NumPy

Chantri Polprasert

Dept. of ICT,AIT

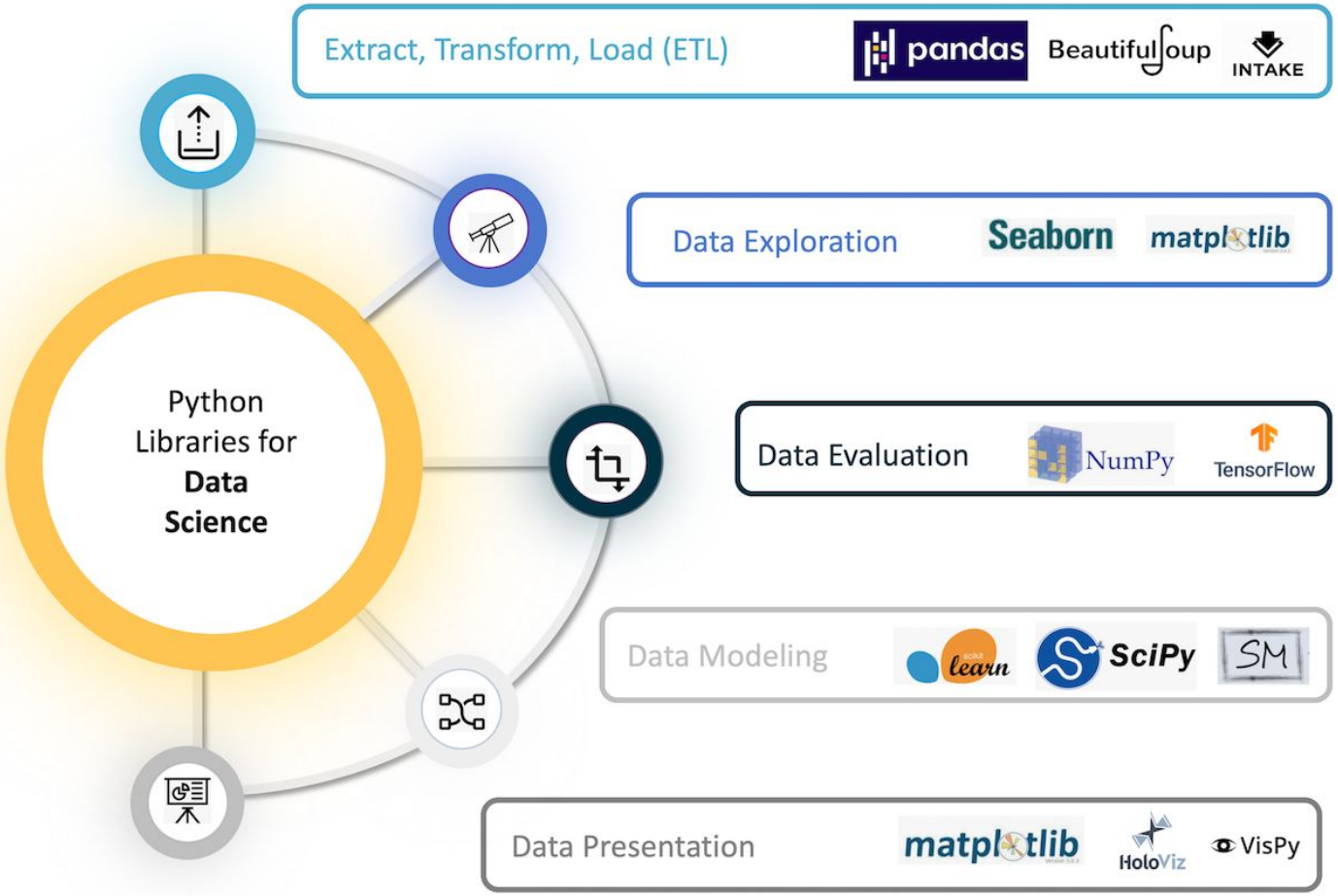
Agenda

- What is NumPy Array?
- Indexing and Accessing Numpy Array
- Basic operations on Numpy Array
- Broadcasting
- Mathematical and statistical functions on NumPy arrays

What is NumPy?

- “Numeric Python” or “Numerical Python”
  - Fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are the de-facto standards of array computing today.
- Powerful N-dimensional arrays
- Open Source
- Numerical computing tools
  - Offers comprehensive mathematical functions, random number generators, linear algebra routines, and more.
- Module of Python
- Interoperable
  - Supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries.
- Performant
  - well-optimized C code
- A part of Data Science and Machine Learning ecosystem

NumPy in Data Science and Machine Learning



NumPy lies at the core of a rich ecosystem of data science libraries. A typical exploratory data science workflow might look like:

- **Extract, Transform, Load:** Pandas, Intake, PyJanitor
- **Exploratory analysis:** Jupyter, Seaborn, Matplotlib, Altair
- **Model and evaluate:** scikit-learn, statsmodels, PyMC3, spaCy
- **Report in a dashboard:** Dash, Panel, Voila

Why do we need NumPy instead of list?

Calculating the BMI for an array of weight and height
BMI = weight(kg)/height(m)^2

```
In [13]: height = [1.73, 1.68, 1.71, 1.89, 1.79]

In [14]: weight = [65.4, 59.2, 63.6, 88.4, 68.7]

In [15]: weight / height ** 2
TypeError: unsupported operand type(s) for **: 'list' and 'int'
```

```
In [16]: np_height = np.array(height)

In [17]: np_weight = np.array(weight)

In [18]: np_weight / np_height ** 2
Out[18]: array([ 21.852,  20.975,  21.75 ,  24.747,  21.441])
```

Why do we need NumPy instead of list?

- List does slow numerical computations.
- For, 1000 x 1000 matrix multiplication
  - Python's triple loops takes around 2 minutes
  - Numpy takes ~0.01 seconds

```
def dotProduct(x,y):
  return [[sum(a*b for a,b in zip(x_row,y_col)) for y_col in zip(*y)] for x_row in x]
```

```
import numpy as np
A = np.random.randint(10, size=(1000, 1000))
B = np.random.randint(10, size=(1000, 1000))
x = A.tolist()
y = B.tolist()
```

```
import time
start = time.time()
dotProduct(x,y)
end = time.time()
print("----- Elapsed time--------")
time.strftime("%H:%M:%S", time.gmtime(end-start))
```

```
----- Elapsed time--------
'00:01:47'
```

```
import time
start = time.time()
np.dot(A,B)
end = time.time()
print("----- Elapsed time--------")
time.strftime("%H:%M:%S", time.gmtime(end-start))
```

```
----- Elapsed time--------
'00:00:01'
```

NumPy Array vs Python List

When array is preferred

- **Efficiency**
  - Array is more compact and consumes less memory than a list
  - Preferred when dealing with same type of data (less flexible)
- **Convenience**: built-in matrix operation
- **Functionality**: lots of built-in functions
- **Speed**: Fast since arrays are implemented in C (list uses built-in Python)

When list is preferred

- Dealing with small and different data types (more flexible)
- Not certain about the size of data (list can be resized)

