Advanced

Lisa Schwetlick & Aina Frau-Pascual, ASPP 2022

ASPP 2022

MONDRAGON UNIBERTSITATEA

Numpy

```
Python 3.8.0 (default, Feb  3 2020, 09:42:15)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.12.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import numpy as np
```

... it's that library you always load, without even thinking about it.

REALLY, REALLY

~~Advanced~~ BASIC

THAT YOU NEVER THOUGHT ABOUT

Lisa Schwetlick & Aina Frau-Pascual, ASPP 2022

ASPP 2022

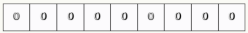
MONDRAGON UNIBERTSITATEA

Structure

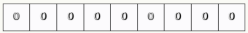
1. **Numpy Array Warmup**
2. **Indexing**
   - Python lists
   - Numpy indexes, slices, and some fancy-ness
3. **At the heart of Numpy**
   - Underneath the ndarray
   - Views and copies
3. **At the heart of Numpy**
   - Broadcasting
4. **Tricks, Conventions, and Applications**
   - Saving stuff
   - Errors
   - Reshaping

Making them (1D)

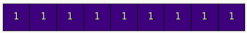
`np.array([0,0,0,0,0,0,0,0,0])`



`np.zeros(9)`



`np.ones(9)`



`np.arange(9)`

`np.random.randint(0,9,(1,9))`

Making them (2D)

```
np.array(
[[0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0,0,0]])
```

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

`np.zeros((5,9))`

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

`np.ones((5,9))`

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

`np.arange(5*9).reshape(5,9)`

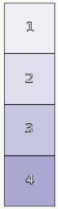
0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44

`np.random.randint(0,9,(5,9))`

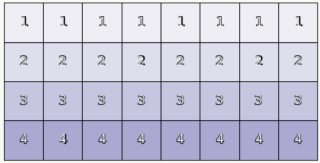
2	5	7	6	1	3	1	6	7
5	0	0	3	3	3	2	5	7
7	6	2	8	8	3	8	5	1
1	3	5	7	2	2	8	4	4
4	6	8	8	0	3	7	4	3

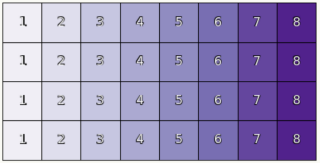
Making Grids

```
yy, xx = np.ogrid[1:5, 1:9]
```




```
yy, xx = np.mgrid[1:5, 1:9]
```





Reminder of basic python indexes

Lists can be indexed with a single number:

```
1 my_list = [1,2,3,4,5]
2 print(my_list)
```

```
>>> [1,2,3,4,5]
```

Reminder of basic python indexes

Lists can be indexed with a single number:

```
1 my_list = [1,2,3,4,5]
2 print(my_list)
```

```
>>> [1,2,3,4,5]
```

```
1 print(my_list[2])
```

Reminder of basic python indexes

Lists can be indexed with a single number:

```
1 my_list = [1,2,3,4,5]
2 print(my_list)
```

```
>>> [1,2,3,4,5]
```

```
1 print(my_list[2])
```

```
>>> 3
```

Reminder of basic python indexes

```
1 my_list = [1,2,3,4,5]
```

Indexing python lists also works using a ***slice*** object:

- it is made using the *slice*() function OR triggered when there is a ":" inside "[ ]"
- sliced lists return a part of the original list
- can be used to modify *in-place*

```
1 print(my_list[2:4])
2 print(my_list[::2])
3 print(my_list[slice(0,4,2)]) # equivalent
```

```
>>> [3,4]
>>> [1,3,5]
>>> [1,3,5]
```

Reminder of basic python indexes

```
1 my_list = [1,2,3,4,5]
```

Indexing python lists also works using a ***slice*** object:

- it is made using the *slice*() function OR triggered when there is a ":" inside "[ ]"
- sliced lists return a part of the original list
- can be used to modify *in-place*

```
1 print(my_list[2:4])
```

```
>>> [3,4]
```

Reminder of basic python indexes

```
1 my_list = [1,2,3,4,5]
```

Indexing python lists also works using a ***slice*** object:

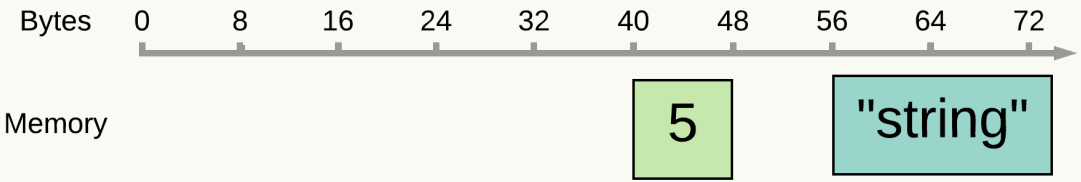
- it is made using the *slice*() function OR triggered when there is a ":" inside "[ ]"
- sliced lists return a part of the original list
- can be used to modify *in-place*

```
1 print(my_list, "at", hex(id(my_list)))
2 my_list[2:4] = [9,8]
3 print(my_list, "at", hex(id(my_list)))
```

```
>>> [1, 2, 3, 4, 5] at 0x127885500
>>> [1, 2, 9, 8, 5] at 0x127885500
```

