CHBE 230 - Computational Methods with Python

Tutorial 1

Python Crash Course I

Instructor:
Arman Seyed-Ahmadi

What We Will Learn Today

Today, we are going to learn about:

- The Python language and Jupyter Lab
- Basic math operations
- Variables and data types
- Basic output with formatted strings
- Lists and tuples, as well as indexing and slicing them
- The Numpy package and how to works with arrays.

Where should I write my code?

- As soon as you have the Python interpreter installed, you can use any text editor you want to create your Python programs (that is to say, Notepad is fine, but not recommended at all).
- However, there are specialized text editors that make life much easier for you!
- **IDE**s (**I**ntegrated **D**evelopment **E**nvironments) are a group of software particularly suited for program development. They have a text editor, a console for the interpreter, debugging capabilities, and a lot of other useful features.
- Examples include: Spyder, PyCharm, Visual Studio Code, etc.
- In this course, we will use **Jupyter Lab Notebooks** to write and run our programs.

Hello World!

- Here, we write our first program called “Hello World!”
- This program simply prints the phrase “Hello World!” to the output.
- It is believed that writing and running the “Hello World!” program successfully the first time will bring you luck for all your future programming efforts ☺
- So let’s get started!

```
[]: print('Hello World!')
```

This tells you that we are typing the command in the IPython console, instead of writing it in a cell inside Jupyter Lab.

Numbers and Math

- You can think of the IPython console as advanced calculator.
- Operations such as **+ (addition), - (subtraction), / (division), * (multiplication) and ** (exponentiation)** can be done right away in the console.
- The order of operations is **PE(M&D)(A&S)** = **P**arentheses, **E**xponentiation, **M**ultiplication and **D**ivision, **A**ddition and **S**ubtraction.
- Augmented operators!
- The % sign gives you the remainder of the division of two numbers, e.g. a+%b gives you the remainder of a/b.
- We can also compare two values with ==, !=, <, >, <=, >=. The result of such operations is either `True` or `False`.

Variables

- Variables are places in the memory to store data.

```
[]: x = 5.2
```

The = is not equal sign, it tells Python to assign 5.2 to variable x.

- You can store almost "anything" in a variable (in Python) including numbers, characters. Variables can also store a list of numbers, characters, and so on.

```
[]: my_name = 'Arman'
```

- You can join strings by using the + sign:

```
[]: uni_name = 'University ' + 'of ' + 'British Columbia'
```

Variables

- The name of variables can contain **characters**, **numbers** and **underscores (_)**.
- Variable names CANNOT start with a number.
- Special characters are not allowed (#, %,!, ^, &, etc.) in naming variables.
- Variables names cannot be chosen from the following list. These are **reserved Python keywords** which are interpreted as having special meanings in Python:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Variable or Data Types

- Booleans: `True` or `False`
- Numbers:

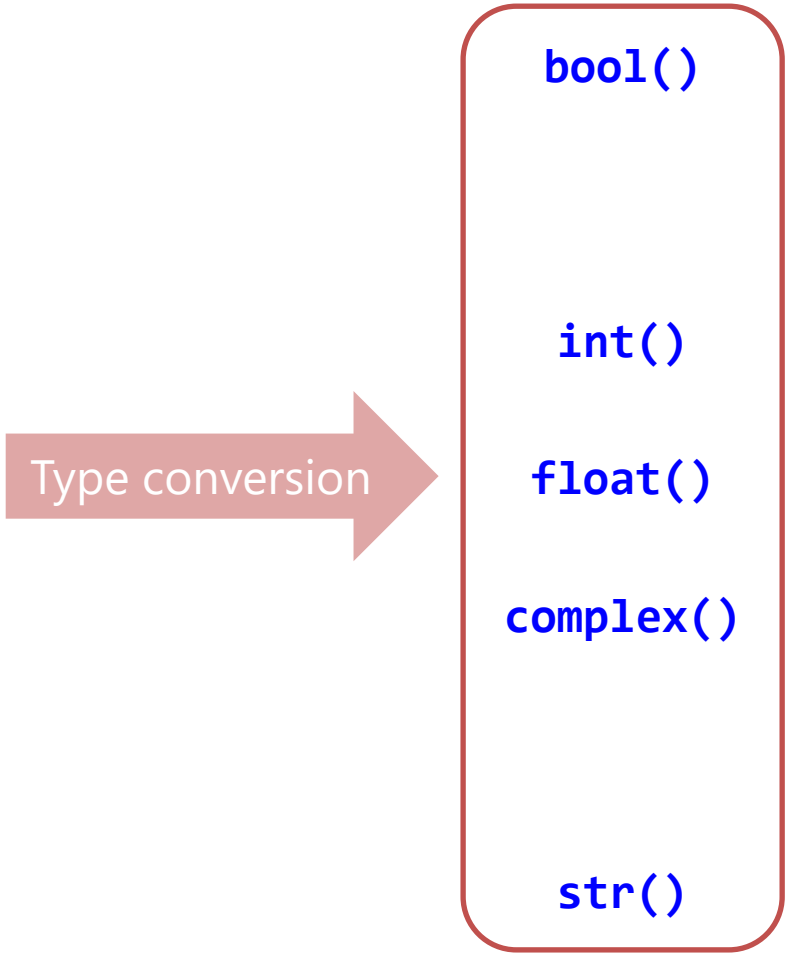
  Integers (`1, 108, -12`),

  floating-point numbers (`0.2, 3.14`) ,

  complex numbers (`1+3j, 3.5+12j`)

- Strings: Any piece of text:

  `'John', 'Python', 'Morning'`



- We can get see the data type of any variable or data by using the `type()` function in Python.

Let's Play with Text!

- Let's take a loot at the string variable below:

```
uni_name = 'University of British Columbia'
```

- What if we want to capitalize all the letters?

To do this, we can write:

```
uni_name.upper()
[]: 'UNIVERSITY OF BRITISH COLUMBIA'
```

Similarly:

```
uni_name.lower()
[]: 'university of british Columbia'
```

These little tools that help us work with these variables are called **Methods**.

These methods exist for other variable types as well. We'll talk more about them later!

It's Coding Time!

- Create two variables **a** and **b** and assign two numbers to them.

- Using the **print** function we learned before, display the phrase `'The sum of a and b is ='`, and in front of that, print **a**+**b**.

Your program probably looks like this:

```
[]: a = 65.2
[]: b = 72
[]: print('The sum of a and b is =', a + b)
```

- Can you write the same program a little differently?

- How many ways can you think of to do the same thing as this little program does?

Formatted Strings

- Try to display this in the output:

```
a = 65.2, b = 72, and the sum of a and b is 137.2
```

- You can do this by joining many pieces of text (and you still don't have any control on how the numbers are shown).

- **Formatted Strings:** An efficient way to embed variables (numbers or other strings) inside of a string.

```
a = 65.2
print(f'a is equal to {a}')
```

```
[]: a is equal to 65.2
```

The **f** before the quotation marks tells Python that this is a '**f**'**ormatted** string.

More Fun with Formatted Strings

- With formatted strings, not only you can easily place variables inside text, but also you can control the **format** of the shown variable.

- The pattern is:

```
f'{(variable_name):(width).(precision)(type)}'
```

- Example:

```
a = 3.1415926
print(f'a is {a}')
print(f'a is {a:2.3f}')
```

```
[]: a is 3.1415926
    a is 3.142
```

Formatting Table

Number	Format	Output	Description
3.1415926	{num:.2f}	3.14	2 decimal places
3.1415926	{num:+.2f}	+3.14	2 decimal places with sign
-1	{num:+.2f}	-1.00	2 decimal places with sign
2.71828	{num:.0f}	3	No decimal places
5	{num:0>2d}	05	Pad number with zeros (left padding, width 2)
5	{num:x<4d}	5xxx	Pad number with x's (right padding, width 4)
10	{num:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{num:,}	1,000,000	Number format with comma separator
0.25	{num:.2%}	25.00%	Format percentage
1000000000	{num:.2e}	1.00e+09	Exponent notation
13	{num:10d}	13	Right aligned (default, width 10)
13	{num:<10d}	13	Left aligned (width 10)
13	{num:^10d}	13	Center aligned (width 10)

Comments in Python

- Comments in Python usually start with a # sign, like the example below:

```
# This line is a comment, Python ignores this.
a = 65.2
print(f'a is equal to {a}')
```

```
[]: a is equal to 65.2
```

- We can also have **multi-line** comments. Instead of typing a # sign, we can use triple quotation marks:

```
'''
This is a long comment which can extend beyond one line
without typing # on every line.
'''
```

- Note that we can use {} for formatted strings inside of multi-line comments too!

Bags of Data in Python

- Let's assume that we want to save the x coordinates of a number of points.

- How would you do it? Separate variables for each x?