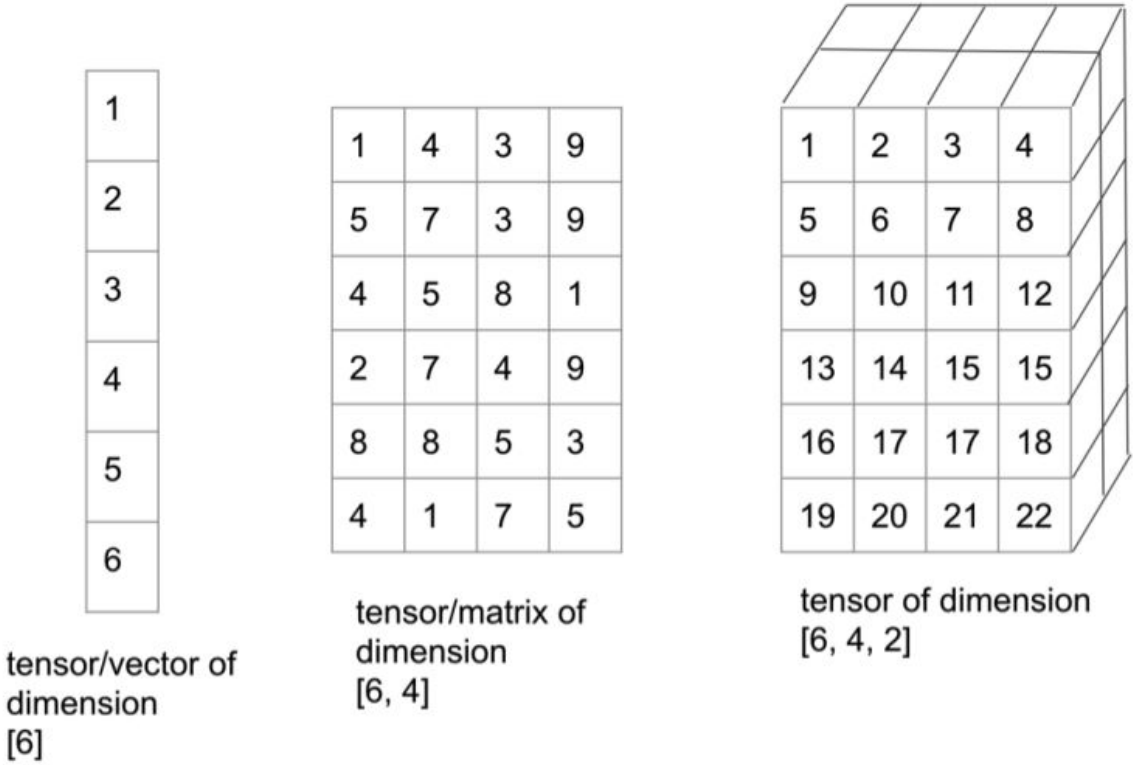
What do we need to be familiar with NumPy?

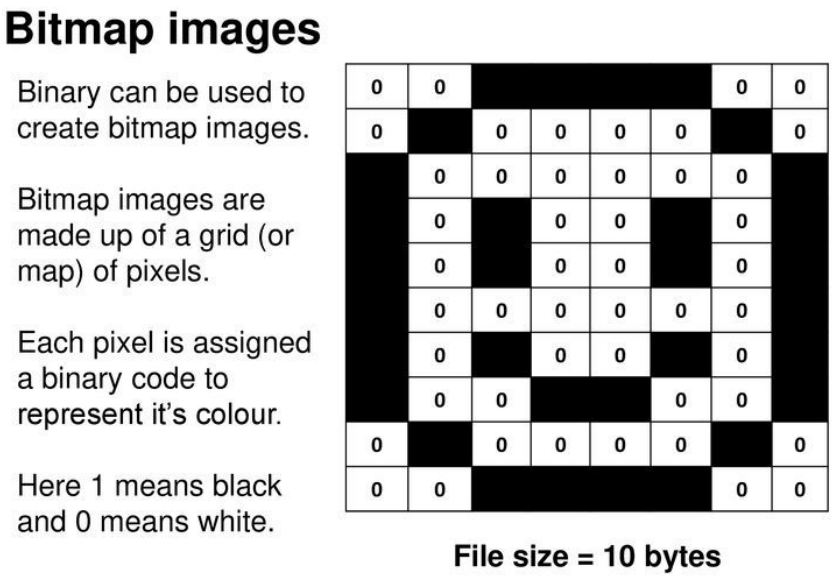
- **Create arrays** with NumPy, copy arrays, and divide arrays
- Exploit **array attribute**
- Perform different operations on NumPy arrays
- Understand array **selections, advanced indexing, and expanding**
- Working with multi-dimensional arrays
- Built-in NumPy functions

Arrays

A collection of data (mostly number)

- Vectors
- Matrices
- Tensors



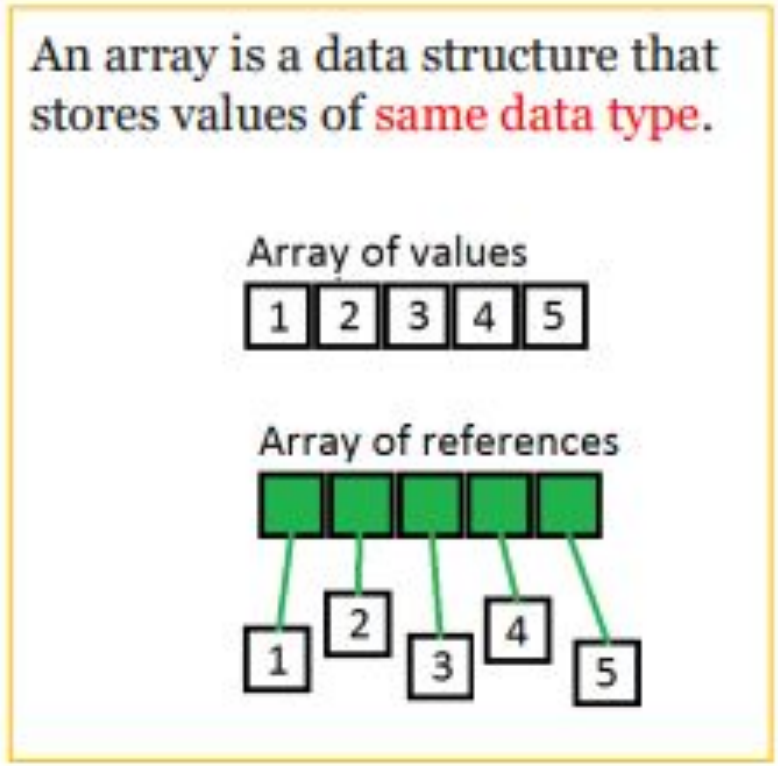


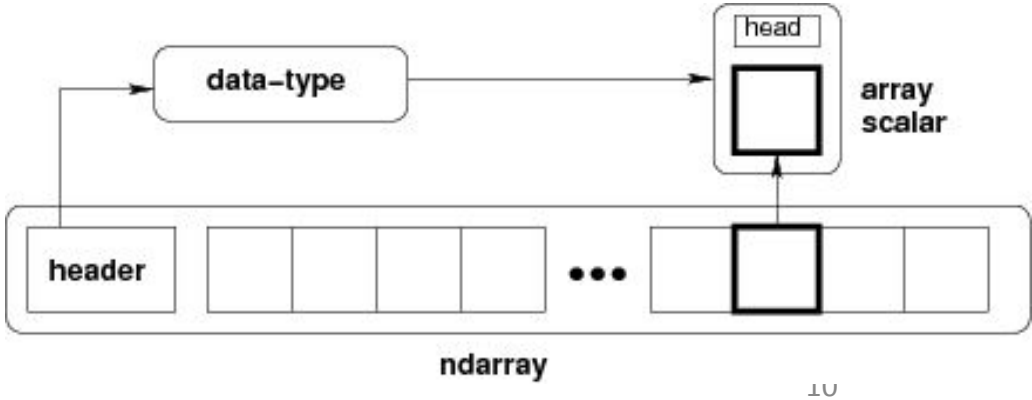
https://builtin.com/data-science/basic-linear-algebra-deep-learning