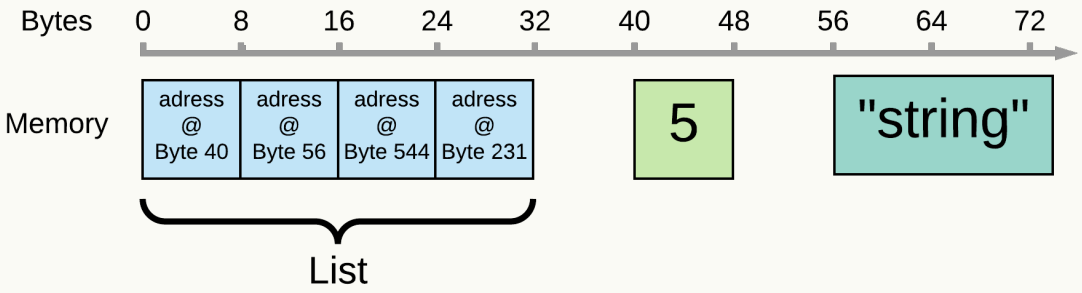
actually, what is a list?

```
In [2]: list = [5, "string", 3.5, ('tu', 'ple')]
```



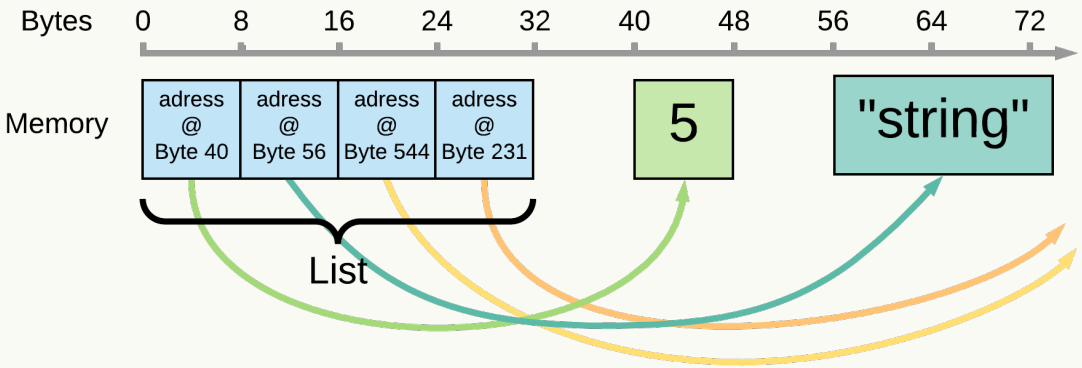
actually, what is a list?

```
In [2]: list = [5, "string", 3.5, ('tu', 'ple')]
```

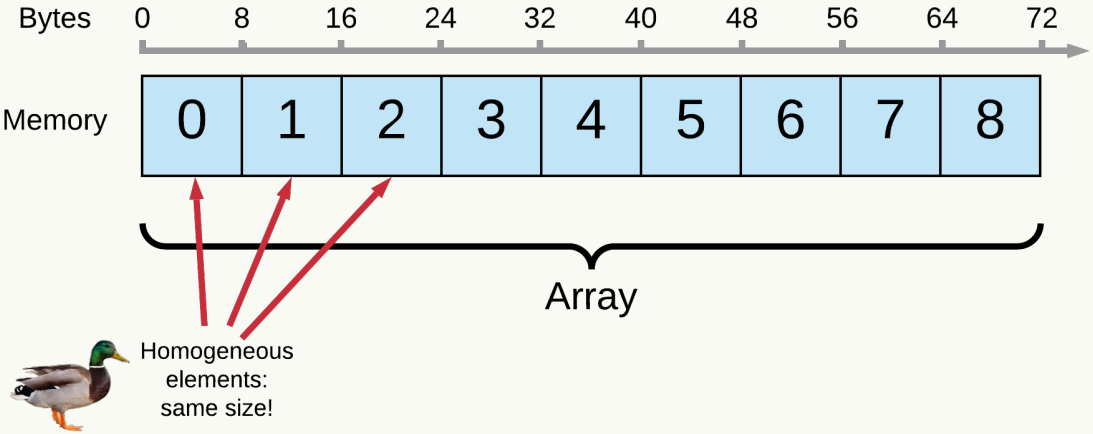


actually, what is a list?

```
In [2]: list = [5, "string", 3.5, ('tu', 'ple')]
```



This is an Array!



so what's the difference?