- We can store a bunch of numbers or strings inside an object, which is like a bag:

```
numbers = [2.5, 1, 2, -110, 2]
names = ['Niagara', 'Rocky', 'Mercedes-Benz']
names_nums = ['Niagara', 'Rocky', 3.14, 9]
```

- Each of the above "bags" of data is called a **List**. Lists are **ordered** collections of numbers, strings, etc. (they can contain pretty much any object, we'll get back to this later).

- The `len()` function gives us the number of items in a list.

Indexing in Python

- So, how can we access different items in a list?

- Each item in a list has a unique **index** or simply a **number**, which shows the location of that item in the list.

```
numbers = [2.5, 1, 2, -110, 2]
numbers[3]
```

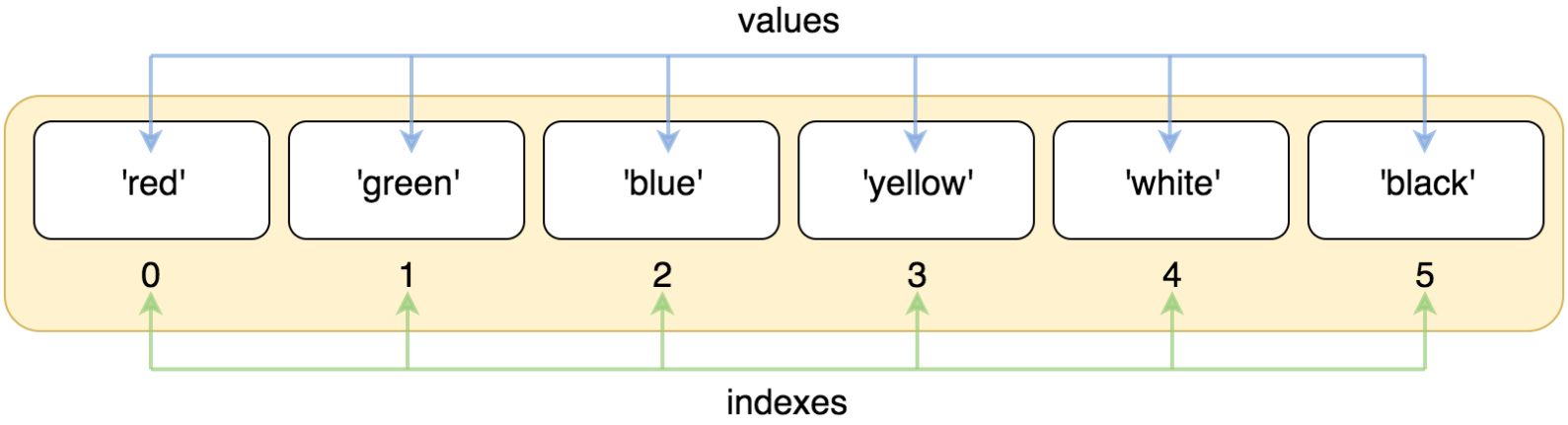
```
[]: -100
```

```
numbers[0]
```

```
[]: 2.5
```

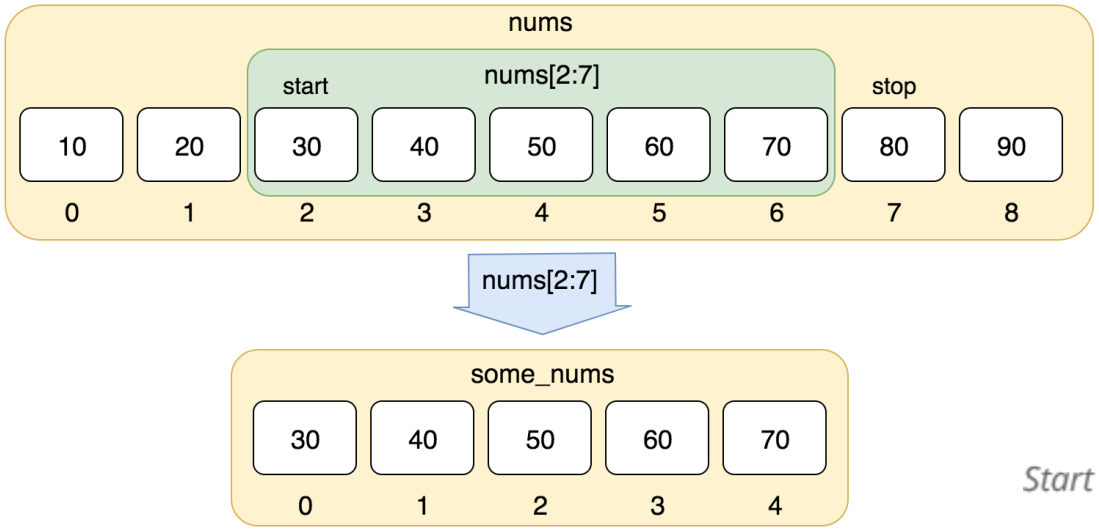
- As seen above, indices in Python start from **0**.