https://slideplayer.com/slide/14467802/

NumPy Introduction

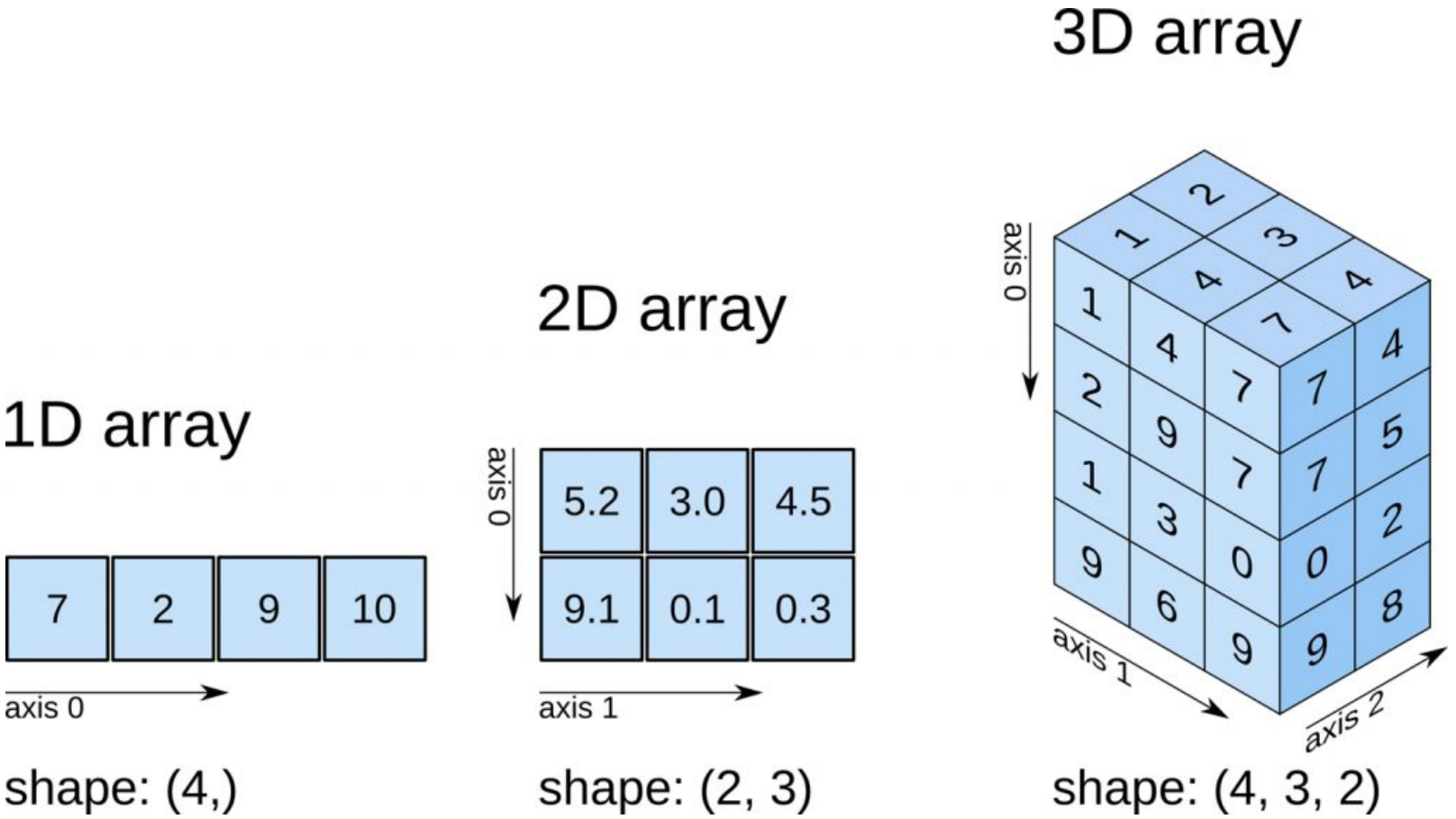
- NumPy provides an N-dimensional array type, the ndarray, which describes a collection of “items” of the same type.
- It is a table of elements
  - Usually, number
  - All of the same type (homogeneous)
  - Indexed by tuple of positive integers
- In NumPy dimensions are called axes (start from 0) e.g. (0 and 1 for a 2-dimensional array)
- The number of axes is rank (A simple list has rank 1: a 2-dimensional array (sometimes called a matrix) has rank 2)
- A data type object (an instance of numpy.dtype class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted.





https://docs.scipy.org/doc//numpy-1.10.4/reference/arrays.html

NumPy: ndarray



shape: (4,)

- **axis=0** (first axis) :horizontal direction

shape: (2, 3)

- **axis=0** (first axis) :vertical direction
- **axis=1** (second axis): horizontal direction.

shape: (4, 3, 2)

https://predictivehacks.com/tips-about-numpy-arrays/

Numpy Array Attributes

- ndim: the number of dimensions of an array is contained in the ndim attribute.
- shape: The shape of an array is a tuple of non-negative integers that specify the number of elements along each dimension.
- size: the fixed, total number of elements in array is contained in the size attribute
- dtype: Arrays are typically "homogeneous", meaning that they contain elements of only one "data type". The data type is recorded in the dtype attribute.

```
[13] x = np.array([[1.,2.],[3.,4.],[5.,6.]])

     x.ndim
     x.shape
     x.size
     x.dtype
```

Important NumPy attributes

Attributes	Description
ndim	returns number of dimensions (axes) of the array
shape	returns the dimension (size) of the array in each dimension.
size	returns number of elements in the array
dtype	returns data type of elements in the array
itemsize	returns the size (in bytes) of each elements in the array

```
x = np.array([[1,0,0],[0,1,2]])
print("The dimension of x is", x.ndim)
print("The shape of x is", x.shape)
print("The size of x is", x.size)
print("The data type of x is", x.dtype)
print("The size in byte of each x is", x.itemsize)
```

```
The dimension of x is 2
The shape of x is (2, 3)
The size of x is 6
The data type of x is int64
The size in byte of each x is 8
```

Creating NumPy array

There are 5 general mechanisms for creating arrays:

1. Conversion from other Python structures (e.g., lists, tuples)
2. Intrinsic numpy array array creation objects (e.g., arange, ones, zeros, etc.)
3. Reading arrays from disk, either from standard or custom formats
4. Use of special library functions (e.g., random)
5. Replicating, joining, or mutating existing arrays

1. Converting Python array_like Objects to Numpy Arrays

- np.array – Creating NumPy array from Python List/Tuple
- NumPy arrays can be created from Python lists or tuple in the following way.
-

```
1  import numpy as np
2  a = np.array([1,2,3])
3  print(type(a))
4
5  b = np.array((3, 4, 5))
6  print(type(b))
```

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

What will happen?

```
x = np.array([[1,2.0],[0,0],(1+1j,3.)])
```

2 Intrinsic Numpy Array Creation

Numpy has built-in function for creating array from scratch

- zeros(shape), ones(shape), arange(), linspace(), indices()
- np.zeros((2, 3)) #An array with all zeroes
- np.arange(5) #An array of a sequence of number [0,1,2,3,4]
- np.arange(2, 5, dtype=np.float32) #[start, stop), include start, **exclude** stop) ([2,3,4])
- np.arange(2, 3, 0.2) #(start, stop, step) ([2,2.2,2.4,2.6,2.8])
- np.linspace(1., 4., 6) #An array with evenly distributed numbers (start, stop, #points)
- np.indices(3,3)

Ranges

- A range is an array of numbers in increasing or decreasing order, each separated by a regular interval.
- Ranges are defined using the np.arange function, which takes either one, two, or three arguments: a start, and end, and a ‘step’.
- If you pass one argument to np.arange, this becomes the end value, with start=0, step=1 assumed. Two arguments give the start and end with step=1 assumed. Three arguments give the start, end and step explicitly.
- A range always includes its start value, but does not include its end value. It counts up by step, and it stops before it gets to the end.

2 Intrinsic Numpy Array Creation

```
#2 Intrinsic Numpy Array Creation
print('no.zeros((2,3)) = ',np.zeros((2,3)))
print('np.arange(10) = ', np.arange(10))
print('np.arange(2, 10, dtype=np.float32) = ', np.arange(2, 10, dtype=np.float32))
print('np.arange(2, 3, 0.1) = ', np.arange(2, 3, 0.1))
print('np.linspace(1., 4., 6) = ', np.linspace(1., 4., 6))
```

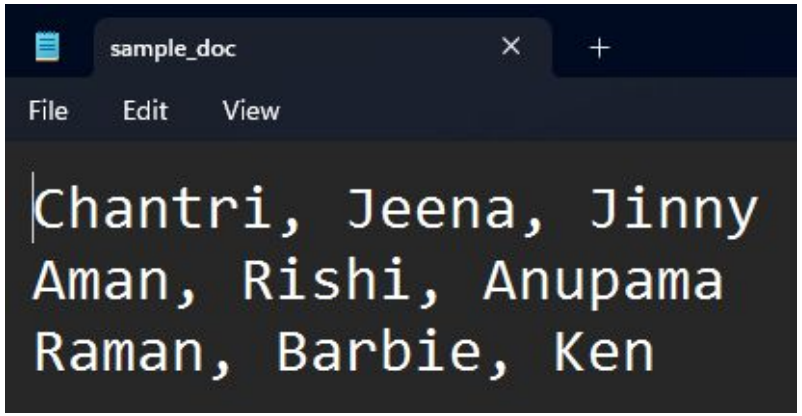
```
no.zeros((2,3)) =  [[0. 0. 0.]
 [0. 0. 0.]]
np.arange(10) =  [0 1 2 3 4 5 6 7 8 9]
np.arange(2, 10, dtype=np.float32) =  [2. 3. 4. 5. 6. 7. 8. 9.]
```

np.arange(2, 3, 0.1) = ?

np.arange(1., 4., .6) = ?