	Python List	**Numpy Array**
Dimensions	1	n
Elements	can be different types	must be homogeneous
Memory Layout	Stores addresses at which the element data is located	One contiguous block of memory
Memory Access	Access address, then access data	direct data access
Size	Expandable	Not expandable
Contents	Mutable	Mutable
Good for...	variable sequences	large amounts of data; data that will be operated on; vectorization

> Exercise 1

1. Make a random 10x10 numpy array and visualize it.
2. Simulate some roulette rounds in Python and using Numpy. Test which is faster.
3. Use print(hex(id(mylist))) to look at tuples and lists and when you change them.
4. Make a random 10x10 numpy array. Print the element in the fourth column and the fifth row.
5. Print the entire fifth row.

see file `Exercises_Notebook_1.py`

Numpy arrays can have multiple dimensions

```
1 a = np.random.randint(1, 10, (5,5))
2 print(a)
```

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

Ways to index a numpy array

1. field access indexing ‘a[1][2][3]‘
2. regular indexing ‘a[1,2,3]‘
3. slicing ‘a[1:3, 1:3]‘
4. fancy indexing ‘a[[1,2], [1,2]]‘

if you want more: Check out the indexing.ipynb. It shows off all the different kinds of indexing and combinations!

Ways to index a numpy array

1. field access indexing ‘a[1][2][3]‘
2. regular indexing ‘a[1,2,3]‘
3. slicing ‘a[1:3, 1:3]‘
4. fancy indexing ‘a[[1,2], [1,2]]‘

if you want more: Check out the indexing.ipynb. It shows off all the different kinds of indexing and combinations!

Ways to index a numpy array

1. field access indexing ‘a[1][2][3]‘
2. regular indexing ‘a[1,2,3]‘
3. slicing ‘a[1:3, 1:3]‘
4. fancy indexing ‘a[[1,2], [1,2]]‘

if you want more: Check out the indexing.ipynb. It shows off all the different kinds of indexing and combinations!

Ways to index a numpy array

1. field access indexing ‘a[1][2][3]‘
2. regular indexing ‘a[1,2,3]‘
3. slicing ‘a[1:3, 1:3]‘
4. fancy indexing ‘a[[1,2], [1,2]]‘

if you want more: Check out the indexing.ipynb. It shows off all the different kinds of indexing and combinations!

Field Access Indexing

Numpy arrays are compatible with lists of lists in pure python and they can be accessed like this:

```
# 3-dimensional array
a = zeros(3,3,3)
z = x = y = 1
a[z][x][y]
```

Regular Indexing

The more common usage is with just one set of square brackets:
Dimensions are represented by commas. To index explicitly, you need as many values separated by commas as dimensions.

```
a = np.random.randint(1, 10, (5,5))
row = 2
col = 3
a[row, col]
```

```
>>> 2
```

Common pitfall: don't think of the index as coordinates (x and y): They are interpreted in the reverse order!

```
a = np.random.randint(1, 10, (2, 3, 4))
print(a)
```

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

```
>>> x = y = z = 1
>>> print(a[z, y, x])
```

```
>>> 4
```

Numpy Slice

- Numpy slicing expands the slice syntax into n dimensions
- you pass one slice object per dimension in the array
- There are three big differences to base python slicing:
  1. slicing can be done over multiple dimensions
  2. exactly one ellipsis object "..." can be used to indicate several dimensions at once,
  3. slicing cannot be used to expand the size of an array (unlike lists).

Slicing Shorthands

There are some convenient shorthands/default expansions like:

- If you want all indexes from a dimension use ":"
- Ellipses [...] can be used to replace zero or more ":" terms
- If you give fewer comma-separated values than dimensions, it assumes you provided the first dimension's indexes.

```
# 3-dimensional array
a = zeros(3,3,3)
# access the  second element of the first dimension
a[1, :, :] # explicit
a[1, ...] # equivalent
a[1] # equivalent
```

What Dimensionality is my output?

When slicing Dimensionality of the output is determined by the combined dimensionality of the indexes

```
1 a = np.zeros((5,5,5))
2 a[1, ::2, ::2].shape
```

What Dimensionality is my output?

When slicing Dimensionality of the output is determined by the combined dimensionality of the indexes

```
a = np.zeros((5,5,5))
a[1, ::2, ::2].shape
```

```
>>> (3,3)
```

> Dimension Guessing

see file `Exercises_Notebook_1.py`

What Dimensionality is my output?

Single element dimensions disappear

```
1 a = np.zeros((3,3))
2 a[1,:].shape
```

```
>>> (3,)
```

What Dimensionality is my output?

