Introduction to the Python scientific stack
numpy, scipy & matplotlib

Part 1: why numpy & scipy

Brief historical notes

- Python was first released in 1991. (For reference: later than FORTRAN, C, C++, R, MATLAB, ... but before Java, C#, JavaScript, ...)
- The original goal of Python was to be easier, more flexible, more concise, more readable, safer (less error-prone) than the alternatives. It was designed such that a programmer could write and modify a program quickly.
- It has largely achieved this goal. It has now grown to be certainly one among the 4 most used programming languages in the world (the others being C, C++, Java). Its popularity is still growing.

In case you're not convinced

- A small example: build a list of the first n squares (comparison to C):

```c
int *l, i;
l = calloc(n, sizeof(int));
assert(l != NULL);
for (i = 0; i < n; i++) {
    l[i] = i*i;
}
/* use l here */
free(l); /* need to clean up */
```

```python
l = [i**2 for i in range(n)]
# use l, no clean up needed
```

- In C, you need to declare variables beforehand, manage the memory explicitly, etc. You have a lot of control and you work "close to the metal", but it's a lot of work.
- In Python most of these things are managed automatically for you, many common operations are already available (built-in), it's easier to understand and modify code, etc.
- (One particular case where C is horrible and Python is great is managing strings.)

Easy and expressive has come at a cost

- Python was conceived as a general purpose language, i.e. without a specific goal or set of goals in mind.
- "General purpose" = you can use it for basically anything, e.g. computations, writing a web server, parsing text files, writing a small script that manages your files, etc.
- In particular, it was never designed with the explicit goal of being particularly fast. The design was more focused on relieving the programmer of a lot of work, not necessarily the program. For most tasks, CPUs are so fast anyway that even if the program wastes a few cycles it's fine, a few hundredths of a second don't matter.
- However, in some areas of computer science performance does actually matter a lot. Data science is one such area. Scientific applications often deal with large amount of data or difficult algorithms which take time to execute.

Enter numpy/scipy

- People wanted to have a nice and popular language (Python) but with the possibility to write faster code.
- Also, they wanted a library of commonly used algorithms and functions used when doing science.
- Also, they wanted it to be free and open source.
- (Small aside: an example of a language explicitly tailored for number-crunching and other scientific applications is MATLAB. But as a general purpose language is much less nice, plus it's closed-source, not free and quite expensive. Anther example is Julia, which is more modern, more powerful, faster, and free, but still relatively young and thus not as well-known as Python.)
- Thus numpy was created, and scipy built upon that. Their companion plotting library is called matplotlib (heavily inspired by MATLAB's plotting facilities).

Using numpy/scipy/matplotlib

- From now on, write this on top of your files and at the beginning of all your sessions:

```
import numpy as np
```

- This allows you to call numpy code with the `np` prefix, e.g.

```
np.log(4 + np.exp(3))
```

- (The aliasing `numpy` → `np` is standard and ubiquitous; **always** use it.)
- For matplotlib, the convention is:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Part 2: overview

Overview: numpy

- numpy is a sort of "language extension". It mostly introduces:
  - **a few numeric types** which are adequate for the purposes of heavy numerical computation (number crunching). E.g. 32-bit floating points, 64-bit integers, etc.
  - **a core type called an ndarray** ("N-dimensional array") which essentially is a kind of more efficient list. This is more similar to "traditional" arrays that you find in other languages like C, but more powerful (and they have a ton of methods).
  - **some extra mathematical functions** that operate on the ndarrays and on the different possible numeric types. (Also some constants, like pi.)
  - other functionality, the most important of which is **an alternative "random" module**.
  - there's more stuff, but it's also in scipy
- **Read these parts of the user guide:**

  https://numpy.org/doc/stable/user/index.html
  - What is numpy?
  - Quickstart (up to “Indexing with Boolean Arrays”)
  - The absolute basics for beginners
  - Fundamentals (up to “Broadcasting”)