3. Reading arrays from disk, either from standard or custom formats

- HDF5, CSV, TXT
- loadtxt(), genfromtxt(), h5py()




```
arr = np.loadtxt("sample_doc.txt",
                 delimiter=",", dtype=str)
display(arr)
```

```
array([['Chantri', ' Jeena', ' Jinny'],
       ['Aman', ' Rishi', ' Anupama'],
       ['Raman', ' Barbie', ' Ken']], dtype='<U8')
```

Syntax: numpy.loadtxt()

Parameters:

- **fname**: The file name to load data from.
- **delimiter** (optional): Delimiter to consider while creating array of values from text, default is whitespace.
- **encoding** (optional): Encoding used to decode the inputfile.
- **dtype** (optional): Data type of the resulting array

Return: returns NumPy array

4. Use of special library functions (e.g., random)

- np.random.rand - Create an array with random numbers
- Make (2,3) matrix having random floats between 0 and 1:

```
x = np.random.rand(2,3)
x
```

```
array([[0.42888115, 0.84444963, 0.85473092],
       [0.65717383, 0.40661598, 0.59351114]])
```

5 Replicating, joining, or mutating existing arrays

- .copy()
- Will discuss more in Indexing and Accessing

```
a = np.array([1, 2, 3, 4])
b = a[:2].copy()
b += 1
print('a = ', a, 'b = ', b)
```

```
a =  [1 2 3 4] b =  [2 3]
```

NumPy Data Types (numpy.dtype)

Data types in Numpy

- Numerical
  - boolean (bool)
  - integer (int)
  - unsigned integer (unit)
  - floating point (float64) (64 bit floating point)
  - complex (complex)
- Strings and Bytes
  - unicode, byte sequences
- Structured datatypes: a collection of numpy datatype or subarray datatype
- Datetimes

Data types in Python

- String
- Integer
- Boolean
- Float
- Complex

Creating Arrays With Defined Data Types

```
int_ones = np.ones((2,2), dtype = np.int8)
print(int_ones)
print(int_ones.dtype)
```

```
[[1 1]
 [1 1]]
int8
```

```
#U is unicode string
#<U4 is a unicode that could vary in length between 1 and 4 bytes
string_arr = np.array(['Sam','Jim', 'Gary'])
print(string_arr)
string_arr.dtype
```

```
['Sam' 'Jim' 'Gary']
dtype('<U4')
```

What's the output?
How to fix this?

```
string_arr[2] = 'Harry'
print(string_arr)
```

Converting Data Type on Existing Arrays

The astype() method converts an array to a specified data type:

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

print("Original Array:", array)
print("Float Array:", array.astype(float))
print("Complex Array:", array.astype(complex))
print("Boolean Array:", array.astype(bool))
print("String Array:", array.astype(str))
```

```
Original Array: [0 1 2 3 4 5]
Float Array: [0. 1. 2. 3. 4. 5.]
Complex Array: [0.+0.j 1.+0.j 2.+0.j 3.+0.j 4.+0.j 5.+0.j]
Boolean Array: [False  True  True  True  True  True]
String Array: ['0' '1' '2' '3' '4' '5']
```

What's the output?

```
arr = np.array([1.1, 2.1, 3.1])
print('Int arr = ', arr.astype(int))
```

Reshaping Arrays

- The function reshape is used to reshape the numpy array.
- The following example illustrates this.

In [41]:

```
1  a = np.arange(6)
2  print(a)
3
4  print('-' * 10)
5
6  b = a.reshape(2, 3)
7  print(b)
```

```
[0 1 2 3 4 5]
----------
[[0 1 2]
 [3 4 5]]
```

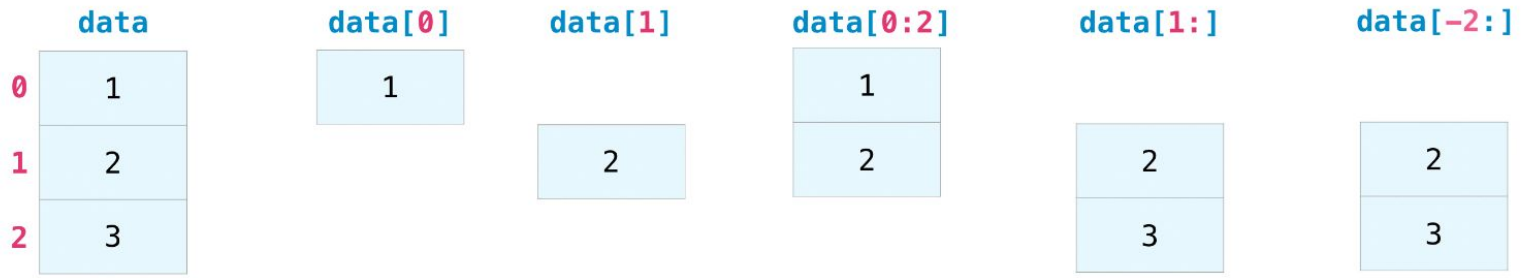
```
b = a.reshape(3,2)
print(b)
```

?

Indexing and Accessing NumPy Arrays

Indexing one dimensional NumPy Arrays

- **[i:j:k]** where **i** is the starting index, **j** is the stopping index, and **k** is the step (k not equal to 0) and **n** is the number of elements in the corresponding dimension
- Negative i and j are interpreted as n + i and n + j. Negative k makes stepping go towards smaller indices.
- If i is not given it defaults to 0 for k > 0 and n - 1 for k < 0.
- If j is not given it defaults to n for k > 0 and -1 for k < 0.
- If k is not given it defaults to 1.
- Note that :: is the same as : and means select all indices along this axis.



Indexing one dimensional NumPy Arrays

```
#index starts from zeros to len of array-1
#index         0 1 2 3 4 5 6 7 8 9
a = np.array([0,1,2,3,4,5,6,7,8,9])
print('a[1:7:2] = ', a[1:7:2]) #array([1, 3, 5])
print('a[-2:10] = ', a[-2:10]) #array([8, 9])
print('a[-3:3:-1] = ', a[-3:3:-1]) #array([7,6,5,4])
print('a[5:] = ', a[5:]) #array([5,6,7,8,9])
print('a[2::2] = ', a[2::2]) #?
print('a[::-1] = ', a[::-1]) #?
```

```
a[1:7:2] =  [1 3 5]
a[-2:10] =  [8 9]
a[-3:3:-1] =  [7 6 5 4]
a[5:] =  [5 6 7 8 9]
```

Difference with regular Python lists

1. If you assign a single value to a ndarray slice, it is copied across the whole slice :

In [47]:
```
1  # copy the assigned value across the whole slice
2  a = np.array([1, 2, 5, 7, 8])
3  a[1:3] = -1
4  a
```

Out[47]: array([1, -1, -1, 7, 8])

In [48]:
```
1  # cannot assign a value to list using the slice operato
2  b = [1, 2, 5, 7, 8]
3  b[1:3] = -1
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-48-6bc155ff552b> in <module>
      1 b = [1, 2, 5, 7, 8]
----> 2 b[1:3] = -1

TypeError: can only assign an iterable
```

Difference with regular Python lists

2. ndarray slices are **actually views on the same data buffer**. If you modify it, it is **going to modify the original** ndarray as well.

In [49]:

```
1  a = np.array([1, 2, 5, 7, 8])
2  print("original array:", a)
3  a_slice = a[1:5]
4  print(a_slice)
5  a_slice[1] = 1000
6  print(a_slice)
7  print("original array wah modified:", a)
```

