16	20	
1	1	2	3	4	5	
2	2	4	6	8	10	
6	6	12	18	24	30	
y						

This can be done by chaining the method **outer** to the binary ufunc methods. The result is a two-dimensional ndarray:

```
>>> np.multiply.outer(x,y)
array([[ 2,  4,  6,  8, 10],
       [ 4,  8, 12, 16, 20],
       [ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 6, 12, 18, 24, 30]])
```

For more general functions, the trick lies in reshaping one of the vectors so that it becomes vertical instead, by using **reshape**.

For example, the vectors below have length 5, so reshape it into an ndarray with 5 rows and 1 columns using **reshape(5,1)** and then apply the arithmetic expression:

```
>>> y = y.reshape(5,1)
>>> y
array([[2],
       [3],
       [4],
       [5],
       [6]])
>>> x * y
```



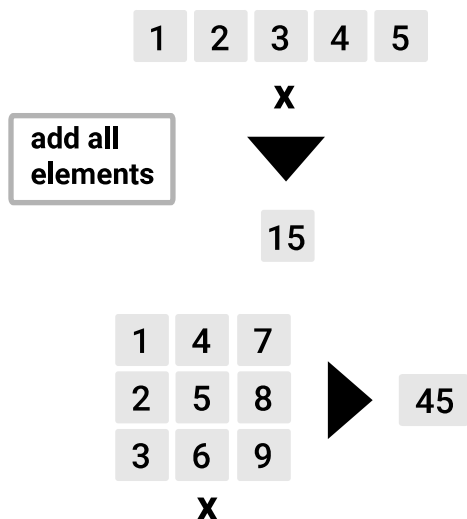
# Aggregations

One type of computation involves turning many values - whether in a vector or matrix - into a single scalar value. Such a many-to-one operation is often called an **aggregation** or **reduction**.

## Methods and functions

In NumPy, many aggregation functions are available as both functions (which belong to the library, i.e. accessed via the variable referring to NumPy) or as a method:

```
>>> x
array([1,2,3,4,5])
>>> np.sum(x)
15
>>> x.sum()
15
```



## Aggregation methods

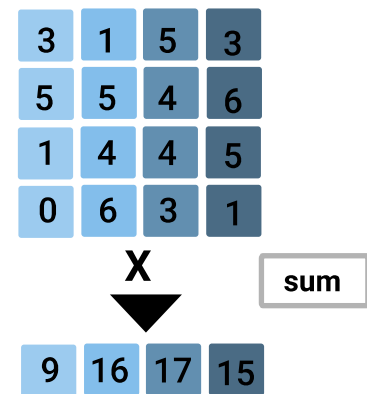
The most common methods are:

<b>sum</b>	Sum all elements
<b>prod</b>	Multiply together all elements
<b>mean</b>	Mean of all elements
<b>np.median</b>	Median of all elements
<b>std, var</b>	Standard deviation, variance
<b>min, max</b>	Minimum, maximum value
<b>argmin, arg-max</b>	Indices of minimum, maximum value

Recall that, to get the length of an ndarray, you use the **.shape** attribute.

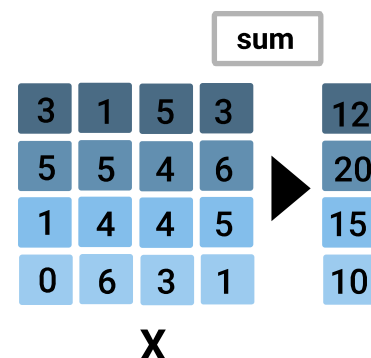
## Row-wise/ Column-wise

In two-dimensional ndarrays, you can perform row-wise and column-wise aggregations by specifying the **axis** argument (row-wise are 1, column-wise are 0).



```
>>> x
array([[3,1,5,3],
       [5,5,4,5],
       [1,4,4,5],
       [0,6,3,1]])
>>> x.sum(axis=0)
array([9,16,17,15])
>>> np.sum(x,axis=0)
array([9,16,17,15])
```

To apply a custom function to rows or columns, you can use the **apply\_along\_axis** NumPy function:



```
>>> np.apply_along_axis(
    x,0,sum)
array([12,20,15,10])
```

# Conditions

Checking whether variables satisfy a condition, for example a numerical value being below a certain value, or a string variable being equal to another string, is pervasive in Python.