Overview: scipy

- scipy is built on top of numpy. It is essentially a collection of sub-modules containing libraries for scientific computing. Here are a few areas that are covered:
  - special functions (e.g. erf, gamma, bessel, etc. There are dozens of those.)
  - numerical integration
  - **optimization**
  - linear algebra
  - statistics
  - signal processing
  - etc.
- In order to use one one of those sub-modules, write this:

```
from scipy import special
```

- then, you can use e.g. `special.erf(1.5)`
- Docs: https://docs.scipy.org/doc/scipy/reference/

Overview: matplotlib

- This is for plotting data. You can do 2-d and 3-d plots. It can produce quite sophisticated plots.
- Use it like this:

```
import matplotlib.pyplot as plt
```

- Then you can do a basic plot like this:

```
plt.plot([1,2,3], [6,2,8]) # x points first, y points next
```

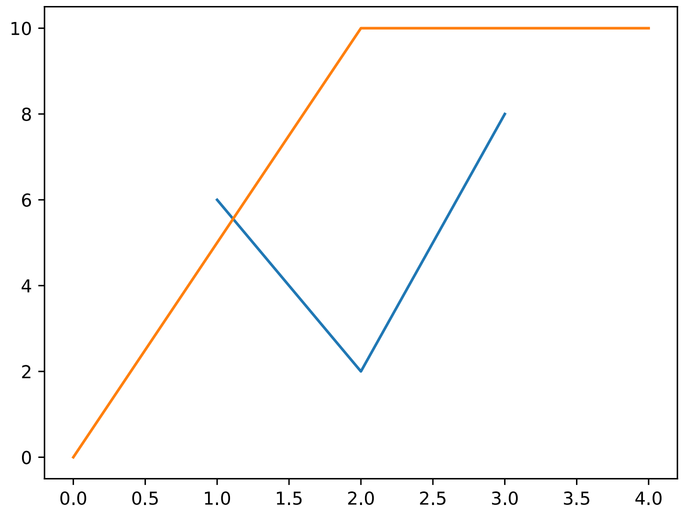
- In Spyder, the plot should appear right away. Otherwise you need to call `plt.show()`.
- If you call `plt.plot` again it will plot on top of the previous one:

```
plt.plot([0,2,4], [0,10,10])
```

- Clear the plot like this:

```
plt.clf() # "clear figure"
```

- **Read this!** https://matplotlib.org/stable/tutorials/introductory/pyplot.html



Overview: more

- There's more: sympy (symbolic mathematics) and pandas (data analysis & manipulation, data frames, etc.)
- See https://projects.scipy.org/index.html

Part 3: numpy is a sort of "language extension"

In order to be fast, numpy introduces a few new things

- The first difference that we're going to look at comes up with integer value types.
- In Python, integers are (in principle) unlimited; in numpy they are not. Try this:

```
In [2]: a = 3 # python integer

In [3]: a**100
Out[3]: 515377520732011331036461129765621272702107522001

In [4]: b = np.int64(3) # numpy integer

In [5]: b**100
__main__:1: RuntimeWarning: overflow encountered in long_scalars
Out[5]: -2984622845537545263

In [6]: a == b # and yet... (equality is not identity)
Out[6]: True
```

- In numpy integers have a fixed number of bits (e.g. 64). Each of them requires a predictable amount of space and the CPU can operate on them in a single instruction (often, even 2 or 4 integers with 1 instruction). It's much faster that way.
- (This is also how the vast majority of programming languages treat integers. Python is an exception.)

Representing integers with a fixed number of bits

- What was the meaning of the negative result in the previous example?
- Suppose that you have `b=4` bits to represent integers. You then can represent at most 2^b=`16` different numbers. You want both negative and positive numbers. The convention in this case is to go from -2^{b-1}=`-8` to $2^{b-1}-1$=`7`, using the first 2^{b-1}=`8` entries for positive numbers and the remaining half for negative ones.
- If you "go over" the maximum or "go below" the minimum (i.e. if you *overflow*), the numbers cycle around. So in out 4-bits example, you get bizarre results like `7+1 → -8`
- With `b=64`, this happens rarely, but it still sometimes does. It's important to know about it. (YouTube knows something about it...)
- There also exist "unsigned" types. In that case you have all values from `0` to 2^b-1, but no negative numbers. In the 4-bits case, you would then have results like `15+1 → 0`, `5*4 → 4` etc. (everything is computed "`mod 16`").
- NOTE: this hypothetical `int4` type does not exist, the smallest one is `int8` (ranges from `-128` to `127`). `uint8` is common in images data (ranges from `0` to `255`)

bits	`int4`	`uint4`
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

(`int4` and `uint4` don't really exist)