```
original array: [1 2 5 7 8]
[2 5 7 8]
[   2 1000    7    8]
original array wah modified: [   1    2 1000    7    8]
```

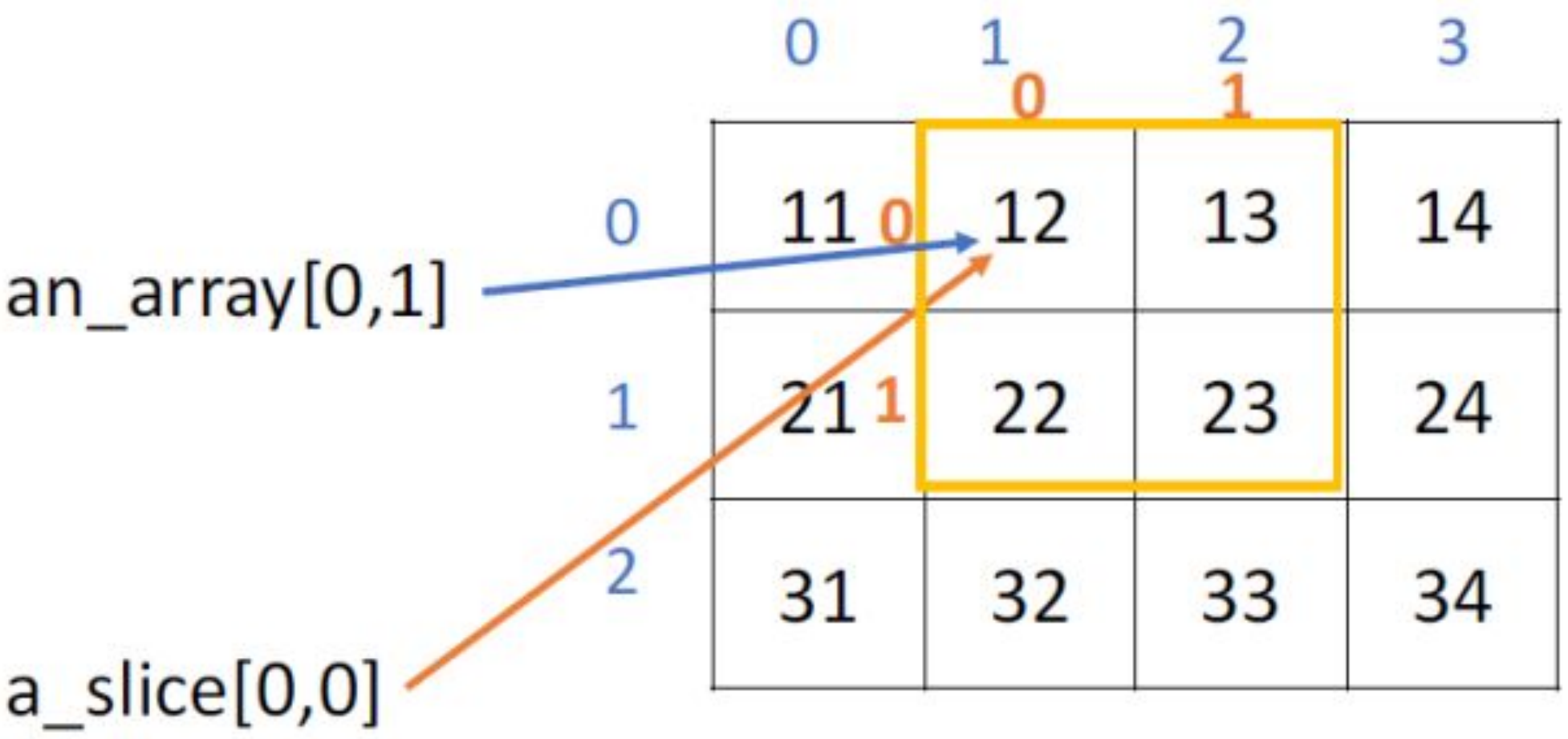
ndarray Indexing and Slicing

an_array =

	0	1	2	3
0	11	12	13	14
1	21	22	23	24
2	31	32	33	34

a_slice = an_array[:2, 1:3]

Sliced array has its own indices



Important attributes of a NumPy object

3. If you want a copy of the data, you need to **use the copy method.**

In [51]:

```
1  a = np.array([1, 2, 5, 7, 8])
2  print("original array:", a)
3  another_slice = a[1:5].copy()  # create new array with copy()
4  print(another_slice)
5  another_slice[1] = 1000
6  print(another_slice)
7  print("original array is not modified:", a)
```

```
original array: [1 2 5 7 8]
[2 5 7 8]
[   2 1000    7    8]
original array is not modified: [1 2 5 7 8]
```

if we modify another slice, a remains same.

Indexing multi-dimensional NumPy arrays

- Multi-dimensional arrays can be accessed as

- The following format is used while indexing multi-dimensional arrays

```
Array[row_start_index:row_end_index,
column_start_index:column_end_index]
```

```
In [65]:
1  b = np.arange(20)
2  b = b.reshape(4, 5)
3  b

Out[65]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14],
                [15, 16, 17, 18, 19]])
```

```
1  print(b[1,2]) #row 1, col 2
2  print(b[1,:]) # row 1, all columns
3  print(b[:, 1]) # all rows, column 1
```

```
7
[5 6 7 8 9]
[ 1  6 11 16]
```

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

How about b[1]?

Boolean indexing

- We can also index arrays using an ndarray of boolean values on one axis to specify the indices that we want to access.

```
# use boolean indexing to create a filter
a = np.arange(12).reshape(3, 4)
rows_on = np.array([ True, False, True])
print(a)
a[rows_on , : ] # Rows 0 and 2, all columns
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
```

```
cols_on = np.array([ True, False, False, True])
a[: , cols_on ] #
```

Basic Operations on NumPy Arrays

Arithmetic Operations

- Arithmetic operators on arrays apply elementwise.
- A new array is created and filled with the result.

```
a = np.array([[10,20],[30,40]])
b = np.array([[1,2],[3,4]])
print('a + b = ', np.add(a,b)) #a+b
print('a - b = ', np.subtract(a,b)) #a-b
print('a * b = ', np.multiply(a,b)) #a*b elementwise
print('a * b = ', np.dot(a,b)) #a*b
print('a */ b = ', np.divide(a,b)) #?
```

```
a + b =  [[11 22]
 [33 44]]
a - b =  [[ 9 18]
 [27 36]]
a * b =  [[ 10  40]
 [ 90 160]]
a * b =  [[ 70 100]
 [150 220]]
```

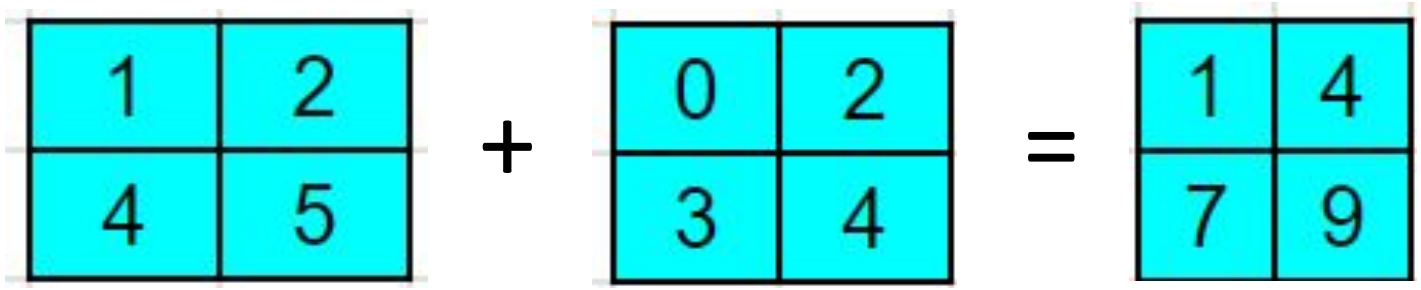
Conditional Operators on NumPy arrays

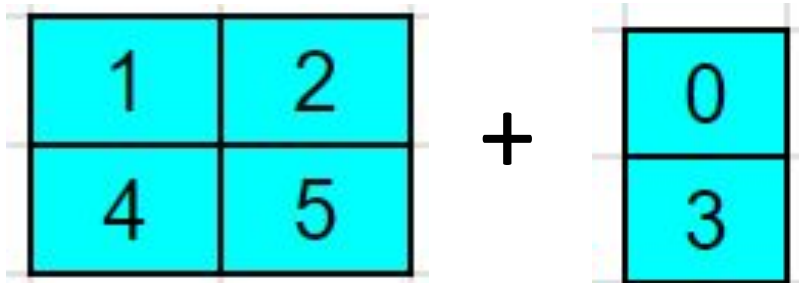
- Conditional operators are also applied element-wise