## Conditional operators

Simple numerical variables can be compared with other values using built-in operators:

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equals
!=	Does not equal

```
>>> x = 5
>>> x > 4
True
>>> x == 4
False
```

The latter two could also be used for strings:

```
>>> z = 'Joe'
>>> z == 'Peter'
False
```

## Vectorised conditional operators

These conditional operators work on ndarrays too, in a vectorised fashion.

```
>>> x
array([1,2,3,4,5,6])
>>> x > 4
array([False,False,False,
       False,True,True])
```

## Multiple Boolean conditions

Python has built-in operators for combining conditions:

```
>>> x = 5
>>> x < 10 and x > 4
True
>>> True and False
False
```

c1 and c2	True if both are True
c1 or c2	True if at least one is True

## Multiple Boolean conditions

In NumPy and Pandas, Python's and/or operators no longer work: you need to use the following "bitwise" operators instead, as well as place brackets around each condition:

&	True if both are True
	True if at least one is True
^	Only one is True

```
>>> x
array([[1,2,3],
       [4,5,6]])
>>> (x!=3)&(x!=4)
array([[True,True,False],
       [False, True, True]])
```

Another useful operator, provided by the NumPy function `.isin`, allows you to check whether the elements exist in another list or ndarray:

```
>>> np.isin(x,[1,2,5,6])
array([[True,True,False],
       [False, True, True]])
```

## Aggregating Booleans

Booleans are internally evaluated as a number - True as 1, False as 0. This means that you can use NumPy's `.sum` to find the number of True values in a Boolean vector, and `.mean` to find the percentage of True values. There are also two methods for checking if a value exists:

any	Is there any True value in it?
all	Are all values in it True?

```
>>> np.any(x==5)
True
```

## Vectorised if-then-else

If you wish to change the value of an ndarray or Series depending on whether it satisfies a condition, you can pass it to the NumPy `.where`:

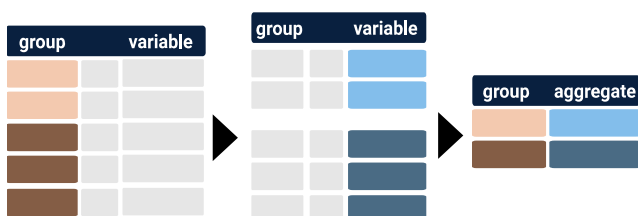
```
>>> x
array([-1, -4, 2, 3])
>>> np.where(x>0, 10, 20)
array([ 20, 20, 10, 10])
```

# Group aggregations in Python

We often wish to compare the sums or means between different groups. Such a **'group-wise aggregation'** requires us to group dataframe rows based on some group variable (for example, gender or country) and then applying a summary statistic to each group.

## Split-apply-combine

In the example below, a dataframe contains a categorical variable ('group') and a numeric variable ('variable'). We want to split rows based on their 'group' category and take each group's 'variable' values and aggregate them, say, calculating their means. This is known as a **'split-apply-combine'** operation. It can be thought of in terms of three steps: grouping, variable selection, and aggregation.



## Grouping & selecting

'Grouping' refers to splitting the rows into groups. The pandas method for grouping is **.groupby**. It works by creating a new intermediate object that internally groups the rows according to the column(s) passed as argument.

Consider the dataframe **df** with two grouping variables, 'A' and 'B'. To group simply by 'A', pass the column name as a string. This outputs a string informing you it is a groupby object:

A	B	X	Y
A1	B1	5	3
A1	B2	4	6
A2	B1	5	5
A2	B2	3	1

df

```
>>> df.groupby('A')
<pandas.core.groupby.generic.DataFrameGroupBy...>
```

You can select a column to aggregate, say 'X', using the simple dot:

```
>>> df.groupby('A').X
<pandas.core.groupby.generic.SeriesGroupBy...>
```