The core type of numpy: ndarrays

- ndarray stands for "N-dimensional array"
- All arrays are homogeneous (the elements have the same type). (This statement is imprecise, but close enough to the truth for now.)
- 1-dimensional arrays:
  - quite similar to standard Python's lists. But all the elements have the same type (e.g. all 64-bit integers, or all 32-bit floats, or all bools…). Also, their size is fixed. Can behave like a mathematical vector.
- 2-dimensional arrays:
  - basically a table of numbers. Can behave like a mathematical matrix.
- You can have any number of dimensions you want. For simple cases, 1 or 2 is enough. It's not totally uncommon to have 3 or even 4 (especially in machine learning or computer vision).

Example of a 1-d array

```
In [2]: a = np.array([4,6,8])

In [3]: type(a)            # the type name is ndarray
Out[3]: numpy.ndarray

In [4]: a[0]               # indexed like standard lists
Out[4]: 4

In [5]: type(a[0])         # but the type is not int!
Out[5]: numpy.int64
```

Example of a 2-d array

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

In [3]: a
Out[3]:
array([[4, 5, 6],
       [2, 3, 1]])

In [4]: a[0,1]          # 2-d: two indices per element (row,column)
Out[4]: 5

In [7]: a.ndim          # number of dimensions (AKA "rank")
Out[7]: 2

In [5]: a.shape         # size along each dimension (AKA "axis")
Out[5]: (2, 3)

In [6]: a.size          # total number of elements (2x3 here)
Out[6]: 6
```

Interlude: why are ndarrays faster than lists?

(a very coarse-grained and imprecise discussion...)

- Consider the following trivial code:

```
1. s = 0
2. for x in l:
3.     s += x
```

- If `l` is a list, Python has no way of knowing what its elements look like. They may be integers, floats, lists, user-defined classes, whatever.
- Each time line 3 is executed, Python needs to check the type of `x`, then the type of `s`, then look for the appropriate method (`__add__`, `__radd__`, `__iadd__`, ...) to call. Note that adding two integers or two floats translates into different machine instructions. The type of `s` may change at each iteration. It may need to emit errors (e.g. if you are adding integers but at some point in the list a string is found).
- If you know in advance that all `x` are integers, <u>a lot</u> of work could be skipped.
- Actually, most of the advantages come from the fact that many functions over ndarrays are available as methods written in C (e.g. the above 3 lines: `l.sum()`), which takes advantage from the previous observation and is also faster (because they can operate on contiguous chunks of memory, and don't require dereferencing operations).
- In a nutshell, working with ndarrays allows your code to work "in bulk".
- (Python has optimizations which save work even for lists; the principle stands, though).

A few important constructors and methods

	description	examples

`np.zeros(tuple)`
`np.ones(tuple)`
`np.full(tuple, x)`
	build a ndarray with tuple sizes filled with ones or zeros or x. Default type is `np.float_` (zeros/ones) or whatever x's type is (full), but you can give it as an option.	`np.zeros((2,3))`
		`np.ones((3,5,2),`
		`        dtype=np.int64)`
		`np.full((2,3), 99)`

`np.arange(...)`	build a 1-d ndarray with a range. Same syntax as `range`. Accepts float arguments.	`np.arange(4)      # 0,1,2,3`
		`np.arange(4,0,-1) # 4,3,2,1`

`np.linspace(start, end, n)`	build a 1-d array of length `n` from `start` to `end` (included), at regular intervals.	`np.linspace(1.0, 2.0, 10)`

`a.ndim`
`a.shape`
`a.size`
	number of dimensions, shape and total size of the array `a`	`# see previous slides`

`a.fill(x)`	fill the array `a` with some value `x`. Note: `x` is converted to the type of `a` elements	`a = np.zeros((3,3));`
		`a.fill(5) # 5 becomes 5.0`

`a.max()`
`a.min()`
`a.sum()`
`a.mean()`
`a.argmax()`
`a.cumsum()`
`...`
	standard, mostly self-descriptive functions (look at the docs, there are many more). All of these accept an "axis" optional argument (they can work along one specified dimension instead of the whole array)	`a = np.array([[1,2],[3,4]])`
		`a.sum()        # 10`
		`a.sum(axis=0) # [4,6]`
		`a.sum(axis=1) # [3,7]`

Some notes about ndarrays internals

- When you create an ndarray, its contents (the data) are stored as a single contiguous block of memory. Its size is simply given by the number of elements times the number of bits required for each element.
- The example of the Matrix class (in the "matrix.py" file) should give you a reasonably good idea of how things work for a 2-d case.
- (In reality, it is possible to have ndarrays that only access a subset of their data; the block is still contiguous, but it is accessed in strides, e.g. every 3 elements; see later.)
- Different ndarrays can share the same data, or even just a part of it. **This can be tricky and lead to confusing bugs!** See next slide for an example where this can happen.

Fun with indexing
(for some definition of fun)

- You can index an ndarray with the same syntax as Python's lists, i.e.:
  - you can use slices, e.g. `0:5:2` or even just `:`
  - you can use negative indices to count from the end
  - This syntax can be used along each dimension, e.g. if `a` is a 2-d array
    - `a[0, 0:5]` gives row `0` and columns from `0` to `4`
    - `a[1:3,:]` gives a sub-matrix with rows `1` and `2` and all the columns of `a`
- **But!** If you use slices, what you get is a so-called **view**: an ndarray that shares the same data as the original! (Example in next slide.)
- NOTE: Slices are not the only functions that produce views. Some methods also do that. Two notable ones are `reshape()` and `ravel()`. (They both give you views of the same data but with a different shape; the first one lets you choose the shape, the latter just flattens the array – see the docs, and we will see some of this in the exercises.)
- NOTE: a view of a view is still a view of the original thing...

Example of creating a view, and the consequences of modifying it

```
In [2]: a = np.arange(5)

In [3]: a
Out[3]: array([0, 1, 2, 3, 4])

In [4]: b = a[0:5:2]               # view of a, every other element

In [5]: b
Out[5]: array([0, 2, 4])

In [6]: b[1] = 999                 # this changes both a and b!

In [7]: a
Out[7]: array([  0,   1, 999,   3,   4])

In [8]: c = a[0:5:2].copy()        # how to get a new array

In [9]: a.base is not None         # how to see that a is not a view
Out[9]: False