Indexing in Python



-index	-6	-5	-4	-3	-2	-1
list1	88	99	4.12	199	993	9999
index	0	1	2	3	4	5

Slicing Lists




```
L[start:stop:step]
```

Start position *End position* *The increment*

Delicious, right?

Just like slices of a big pizza, we can slice up a lists in Python!

Two Dimensional Lists

- Just as we can have lists of numbers or words, we can also define lists of other lists:

```
ones = [1, 1, 1, 1, 1, 1]
twos = [2, 2, 2, 2]
threes = [3, 3, 3, 3, 3]

numbers = [ones, twos, threes]
```

- Now, the `numbers` list is actually a **list of lists**, meaning that it contains three other lists.

- To access items in each sub-list, we just need to use the indexing notation twice:

```
numbers[0][3] = 1
```

- The sub-lists don't necessarily have to be the same size as each other.

Some List Methods

- Let's consider the list `numbers = [2.5, 1, 2, -110, 2]`
- `numbers.append(object)`: Lets you add an item to the end of a list.
- `numbers.sort()`: Sorts the items numerically and alphabetically (only if all items are either numeric or strings).
- `numbers.reverse()`: Reverses the order of the list items.
- `numbers.copy()`: Creates a copy of the list (Why does this method exist? hmmm…)
- `numbers.insert(index, object)`: Lets you insert an object in a specific index in the list.
- `numbers.count(object)`: Returns the number of *objects* in the list.

Removing Items From a List

- How to delete an item from a list?

```
numbers = [2.5, 1, 2, -110, 2]
del(numbers[3])
print(numbers)

[]: [2.5, 1, 2, 2]
```

- `numbers.remove(object)`: Removes the first time it sees *object*.
- `numbers.clear()`: Removes every item in the list. It is equivalent to `numbers = []`.
- `numbers.pop()`: Removes the last item in a list.

List Concatenation

- How can we join two lists together?

```
numbers = [2.5, 1, 2, -110, 2]
names = ['Niagara', 'Rocky']
new_list = numbers + names
print(new_list)
```

```
[]: [2.5, 1, 2, -110, 2, 'Niagara', 'Rocky']
```

The * is another supported operand for lists!

- Did you notice that we can use this method to add items to a list, just like the `numbers.append(object)` method?

- You have to be careful, though! The following will not work (why?):

```
new_list = numbers + 'Niagara'
```

```
[]: ERROR!
```

This + sign does not have the same meaning as in math!

Do you remember where we used the + for joining?

Tuples

- Tuples in Python are collections objects, just like lists. To define tuples, instead of using `[]`, we should use `()`:

```
a_tuple = (2.5, 1, 2, 'Niagara', 'Rocky')
```

- The main difference between a list and a tuple is that you can change the items in a list, remove items and change the order. This is not possible for tuples. In other words, lists are **mutable** and tuples are **immutable**:

```
del a_tuple[2]

[]: ERROR!
```

- Since tuples cannot be changed, it is faster for Python to go through its items. Also, they occupy less memory.