What if I can't use slices, because I want some irregular subset of the data?

```
1 a = np.zeros((5,5))
```

I want every second row and the second and third column!

What Dimensionality is my output?

What if I can't use slices, because I want some irregular subset of the data?

```
a = np.zeros((5,5))
```

I want every second row and the second and third column!

```
a[::2, [2,3]].shape
```

Answer: using a list of indices also works

What Dimensionality is my output?

What if I can't use slices, because I want some irregular subset of the data?

```
1 a = np.zeros((5,5))
```

I want every second row and the second and third column!

```
1 a[::2, [2,3]].shape
```

Answer: using a list of indices also works

```
>>> (3,2)
```

What Dimensionality is my output?

Before we indexed with a slice and a list. Now how about two lists?
What do you think happens here?

```
a = np.zeros((5,5))
a[[2,3], [2,3]].shape
```

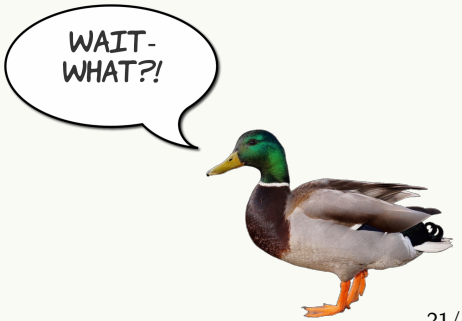
What Dimensionality is my output?

Before we indexed with a slice and a list. Now how about two lists?

What do you think happens here?

```
1 a = np.zeros((5,5))
2 a[[2,3], [2,3]].shape
```

```
>>> (2,)
```



Fancy Indexing

When indexing with multiple lists, they are interpreted as coordinates to look up.

... actually thats just a useful lie to believe for now. Let's talk about it later!

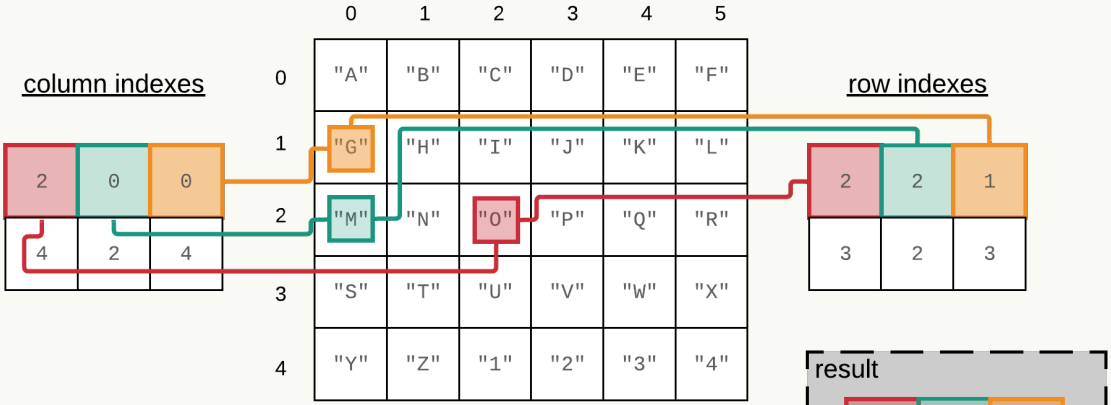
```
a = np.array(list("ABCDEFGHIJKLMNOPQRSTUVWX")).reshape(4,6)
print(a)
```

```
>>> array([['A', 'B', 'C', 'D', 'E', 'F'],
>>>        ['G', 'H', 'I', 'J', 'K', 'L'],
>>>        ['M', 'N', 'O', 'P', 'Q', 'R'],
>>>        ['S', 'T', 'U', 'V', 'W', 'X']], dtype='<U1')
```

```
print(a[[3,3,0],[4,1,5]])
```

```
>>> array(['W', 'T', 'F'], dtype='<U1')
```

Fancy Indexing



Fancy Indexing with Bools

If you pass an array of booleans, Numpy will return the values at the True positions.

```
1 a = np.random.randint(1, 10, (3,3))
2 print(a)
```

```
>>> array([[9, 2, 7],
>>>        [2, 6, 9],
>>>        [1, 7, 3]])
```

Fancy Indexing with Bools

If you pass an array of booleans, Numpy will return the values at the True positions.

```
a = np.random.randint(1, 10, (3,3))
print(a)
```

```
>>> array([[9, 2, 7],
>>>        [2, 6, 9],
>>>        [1, 7, 3]])
```

```
print(a<5)
```

```
>>> array([[False,  True, False],
>>>        [ True, False, False],
>>>        [ True, False,  True]])
```