In [10]: b.base is a               # how to see that b is a view of a
Out[10]: True
```

difference with Python lists!

More indexing...

- There are other ways to index an array which are not available for Python lists.
- Let's say you have a 1-d array `a = np.array([6, 2, 8, 1, 7, 3])`. You can get only a subset of the elements by using:
  - lists, e.g. `b = a[[0,2,3]]` only gives you elements `0,2,3` → `b` is then equal to `np.array([6, 8, 1])`
  - boolean masks, e.g. if `m = np.array([1,0,1,1,0,0,0],dtype=bool)` and you write `a[m]` you get the same result as before. Basically, you're selecting which elements to pick with an indicator (`True`/`False`) per index, instead of using the list of indices.
- <u>In both cases, you don't get views!</u> (well, you may get views, but not of the original array, so you always get independent data)
- You can mix any indexing method with any other along any dimension, e.g. you can do `a[1:3,[2,4,5]]`.
- Rules:
  - If you only use scalars, you get one element (which is not a view).
  - If you only use scalars and slices (at least one), you get views.
  - If you use any lists or masks, you don't get views.

Even more indexing...

- If you omit indices, it's like using `:` in all dimensions that you haven't specified. E.g. you have a 2-d array a and you write `a[3]`, it's like writing `a[3,:]` and thus you get a view. This can be confusing and error-prone, so try not to do that.
- You can use the three dots `...` as a shorthand for "all the rest of the dimensions", e.g. `a[3,...,4]` is like writing `a[3,:,4]` if a is 3-d, or `a[3,:,:,4]` if a is 4-d, etc. Seldom useful.

One last crucial topic: broadcasting

- Basically all functions in numpy use a special application method called "**broadcasting**".
- This can get confusing in its full generality until you get accustomed to it (then it becomes quite powerful). So let's start from the basics.
- At the most basic level, this means that operations are applied element-wise. Let's say that we have two arrays with the same shape, `a` and `b`.
  - If you write `a+b`, you get an array with the element-wise sum of the two. **This is not what happens with Python lists**! (lists would be concatenated!). In this case, it's basically like linear algebra.
  - If you write `a*b`, you get an array with the element-wise product of the two. This is not like linear algebra! (And thus it is quite different from, say, MATLAB.) Even with matrices, therefore, `*` does not give you the matrix product.
  - The same goes for all operators, e.g. `a**b` etc.
  - numpy comes with some functions that are also found in the `math` module, e.g. `sin`, `log`, `exp`, etc. The numpy versions however are a little different (see also a later slide about this). The most notable difference is that the numpy functions broadcast. In most cases, this simply means that you get a new array with the same shape as the original with the result of the function applied to each element.

Basic broadcasting examples

$$\begin{bmatrix} 3 & 4 & 0 \\ 1 & 2 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 3 & 2 \\ 2 & 5 & 1 \end{bmatrix} = \begin{bmatrix} 3*1 & 4*3 & 0*2 \\ 1*2 & 2*5 & 2*1 \end{bmatrix} = \begin{bmatrix} 3 & 12 & 0 \\ 2 & 10 & 2 \end{bmatrix}$$

$$\texttt{np.log2(}\begin{bmatrix} 2 & 4 & 1 \\ 1 & 2 & 2 \end{bmatrix}\texttt{)} = \begin{bmatrix} \texttt{log2(2)} & \texttt{log2(4)} & \texttt{log2(1)} \\ \texttt{log2(1)} & \texttt{log2(2)} & \texttt{log2(2)} \end{bmatrix} = \begin{bmatrix} 1.0 & 2.0 & 0.0 \\ 0.0 & 1.0 & 1.0 \end{bmatrix}$$

Basic broadcasting examples

```
In [2]: a = np.linspace(1.0, 2.0, 6)

In [3]: a
Out[3]: array([ 1. ,  1.2,  1.4,  1.6,  1.8,  2. ])

In [4]: b = np.ones(6)

In [5]: b
Out[5]: array([ 1.,  1.,  1.,  1.,  1.,  1.])

In [6]: a + b      # element-wise summation (NOT CONCATENATION!)
Out[6]: array([ 2. ,  2.2,  2.4,  2.6,  2.8,  3. ])

In [7]: a * b      # also element-wise
Out[7]: array([ 1. ,  1.2,  1.4,  1.6,  1.8,  2. ])

In [8]: np.log(a)  # log applied to each element
Out[8]:
array([ 0.,  0.18232156,  0.33647224,  0.47000363,  0.58778666,
        0.69314718])
```

difference with Python lists!

note: `math.log` wouldn't work here!

One surprise: comparisons also broadcast, even == and !=

```
In [9]: a > b
Out[9]: array([False,  True,  True,  True,  True,  True], dtype=bool)

In [10]: a == b
Out[10]: array([ True, False, False, False, False, False],
dtype=bool)