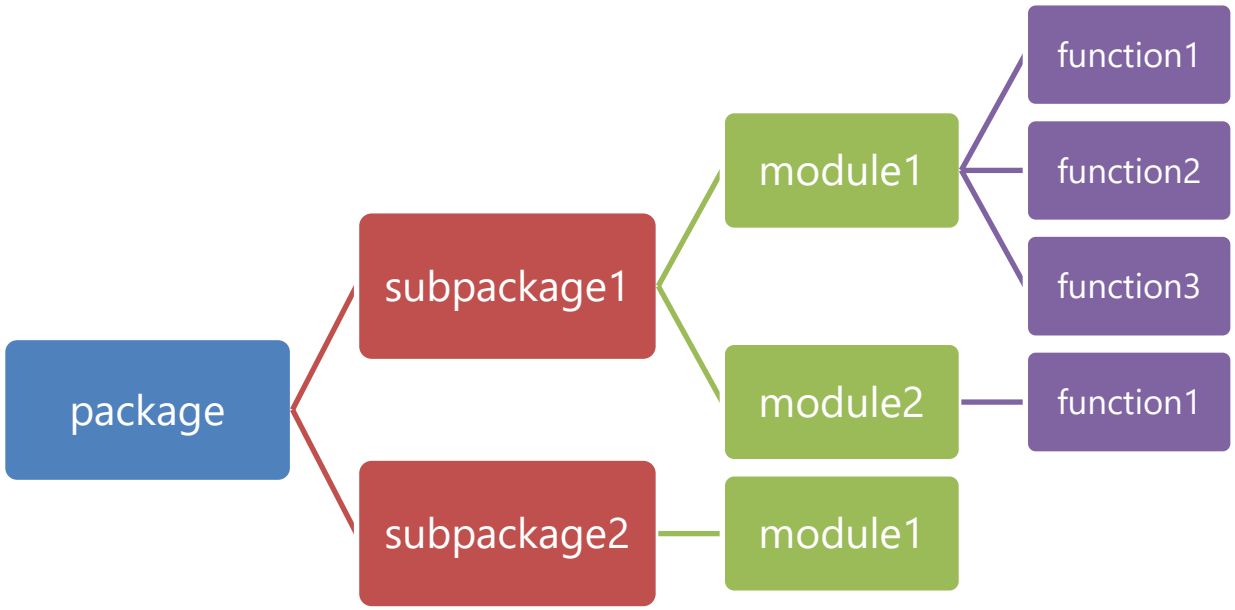
- Tuples have only two methods for them: `a_tuple.count()` and `a_tuple.index()`.

Packages and Modules

- We have seen functions such as `print()`, `type()` or `len()` so far. These are called Python's built-in functions.

- Many other functions that are not loaded by default. Also, there are a lot of external libraries we may want to use.

- In order to use other functions, we have to `import` them in our code.

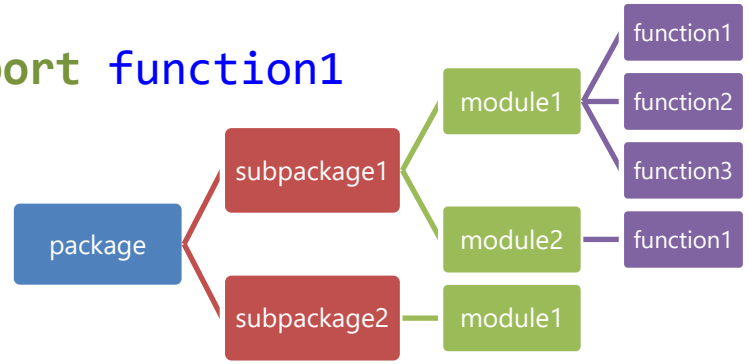


Packages and Modules

- The `import` command can take the following forms:

```
import package.subpackage1.module2
[]: package.subpackage1.function1()
```

```
from package.subpackage1.module2 import function1
[]: function1()
```




```
from package.subpackage1.module2 import *
[]: function1()
```

- Also, we can create a nickname for an imported package, module or function:

```
import package.subpackage1.module2 as mod2
[]: mod2.function1()
```

The Numpy Package

We can import the Numpy package with the following line:

```
import numpy as np
```

- The Numpy (or *Numeric Python*) package is the most important Python library for scientific computing.
- It provides the powerful **array** object, and a vast collection of mathematical tools to work with arrays (linear algebra, basic statistics, etc.)
- An array is basically a (multi-dimensional) grid of data of the same type.
- Most of the higher level packages (e.g. Pandas, Scikit-learn, Scipy, etc.) are built on top of Numpy.
- You remember Python is much slower compared to C++/Fortran… So, how do we deal with that?

Why not Using Lists Instead?

- **ndarray** in Numpy supports vectorized operations:

```
my_list = [2.5, 1, 2, -110, 2]
my_list + 25

[]: ERROR!
```

```
my_array = np.array([2.5, 1, 2, -110, 2])
my_array + 25

[]: array([ 27.5,  26. ,  27. , -85. ,  27. ])
```

- **ndarray** has consistent dimensions. You cannot have an array with columns or rows of different sizes.

- **ndarray** (most often) has consistent data-type, whereas you can store data of different types in a list.

- An array object occupies much less space in memory compared to a list.