Fancy Indexing with Bools

If you pass an array of booleans, Numpy will return the values at the True positions.

```
1 a = np.random.randint(1, 10, (3,3))
2 print(a)
```

```
>>> array([[9, 2, 7],
>>>        [2, 6, 9],
>>>        [1, 7, 3]])
```

```
1 print(a[a<5])
```

```
>>> array([2, 2, 1, 3])
```

> Exercises 2

Recreate these plots

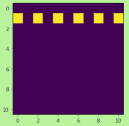
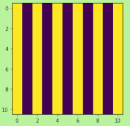
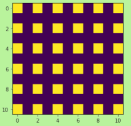
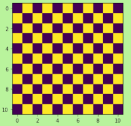
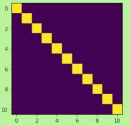
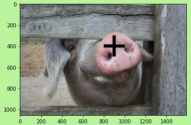
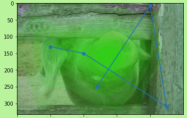
1. 
2. 
3. 
4. 
5. 

Image processing

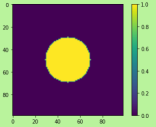




Saliency map evaluation



Recreate this plot



see file `Exercises_Notebook_2.py`

Views and Copies

> [...] a regular indexing expression on an ndarray can always produce an ndarray object **without copying any data**. This is sometimes referred to as the **"view"** feature of array indexing
>
> — Guide to Numpy, pg. 30

- Slicing gives a view
- In-place operations can be done on views
- All fancy indexing returns a copy
- the a.copy() command gives a copy when specifically requested

Views and Copies

```
a = np.array([1,2,3,4,5,6])
v = a[2:5]
print(v)
```

Views and Copies

```
1 a = np.array([1,2,3,4,5,6])
2 v = a[2:5]
3 print(v)
```

```
>>> [3 4 5]
```

Views and Copies

```
1 a = np.array([1,2,3,4,5,6])
2 v = a[2:5]
3 print(v)
```

```
>>> [3 4 5]
```

```
1 v[1] = 999
2 print(v)
```

Views and Copies

```
1 a = np.array([1,2,3,4,5,6])
2 v = a[2:5]
3 print(v)
```

```
>>> [3 4 5]
```

```
1 v[1] = 999
2 print(v)
```

```
>>> [3 999 5]
```

Views and Copies

```
1 a = np.array([1,2,3,4,5,6])
2 v = a[2:5]
3 print(v)
```

```
>>> [3 4 5]
```

```
1 v[1] = 999
2 print(v)
```

```
>>> [3 999 5]
```

```
1 print(a)
```

Views and Copies

```
1 a = np.array([1,2,3,4,5,6])
2 v = a[2:5]
3 print(v)
```

```
>>> [3 4 5]
```

```
1 v[1] = 999
2 print(v)
```

```
>>> [3 999 5]
```

```
1 print(a)
```

```
>>> [1 2 3 999 5 6]
```

Views and Copies

```
1 a = np.zeros([4,4])
2 print(a)
```

```
>>> [[0. 0. 0. 0.]
>>>  [0. 0. 0. 0.]
>>>  [0. 0. 0. 0.]
>>>  [0. 0. 0. 0.]]
```

```
1 v = a[2:4, 2]
2 v[1] = 9
3 print(a)
```

```
>>> [[0. 0. 0. 0.]
>>>  [0. 0. 0. 0.]
>>>  [0. 0. 0. 0.]
>>>  [0. 0. 9. 0.]]
```

```
1 v = a[[2,3], 2]
2 v[1] = 9
3 print(a)
```

```
>>> [[0. 0. 0. 0.]
>>>  [0. 0. 0. 0.]
>>>  [0. 0. 0. 0.]
>>>  [0. 0. 0. 0.]]
```

ndarray

An N-dimensional array is a **homogeneous** collection of "items" indexed using N integers. There are two essential pieces of information that define an N-dimensional array:

1. the shape of the array
2. the kind of item the array is composed of

— Guide to Numpy, pg. 18

ndarray

```
1 a = np.random.randint(1, 10, (3,3))
2 print(a.shape)
3 print(a.dtype)
```

```
>>> (3, 3)
>>> dtype('int64')
```

ndarray properties

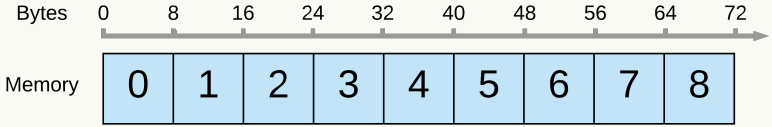
```
>>> a = np.arange(0,9).reshape((3,3))
>>> tools.info(a)
_____________________________

Interface (item)
  shape:       (3, 3)
  dtype:       int64
  length:      3
  size:        9
  endianness:  native (little)
  order:       ☑ C  ☐ Fortran

Memory (byte)
  item size:   8
  array size:  72
  strides:     (24, 8)

Properties
  own data:    ☐ Yes  ☑ No
  writeable:   ☑ Yes  ☐ No
  contiguous:  ☑ Yes  ☐ No
  aligned:     ☑ Yes  ☐ No
_____________________________


>>> a.data
<memory at 0×11b3d21e0>
```

Which information does the ndarray need?