```
In [85]:
1  m = np.array([20, -5, 30, 40])
2  m < [15, 16, 35, 36]

Out[85]: array([False,  True,  True, False])

In [86]:
1  m < 25

Out[86]: array([ True,  True, False, False])

In [87]:
1  #To get the elements below 25
2  m[m < 25]     ?
```

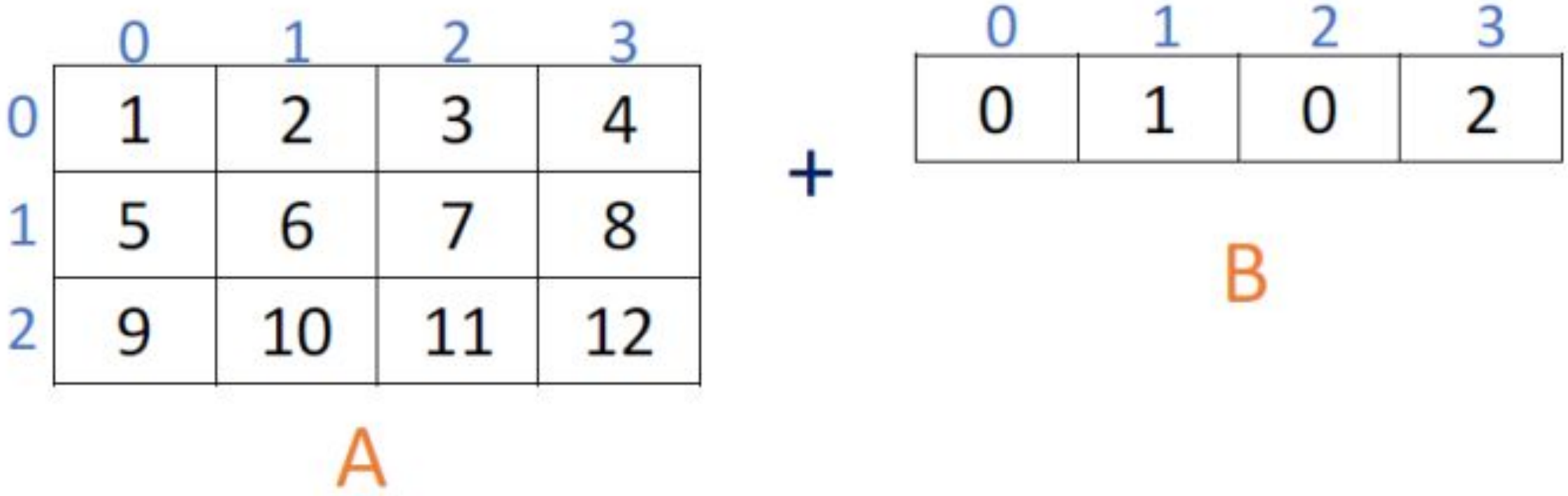
Broadcast in NumPy Arrays

What is Broadcasting?

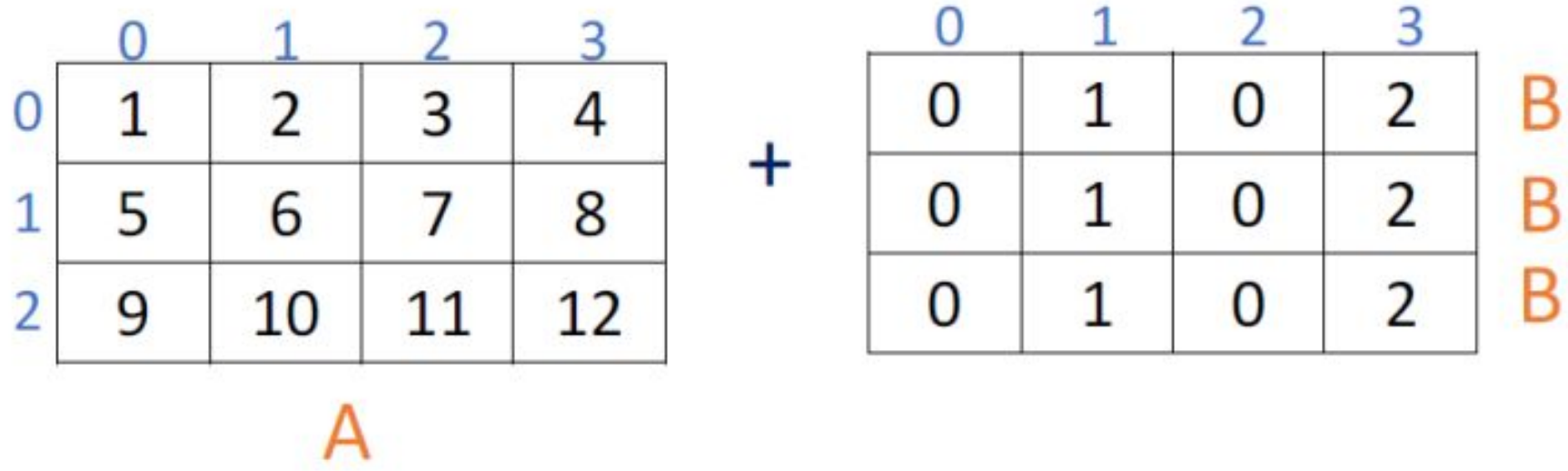




What is Broadcasting?



What is broadcasting?



What is broadcasting?

- Broadcasting describes how numpy treats arrays with different shapes during arithmetic operations

- The smaller array is “broadcast” across the larger array so that they have compatible shapes.

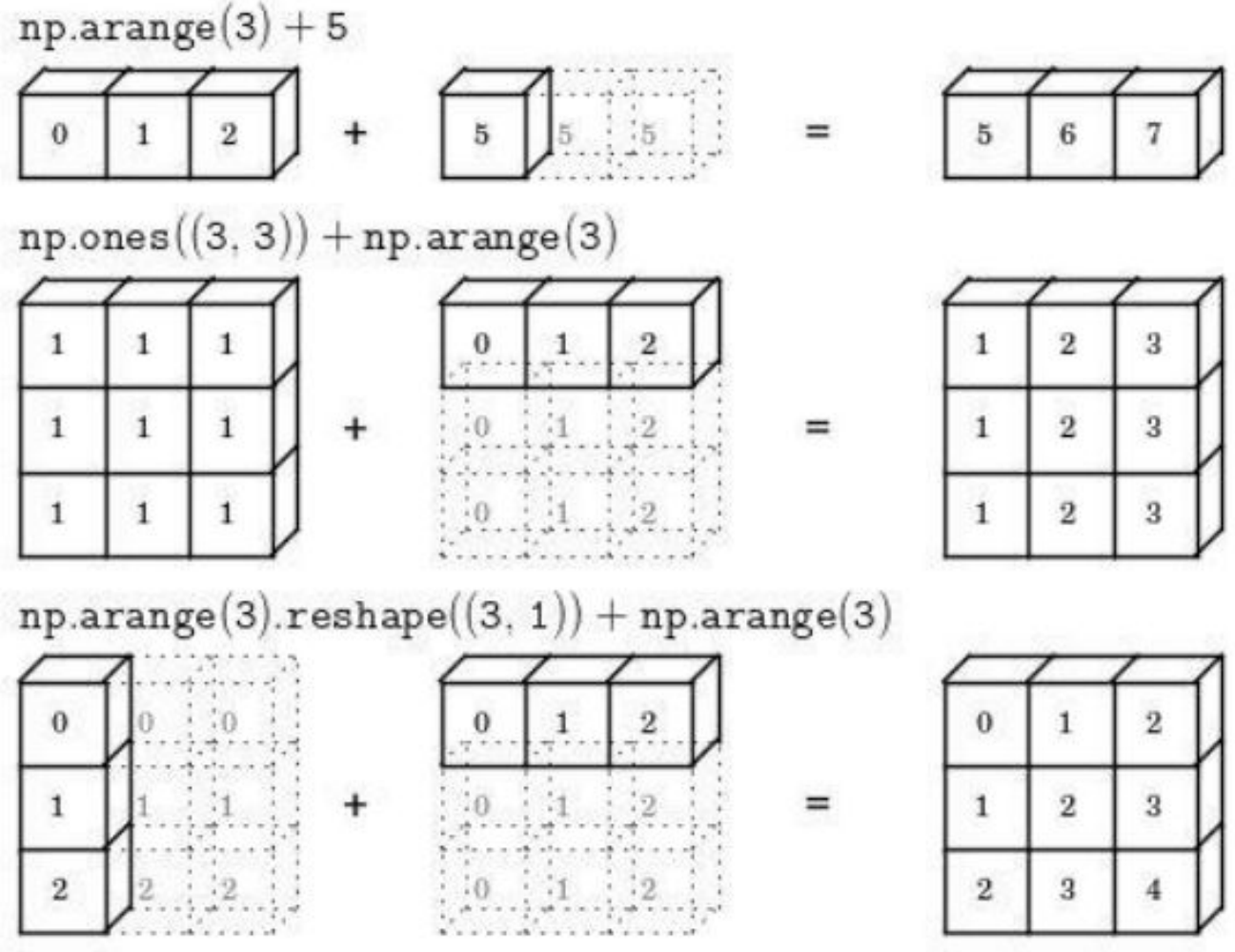
- Provides a means of vectorizing array operations so that looping occurs in C instead of Python

- When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when

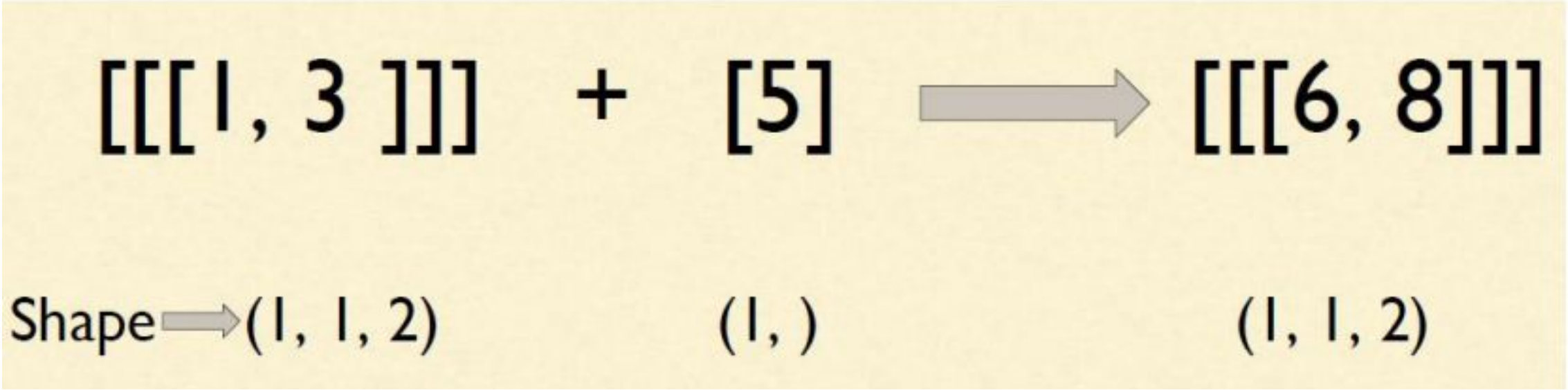
    1. they are equal, or
    2. one of them is 1

- **Note**: If not met, a ValueError: frames are not aligned exception is thrown

Broadcasting



Broadcasting Example1



np.array([[[1,3]]]) + np.array([5])

Broadcasting Example 2