Basics of Numpy Arrays

- Creating a one dimensional array (i.e., a vector) with custom values:

```
array_1d = np.array([2.5, 1, 2, -110, 2])
```

- Creating a two dimensional array (i.e., a matrix) with custom values:

```
array_2d = np.array([[2.5, 1, 2, -110, 2],
                     [0, 1, 0, 1, 1]])
```

```
[]: array([[   2.5,    1. ,    2. , -110. ,    2. ],
          [   0. ,    1. ,    0. ,    1. ,    1. ]])
```

- Size and shape of an array:

```
array_1d.size
[]: 5
```

```
array_2d.size
[]: 10
```

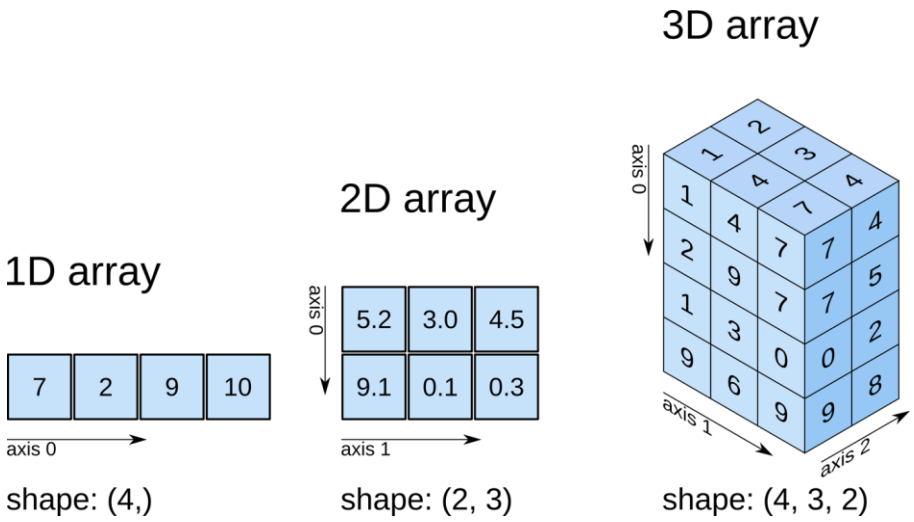
```
array_1d.shape
[]: (5,)
```

```
array_2d.shape
[]: (2,5)
```

Array Indexing & Slicing

- **Array indexing** is similar to **list indexing**.

- When we have more than one dimension, the preferred way is to use a single pair of `[ ]`, instead of `[][]` or more in lists:




```
array_2d[0, 3]
[]: -110
```

```
array_2d[0][3]
[]: -110
```

```
array_2d = np.array([
                     [2.5, 1, 2, -110, 2],
                      [0, 1, 0,1, 1]
                    ])
```

- Slicing arrays is also done in the same way:

```
array_2d[0, 2:4]    or  array_2d[0][2:4]
[]: array([2., -110.])
```

Functions for Creating Arrays

- It's neither easy nor efficient (and even impossible!) to create arrays manually. Numpy has many built-in functions for array generation.

- The two Numpy functions `np.arange(start, stop, step)` and `np.linspace(start, stop, numbers)` are used to create arrays with either a specified **step size** or a specified **number of values**.

- `np.zeros((N, M))` and `np.ones((N, M))` create arrays with shape `(N, M)` filled with zeros or ones.

- `np.eye(N)` creates the identity matrix of size `N`.

- `np.random.rand(N, M)` creates an array of shape `(N, M)` filled with random **real** numbers between 0 and 1.

- `np.random.randint(a, b, size=[N,M])` creates an array of shape `(N, M)` filled with random **integer** numbers between any `a` and `b`.

- There are also similar functions to generate random numbers with a certain probability distribution!