- data: where to start reading
- strides/item size: by how many bits to move
- size: when to stop reading
- data type: how to interpret the bits
- order: which reading convention to use

ndarray properties

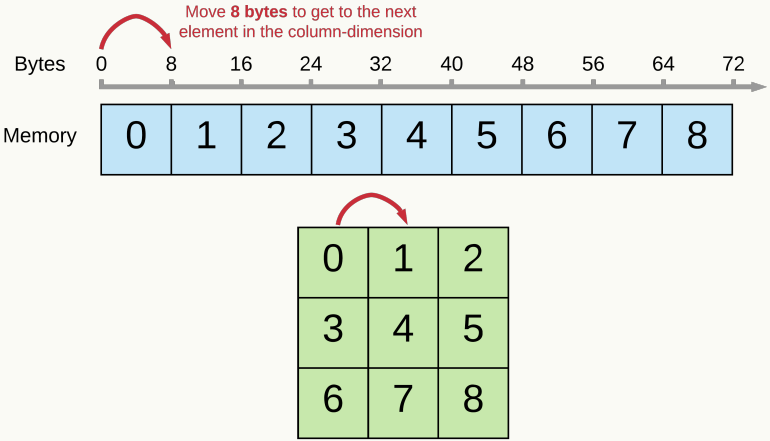
```
>>> a = np.random.rand(3,3)
>>> a.strides
(24, 8)
```



0	1	2
3	4	5
6	7	8

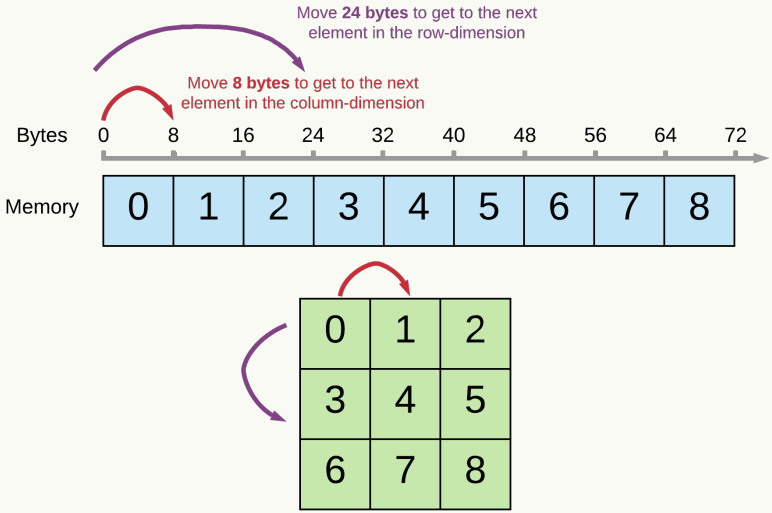
ndarray properties

```
>>> a = np.random.rand(3,3)
>>> a.strides
(24, 8)
```



ndarray properties

```
>>> a = np.random.rand(3,3)
>>> a.strides
(24, 8)
```



Question

Views and Copies

Why does fancy indexing return copies and slices return views?

Stride operations

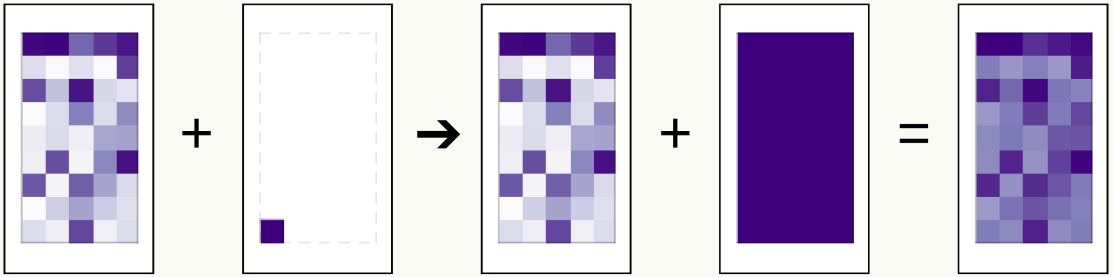
What kind of operations are strides useful for?