In [11]: np.array_equal(a, b) # tells you if the arrays are equal
Out[11]: False
```

difference with
Python lists!

Slightly less basic broadcasting: scalars

$$\begin{array}{|c|c|c|}\hline 3 & 4 & 0 \\ \hline 1 & 2 & 2 \\ \hline \end{array} \; * \; 3 \; =$$

(it works as if)
scalars get "expanded"
to compatible arrays
(then things proceed as in the
previous slides)

$$\begin{array}{|c|c|c|}\hline 3 & 4 & 0 \\ \hline 1 & 2 & 2 \\ \hline \end{array} \; * \; \begin{array}{|c|c|c|}\hline 3 & 3 & 3 \\ \hline 3 & 3 & 3 \\ \hline \end{array} \; = \; \begin{array}{|c|c|c|}\hline 3*3 & 4*3 & 0*3 \\ \hline 1*3 & 2*3 & 2*3 \\ \hline \end{array} \; = \; \begin{array}{|c|c|c|}\hline 9 & 12 & 0 \\ \hline 3 & 6 & 6 \\ \hline \end{array}$$

Slightly less basic broadcasting: scalars

- If you have an array `a` and a scalar `x` and you apply an operation like `x*a` or `a**x`, it's like applying the operation with the same `x` to each element of `a`, e.g.:

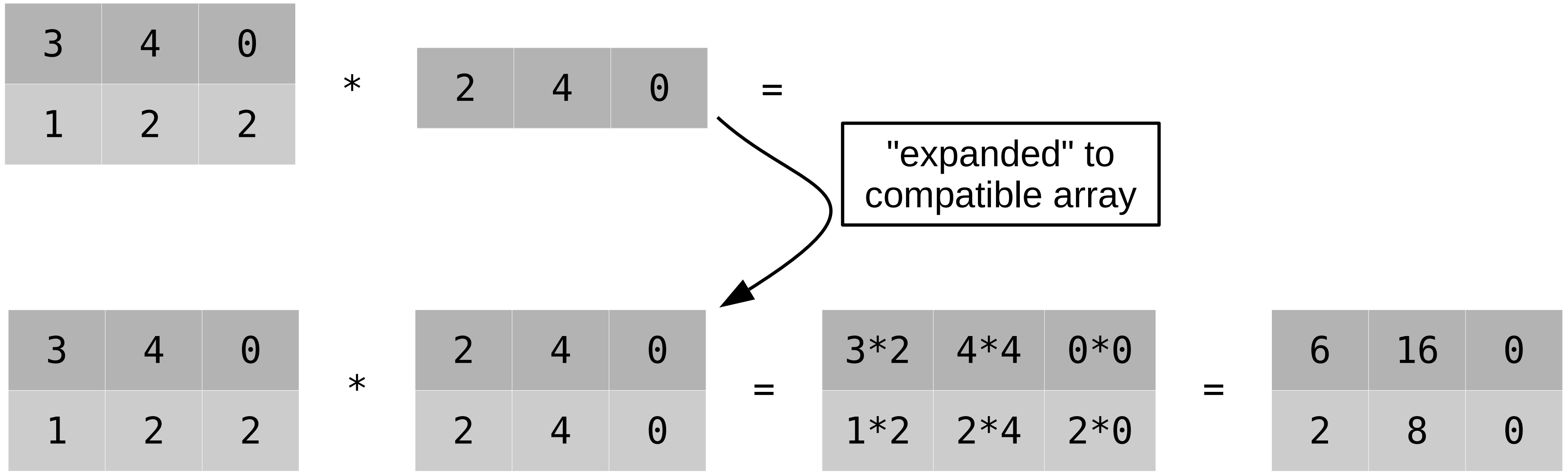
```
In [12]: a
Out[12]: array([  1. ,   1.2,   1.4,  33. ,   1.8,   2. ])

In [13]: a**2 # element-wise power
Out[13]:
array([  1.00000000e+00,   1.44000000e+00,   1.96000000e+00,
         1.08900000e+03,   3.24000000e+00,   4.00000000e+00]