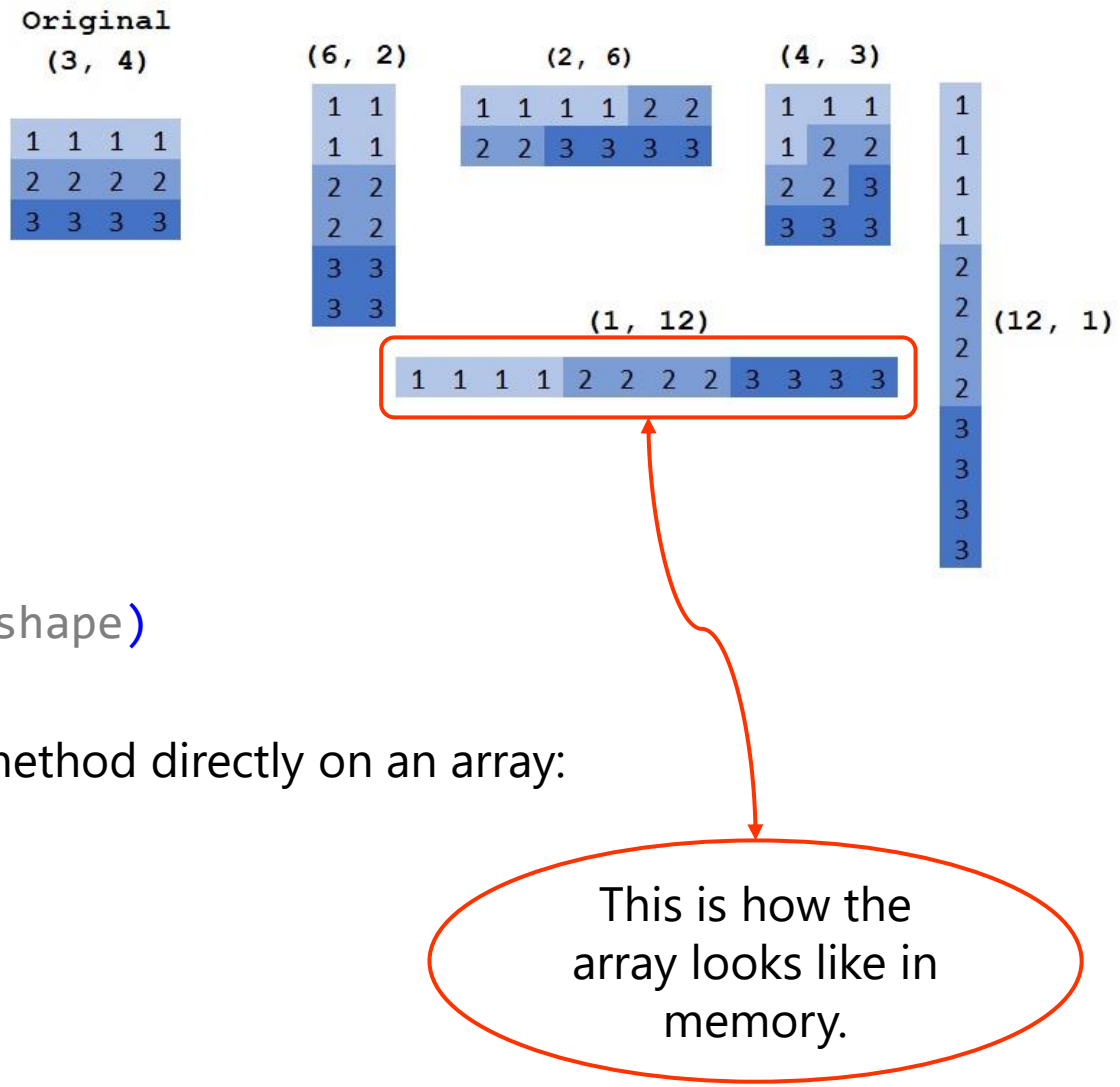
Basic Array Statistics

- `np.mean(array, axis=None)` computes the mean of all values in an array, or along a particular `axis`.
- `np.median(array, axis=None)` computes the median of all values in an array, or along a particular `axis`.
- `np.std(array, axis=None)` computes the standard deviation (a measure of the scatter in data) of all values in an array, or along a particular `axis`.
- `np.max(array, axis=None)` or `np.min(array, axis=None)` computes the min/max of all values in an array, or along a particular `axis`.
- We can also compute these values with **array methods** as `array_2d.mean()`, `array_2d.max()`, `array_2d.min()` and `array_2d.std()`.

Reshaping Arrays

- Array elements are stored **sequentially** in memory. The shape of an array is just a way that it is viewed.