## Aggregating

Having accessed a column, you could use the built-in aggregation methods, like **.sum**:

```
>>> df.groupby('A').X.sum()
A
A1      9
A2      8
Name: X, dtype: int64
```

This results in a Series whose rows are given the names of the groups. You could turn this into a DataFrame using **.reset\_index**:

A	
A1	9
A2	8

```
>>> df.groupby('A').X.sum()\
.reset_index()
```

	A	X
0	A1	9
1	A2	8

Alternatively, you can use the **as\_index** argument set to False to the same effect:

```
>>> df.groupby('A',
               set_index = False) \
      .agg('sum')
```

The DataFrame now turns the row-names into a column, and the aggregates are given the column name of the column it aggregated.

A	X
A1	9
A2	8

Other aggregation methods include:

count	Number of non-NA values
min, max	Minimum, max value
sum	Sum of values
mean	Mean of values
prod	Product of values
median	Median of values
std	Standard deviation of values
var	Variance of values

Another way of aggregating is to pass the method name as a string to the function **.agg**:

```
>>> df.groupby('A').X \
      .agg('sum')
```

## Other scenarios

### Grouping by several columns

We could group by several columns, say both 'X' and 'Y', by passing them as a list:

```
>>> df.groupby(['A', 'B'])
```

A	B		A	B	X
'A1'	'B1'	5	'A1'	'B1'	5
'A1'	'B2'	4	'A1'	'B2'	4
'A2'	'B1'	5	'A2'	'B1'	5
'A2'	'B2'	3	'A2'	'B2'	3

Once aggregated, this would again return a Series with hierarchical indexing, but you could always use `.reset_index`.

### Aggregating several columns

We could also aggregate several columns, using the regular indexing operator with a list:

```
>>> df.groupby('A')[['X', 'Y']]
```

We can also use the slicing operator, numerical indexing, and all the other ways of selecting columns. An aggregation function will by default be applied to all those columns.

A	X	Y
'A1'	9	9
'A2'	8	6

### Multiple aggregation functions

If you want to apply multiple aggregation functions to all columns, then you can pass those names as a list to `.agg`:

```
>>> df.groupby('A').X \
    .agg(['mean', 'std'])
```

If you want to use different functions for different columns, you can pass a dictionary:

```
>>> df.groupby('A')[['X', 'Y']]
    .agg({'X': ['mean', 'std'],
          'Y': 'sum'})
```

### Aggregating all functions

You can choose to select no variables, in which case all non-grouping variables will be aggregated. Here both X and Y will be summed:

```
>>> df.groupby('A').sum()
```

### NumPy functions

NumPy functions work on Series, and can therefore be invoked inside the `.agg`, but remember to access it via the library variable and to not add quotation marks:

```
>>> df.groupby('A').X \
    .agg(np.max)
```

### Custom functions

You can pass a self-defined function into agg function, assuming that the function aggregates a Series into a number. Here we define one using the `lambda` syntax. If your function accepts an argument, you can add those after the function name:

```
>>> fun = lambda x,y: \
    x.sum()/y
>>> df.groupby('A').Y \
    .agg(fun,y)
```

### Renaming columns

You can always give the aggregate columns more descriptive names by passing a list of names to `.columns`:

```
>>> df.columns = ['gr', 'Xsum']
```

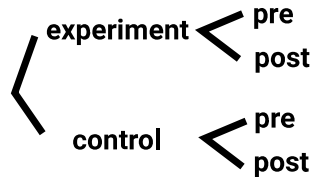
Another alternative is to pass tuples (nested inside a list) to a list inside `.agg` so as to map a label to the corresponding function.

```
>>> df.groupby('A').Y \
    .agg([('Ysum', sum),
          ('Ymean', mean)])
```

Also check out `.apply` for more even more flexible aggregations.