```
In [89]:
1 h = np.arange(5).reshape(1, 1, 5)
2 h

Out[89]: array([[[0, 1, 2, 3, 4]]])
```

- Let's try to add a 1D array of shape (5,) to this 3D array of shape (1,1,5), applying the first rule of broadcasting.

```
In [90]:
1 h + [10, 20, 30, 40, 50] # same as: h + [[[10, 20, 30, 40, 50]]]

Out[90]: array([[[10, 21, 32, 43, 54]]])
```

Broadcasting Example 3

- On adding a 2D array of shape (2,1) to a 2D ndarray of shape (2, 3). NumPy will apply the second rule of broadcasting

```
In [108]:
1 k = np.arange(6).reshape(2, 3)
2 k

Out[108]: array([[0, 1, 2],
                 [3, 4, 5]])

In [109]:
1 k + np.arange(100,300,100).reshape(2,1)
```

Mathematical and statistical functions on NumPy arrays

Some useful ndarray methods

Sample aggregate functions (Useful to explore structured data)

- min - returns the minimum element in the ndarray
- max - returns the maximum element in the ndarray
- sum - returns the sum of the elements in the ndarray
- prod - returns the product of the elements in the ndarray
- std - returns the standard deviation of the elements in the ndarray.
- var - returns the variance of the elements in the ndarray.
- mean -returns the mean of elements in the ndarray

Pandas is the preferred choice when implementing complex data manipulation.

Some useful ndarray methods

In [95]:

```
1  a = np.array([[-2.5, 3.1, 7], [10, 11, 12]])
2  for func in (a.min, a.max, a.sum, a.prod, a.std, a.var):
3      print(func.__name__, "=", func())
```

```
min = -2.5
max = 12.0
sum = 40.6
prod = -71610.0
std = 5.084835843520964
var = 25.855555555555554
```

Summing across different axes

- We can sum across different axes of a numpy array by specifying the axis parameter of the sum function.

```
import numpy as np
a = np.arange(6).reshape(2,3)
a
```

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

```
print("a.sum(axis=0) = ", a.sum(axis=0))
print("a.sum(axis=1) = ", a.sum(axis=1))
```

```
a.sum(axis=0) =  [3 5 7]
a.sum(axis=1) =  [ 3 12]
```

Summing across different axes

- We can sum across different axes of a numpy array by specifying the axis parameter of the sum function.

```
] import numpy as np
  a = np.arange(12).reshape(2,2,3)
  a
```

```
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

```
a.sum(axis=0)
```

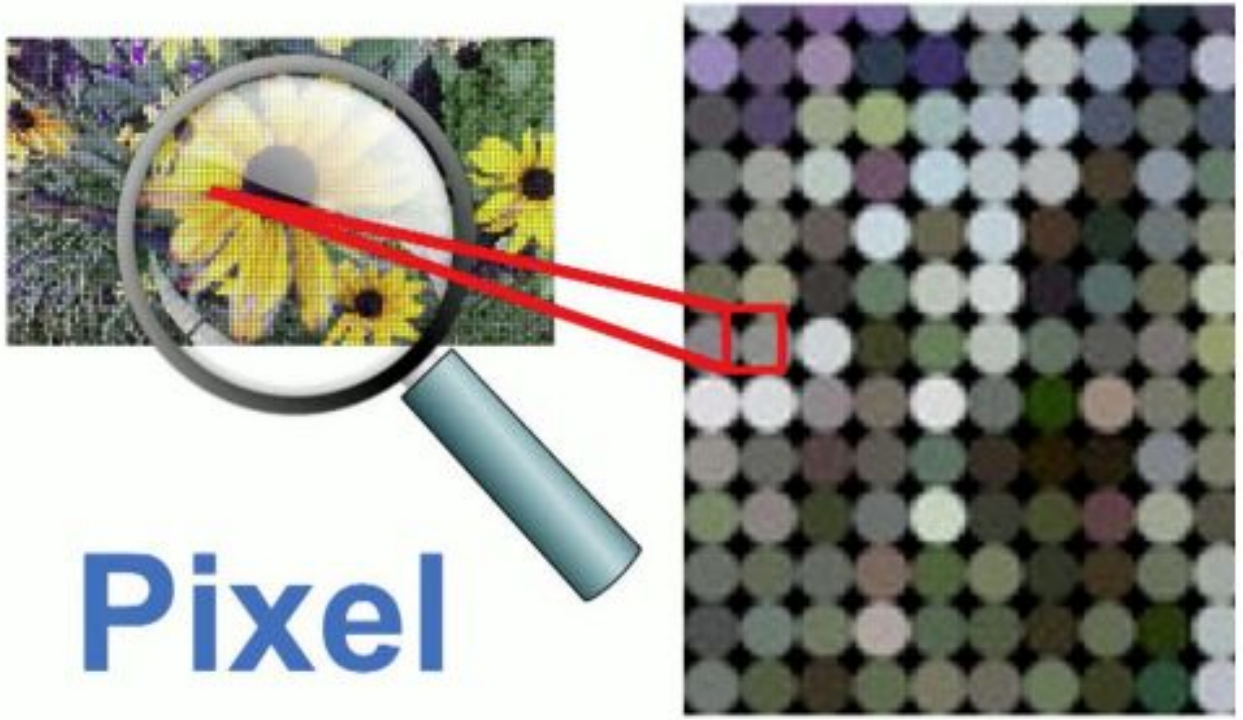
```
array([[ 6,  8, 10],
       [12, 14, 16]])
```