ndarray properties

Attribute	Settable	Description
flags	No	special array-connected dictionary-like object with attributes showing the state of flags in this array; only the flags WRITEABLE, ALIGNED, and UPDATEIFCOPY can be modified by setting attributes of this object
shape	Yes	tuple showing the array shape; setting this attribute re-shapes the array
strides	Yes	tuple showing how many *bytes* must be jumped in the data segment to get from one entry to the next
ndim	No	number of dimensions in array
data	Yes	buffer object loosely wrapping the array data (only works for single-segment arrays)
size	No	total number of elements
itemsize	No	size (in bytes) of each element
nbytes	No	total number of bytes used
base	No	object this array is using for its data buffer, or None if it owns its own memory
dtype	Yes	data-type object for this array
real	Yes	real part of the array; setting copies data to real part of current array
imag	Yes	imaginary part, or read-only zero array if type is not complex; setting works only if type is complex
flat	Yes	one-dimensional, indexable iterator object that acts somewhat like a 1-d array
ctypes	No	object to simplify the interaction of this array with the ctypes module
__array_interface__	No	dictionary with keys (data, typestr, descr, shape, strides) for compliance with Python side of array protocol
__array_struct__	No	array interface on C-level
__array_priority__	No	always 0.0 for base type `ndarray`

Broadcasting

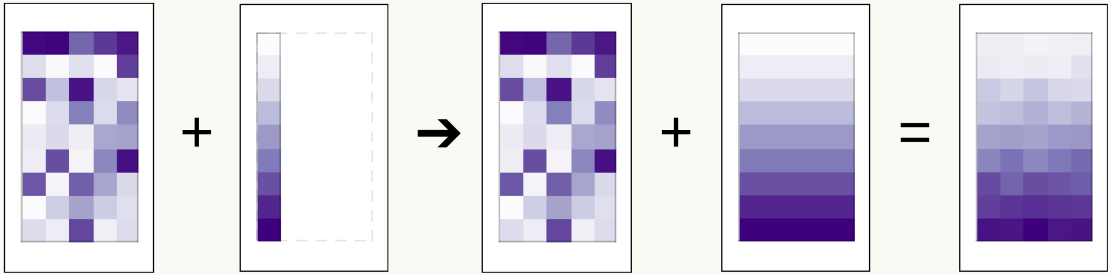
... allows ufuncs to deal with inputs that do not have exactly the same shape.



— plots: https://github.com/rougier/numpy-tutorial

Broadcasting

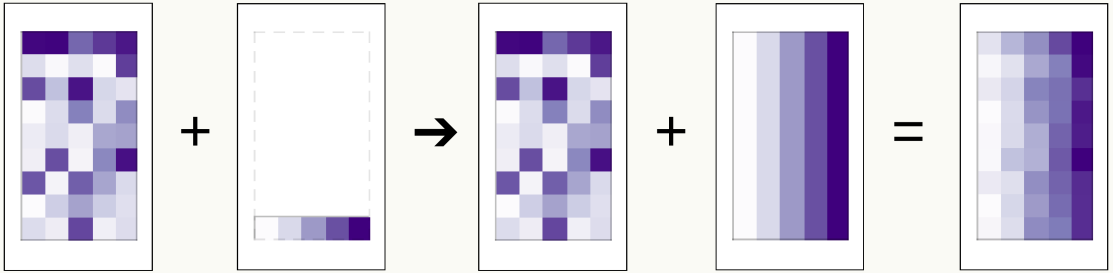
... allows ufuncs to deal with inputs that do not have exactly the same shape.



— plots: https://github.com/rougier/numpy-tutorial

Broadcasting

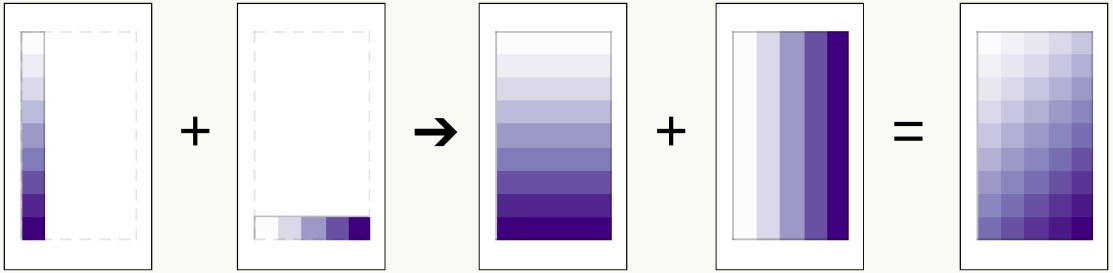
... allows ufuncs to deal with inputs that do not have exactly the same shape.



— plots: https://github.com/rougier/numpy-tutorial

Broadcasting

... allows ufuncs to deal with inputs that do not have exactly the same shape.



— plots: https://github.com/rougier/numpy-tutorial

Broadcasting

Broadcasting can be understood by four rules:

1. All input arrays with ndim smaller than the input array of largest ndim have 1's pre-pended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input shapes in that dimension.
3. An input can be used in the calculation if it's shape in a particular dimension either matches the output shape or has value exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension.