# Pivoting in Python

Suppose every row represents a person, and there are two conditions: an experimental and control condition. You take a measure prior and after each condition. Conceptually, you have a categorical variable that is nested:



This hierarchical structure is often implicit, inferred from column-names:

pre_exp	post_exp	pre_con	post_con

df

Pandas allows you to make this hierarchy explicit, by using an **Multindex** object as your index. This is called **hierarchical indexing**.

experiment		control	
pre	post	pre	post

df

## Hierarchical columns

A hierarchical DataFrame prints as follows:

```
>>> df
condition exper.      control
time      pre post      pre post
0         1   5         3   1
1         2   6         4   1
2         3   7         5   1
3         3   4         5   6
```

You do this by assigning a nested list and, optionally, give the different levels names:

```
>>> df.columns=[['experimental',\
'experimental','control','control'],
['pre','post','pre','post']]
>>> df.columns.names=['condition','time']
```

This allows you to access things based on the highest level:

```
>>> df['control']
      pre  post
0       3    1
1       4    1
2       5    1
3       5    6
```

You can add the indices on lower levels too, but always make sure you proceed top-down:

```
>>> df['control', 'pre']
      pre
0       3
1       4
2       5
3       5
```

## Hierarchical rows

DataFrames are, on a conceptual level, fairly symmetrical. You could store hierarchical information in the row index too. We could store each person's identifier not through the default RangeIndex (0...N-1), but rather as the top level in the index, with condition as the second level.

		pre	post
0	exp		
	con		
1	exp		
	con		
2	exp		
	con		
3	exp		
	con		

df

The index is set in the same way and appears like this:

```
>>> df
time      pre  post
person condition
0      exp    1    5
      con    3    1
1      exp    2    6
      con    4    1
2      exp    3    7
      con    5    1
3      exp    3    4
```

## Moving index to columns

If you have a DataFrame (or Series) with hierarchical index, you can use `.reset_index` to turn those into columns:

```
>>> df.reset_index()
```

				level_0	level_1	
0	exp	pre		0	exp	pre
		post		1	exp	post
1	con	pre		2	con	pre
		post		3	con	post
2	exp	pre		4	exp	pre
		post		5	exp	post
3	con	pre		6	con	pre
		post		7	con	post
4	exp	pre		8	exp	pre
		post		9	exp	post
5	con	pre		10	con	pre
		post		11	con	post
6	exp	pre		12	exp	pre
		post		13	exp	post
7	con	pre		14	con	pre
		post		15	con	post

df

df

This is useful for handling the results of group-by-aggregations, which return a hierarchically indexed Series.

## Moving columns to index

Conversely, you can turn columns into index, use the `.set_index` function with column names:

```
>>> df.set_index(['level_0', 'level_1'])
```

## Pivoting DataFrames

Suppose you have different categories of a categorical variable, for example A and B, represented as two columns. This is known as wide format. The actual measurements for a category are the values in its column.

We may wish to convert this into long format, where the categorical variable is a column (say, "Var") and serves to indicate whether a row (and the measurement inside "Val") represents an A-measurement or a B-measurement. One sample may thus spread across several rows, as indicated by ID.

ID	A	B
1	9	6
2	5	7

df

ID	Var	Val
1	A	9
1	B	6
2	A	5
2	B	7

df

## If all columns are categories

If you already have a wide DataFrame where the variable indicating sample ID is stored as index, and all the columns represent categories or sub-categories (with index serving as identifier), you can use `.stack` to pivot it into a hierarchically indexed Series.

A	
A1	A2

→

A
A1
A2
A1
A2
A1
A2

```
>>> df.stack()
```

This can be reversed, pivoting long to wide, using `.unstack`:

```
>>> df.unstack()
```

## If one column is an identifier

If you have an ID column (instead of index), then you can specify which columns to pivot using the function `.melt`. The only required argument is that of `id_vars`.

ID	A1	A2

→

ID	Var	Value
	A1	
	A2	
	A1	
	A2	
	A1	
	A2	
	A1	
	A2	

```
>>> df.melt(id_vars=['ID'],
            value_vars=['A1', 'A2'],
            var_name='Var',
            value_name='Value')
```

The opposite of this is :

```
>>> df.pivot(index='ID',
             columns='Var',
             values='Value')
```

The dataframe that results will store ID as an index. To restore the original column, just use `.reset_index` at the end.

For more options of pivoting, for example inferring a hierarchical index from column names, explore the pandas function `.wide_to_long`.