- The reshape function provides a way to give a new shape to an array:



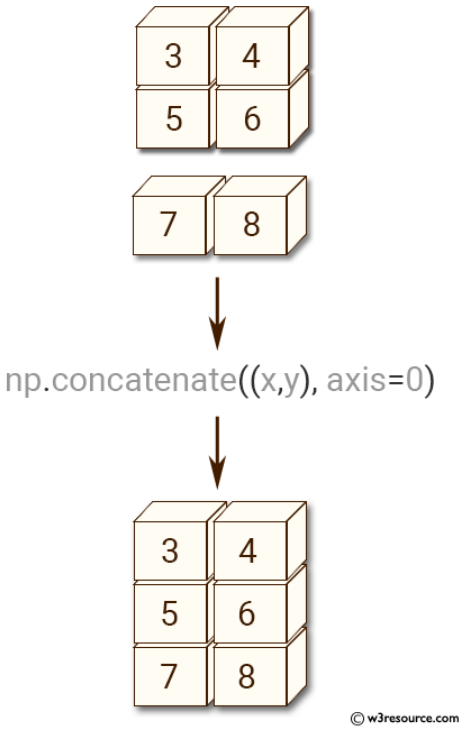

```
numpy.reshape(array, newshape)
```

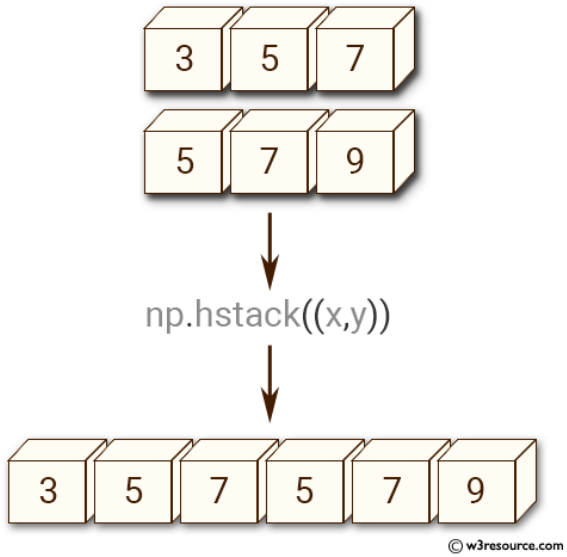
We can also use the reshape method directly on an array:

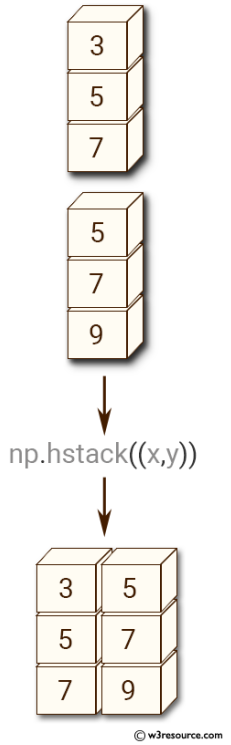
```
array.reshape(newshape)
```

Combining Arrays

- `np.concatenate((array1, array2), axis=0)`
- `np.hstack((array1, array2))`
- `np.vstack((array1, array2))`







More Numpy...

- Numpy provides the **vectorized** version of most math functions such as `np.sin()`, `np.cos()`, `np.tan()`, `np.exp()`, `np.log()`, `np.sqrt()` and many others. Vectorized functions operate directly on each element of an array. These are called **element-wise** operations.

- With Numpy arrays, operations such as +, -, *, / and ** are all **element-wise** operations by default.

- Slicing in numpy only returns a view of the original array. In order to explicitly copy an array, we have to use `array2 = array1.copy()`.

- To delete elements, rows or columns we can use `np.delete(arr, index, axis=None)`.

- To add elements to an array, we can use `numpy.append(arr, values, axis=None)` to add to the end of an array or `numpy.insert(arr, index, values, axis=None)`.

Summary

- Today, we have started to use **IPython** and **Jupyter Lab** to write our programs.
- We have learned a number of core concepts such as variables and how to work with them in Python.
- Many of the concepts you've learned today, such as variables and their different types, math operations and many other that you will learn in the coming sessions, are **common** between most programming languages.
- The differences are usually about syntax, and some other subtle things here and there that we talk about.
- There are also some other concepts that are exclusive to Python.

References

1. https://railsware.com/blog/python-for-machine-learning-indexing-and-slicing-for-lists-tuples-strings-and-other-sequential-types
2. https://www.datacamp.com/community/tutorials/python-numpy-tutorial
3. https://www.guru99.com/numpy-tutorial.html
4. https://medium.com/datadriveninvestor/artificial-intelligence-series-part-2-numpy-walkthrough-64461f26af4f
5. https://backtobazics.com/python/python-reshaping-numpy-array-examples/attachment/numpy-reshape-examples/
6. https://www.w3resource.com/numpy/