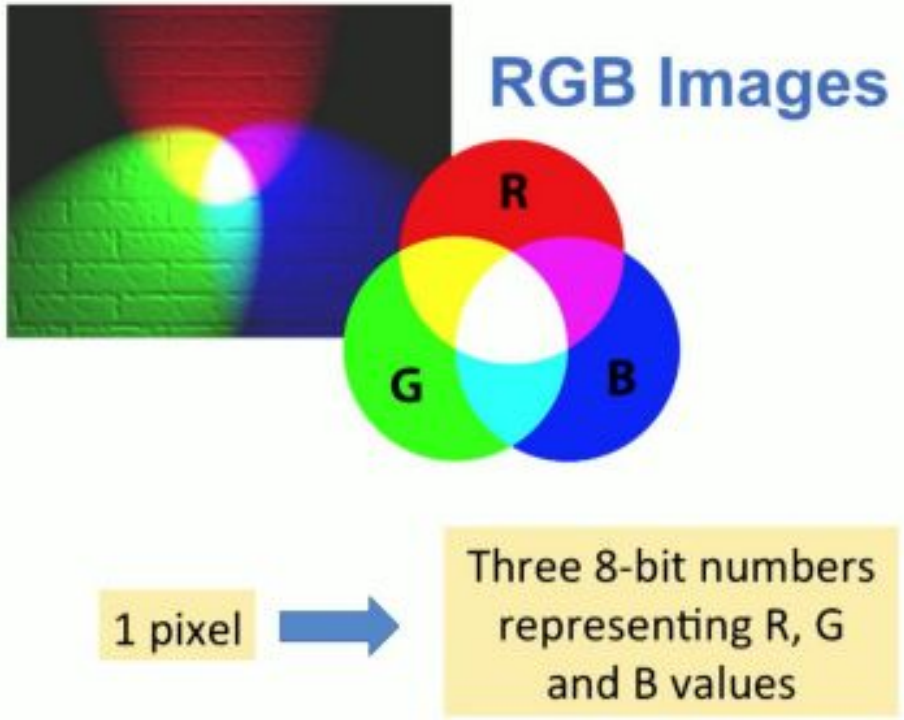
```
a.sum(axis=1)?
```

```
a.sum(axis=2) ?
```

NumPy for image processing

Pixel Image

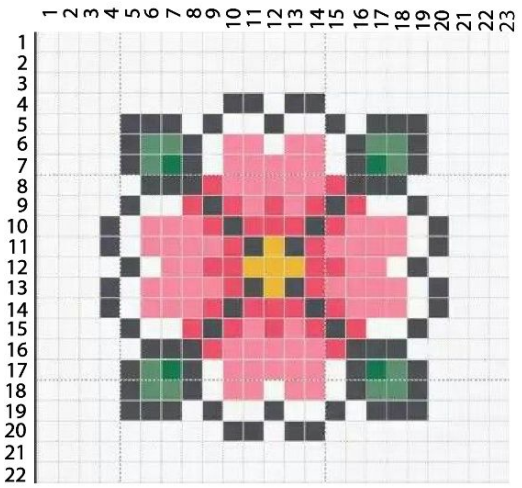

Pixel

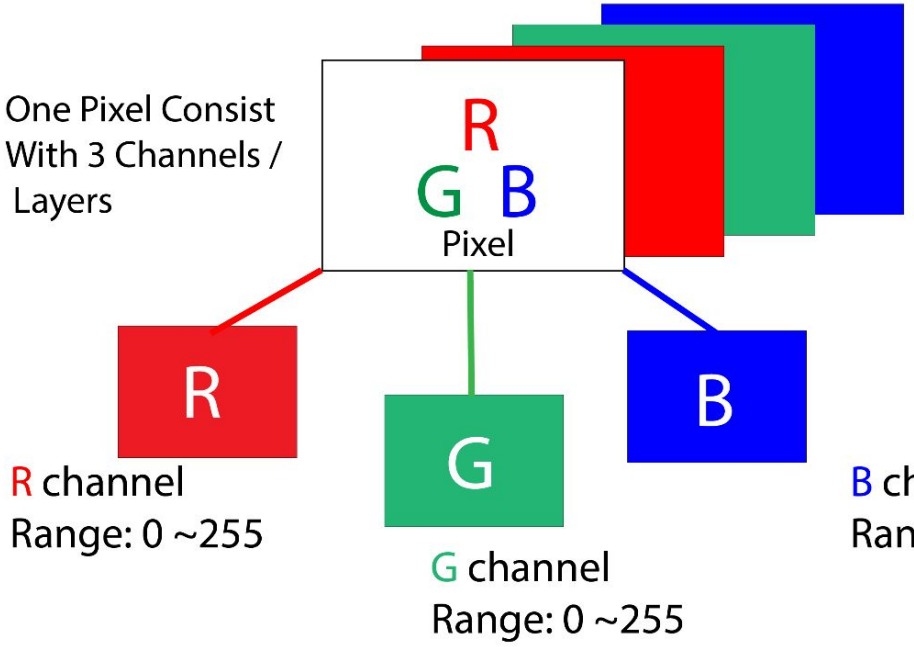

RGB Images
R
G
B
1 pixel
Three 8-bit numbers representing R, G and B values

Pixel Image

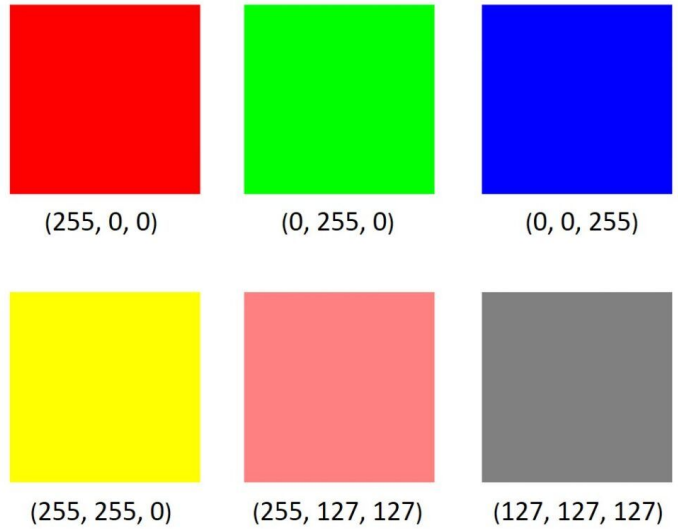
Each pixel coordinate (x, y) contains 3 values ranging for intensities of 0 to 255 (8-bit)
- Red - Green - Blue

Mixing different intensities of each color gives us the full color spectrum.





RGB model



RGB Images in Python

imageio

```python
import imageio.v3 as iio
im = iio.imread('imageio:chelsea.png')  # read a standard image
im.shape  # im is a NumPy array of shape (300, 451, 3)
iio.imwrite('chelsea.jpg', im)  # convert to jpg
```

- A Python library that provides an easy interface to read and write a wide range of image data, including animated images, video, volumetric data, and scientific formats.
- Simple interface via a concise set of functions
- Easy to install using Conda or pip
- Few dependencies (only NumPy and Pillow)
- Pure Python, runs on Python 3.8+, and PyPy
- Read/Write support for various resources (files, URLs, bytes, FileLike objects, ...)

- Some handful functions:
  - imread() - for reading
  - imwrite() - for writing
  - improps() - for standardized metadata
  - immeta() - for format-specific metadata
  - imopen() - for advanced usage

References

[1] https://docs.scipy.org/doc//numpy-1.10.4/user/basics.creation.html#arrays-creation

[2] S. Mongkonlaksami: DS511 Data Science, SWU

[3] https://www.geeksforgeeks.org/python-operations-on-numpy-arrays/

[4] https://numpy.org/devdocs/index.html

[5] https://www.codementor.io/@innat_2k14/image-data-analysis-using-numpy-opencv-part-1-kfadbafx6

[6] https://imageio.readthedocs.io/en/stable/index.html