— Guide to Numpy, pg. 154

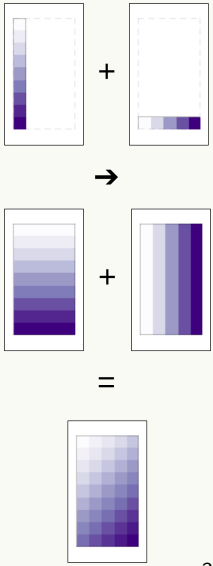
Broadcasting

```
1 a = np.array([1,2,3]) #row vector
2 b = np.array([[4],[5],[6]]) #column vector
3 print(a.shape)
4 print(b.shape)
```

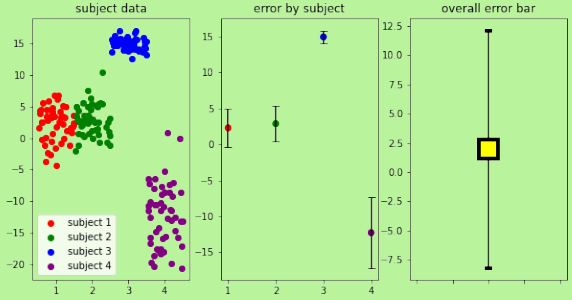
```
>>> (3,)
>>> (3, 1)
```

```
1 np.broadcast_arrays(a, b)
```

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



> Exercise 3



Calculate out the between subject variance, so that we can get a better estimate of the variance for each subject!

see file `Exercises_Notebook_3.ipynb`

Save and Load

The .npy format is the standard binary file format in NumPy for persisting a **single** arbitrary NumPy array on disk.

- for one numpy array
- when called on something other than a pure np array, will use python pickle
- in that case the flag "allow_pickle" needs to be True

```
1 a = np.random.rand((5,5))
2 np.save(a, "my_file.npy")
3 b = np.load("my_file.npy")
```

Errors

What happens when things go wrong?

```
>>> np.geterr()
{'divide': 'warn',
 'over': 'warn',
 'under': 'ignore',
 'invalid': 'warn'}
```

When something goes wrong, but no error is raised, typically NaN is returned.

Errors: divide

Division by zero: infinite result obtained from finite numbers.

```
a = np.divide(3,0)
```

```
>>> <stdin>:1: RuntimeWarning: divide by zero encountered in true_divide
```

```
a
```

```
>>> inf
```

Default consequence: Warning

Errors: over

Overflow: result too large to be expressed.

```
1 np.exp(1234.1)
```

```
>>> <stdin>:1: RuntimeWarning: overflow encountered in exp
>>> inf
```

Default consequence: Warning

Errors: under

Underflow: result so close to zero that some precision was lost.

```
a = np.exp(-(40)**2)
a
```

```
>>> 0.0
```

Default consequence: Ignore

Errors: invalid

Invalid operation: result is not an expressible number, typically indicates that a NaN was produced.

```
a = np.arange(5)
b = np.array([4, 3, 2, np.NaN, 9])
b
```

```
>>> array([ 4.,  3.,  2., nan,  9.])
```

```
a<=b
```

```
>>> <stdin>:1: RuntimeWarning: invalid value encountered in less_equal
>>> array([ True,  True,  True, False,  True])
```

Default consequence: Warn

Errors

You can configure your own safety net, by escalating particularly relevant errors

```
np.seterr(under='raise', over='ignore')
```

it can also be useful to use context managers

```
a = np.arange(5)
b = np.array([4, 3, 2, np.NaN, 9])
with np.errstate(invalid='ignore'):
    c = a <= b
```

Reshaping

```
1 np.arange(8).reshape(2,4)
```

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

- reshaping is very fast (only changes string information)
- -1 means "just make it fit with the other dimensions"
- reshape gives a new view. resize is the same, but in place.

Reshaping

- the corresponding opposites are flatten (copy) and ravel(in-place)
- also useful is squeeze(): return an array with all single dimensions squeezed out.

```
1 np.arange(8).reshape(2,4).flatten()
```

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

> Exercises 4

1. Go insanity mode: switch off all the errors and do all the things you never thought you could!
2. Chose one of every two elements in a 1D array.

see file `Exercises_Notebook_4.ipynb`

In short Numpy introduces:

1. the Numpy array
   - at the core of Numpy functionality
   - it expands the Python List into n dimensions
   - it cleverly uses memory to avoid time-consuming copying and looping
   - is a homogeneous container for various datatypes
2. the ufunc
   - a standard class for functions
   - common interface to mathematical functions that operate on scalars
   - can be made to operate on arrays in an element-by-element fashion
   - uses broadcasting to deal with arrays that are not the same shape

Thank you