In [14]: a*3  # element-wise multiplication (NOT REPETITION)
Out[14]: array([ 3. ,  3.6,  4.2,  4.8,  5.4,  6. ])
```

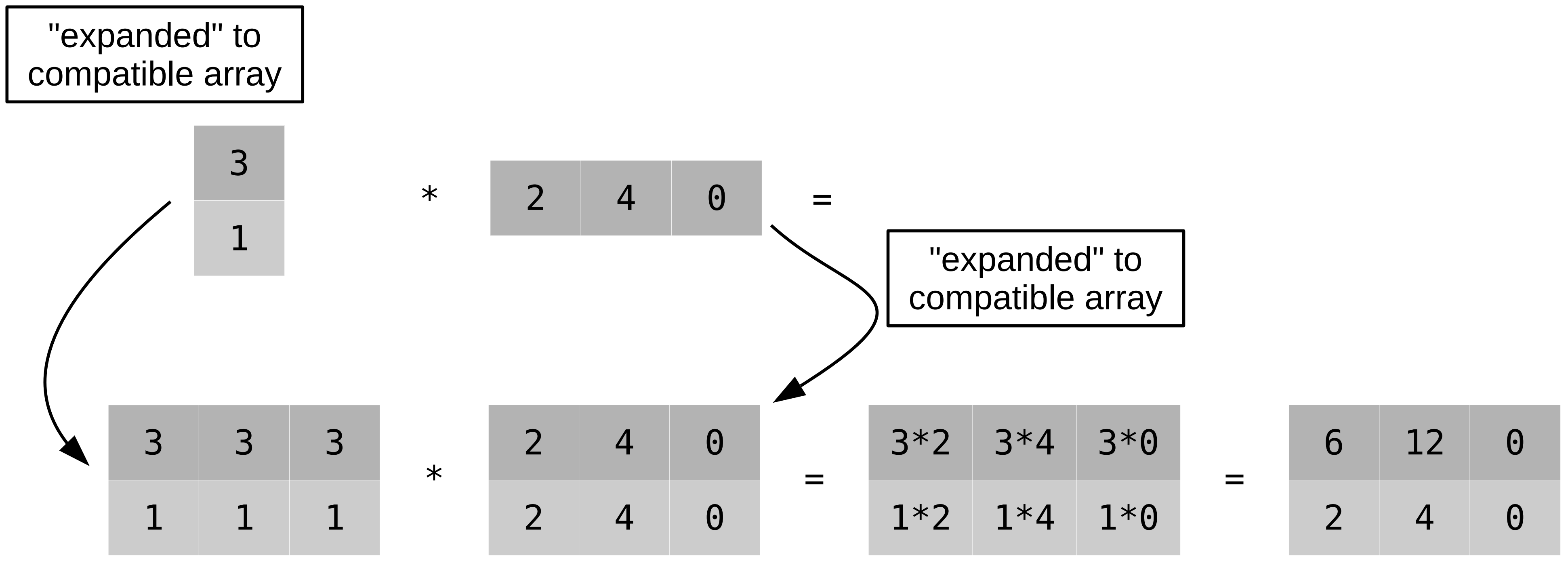
difference with Python lists!

Even less basic broadcasting: 1-d and 2-d (I)



- shape `(3,)` is equivalent to `(1,3)` or to `(1,1,3)`, or to `(1,1,...,1,3)` etc.
- shapes `(2,3)` and `(1,3)` are broadcast to `(2,3)`

Even less basic broadcasting: 1-d and 2-d (II)



- shape `(3,)` is equivalent to `(1,3)` or to `(1,1,3)`, or to `(1,1,...,1,3)` etc.
- shapes `(2,1)` and `(1,3)` are broadcast to `(2,3)`

Even less basic broadcasting: 1-d and 2-d

- If you have a 2-d array `a` and a 1-d array `b`, and you apply an operation like `a*b`, broadcasting "expands" `b` into a 2-d array by repeating it, as if it were a 2-d array with identical rows, e.g.

```
In [3]: a = np.array([[3,4,0],[1,2,2]])

In [4]: b = np.array([2,4,0])

In [5]: a * b   # not a matrix-vector product
Out[5]:
array([[ 6, 16, 0],
       [ 2,  8, 0]])

In [6]: b2 = np.array([[2,4,0],[2,4,0]]) # same as this

In [7]: a * b2
Out[7]:
array([[ 6, 16, 0],
       [ 2,  8, 0]])
```

- This only works if `a.shape==(n,p)` and `b.shape==p` for some `n` and `p`. Otherwise, after the "expansion", you would get two 2-d arrays with different shape (and thus an error).
- The "expansion" is not actually performed, it's done implicitly as not to waste time and memory.

FYI: General broadcasting rules

- This is how broadcasting works in general when you apply an operation to two or more ndarrays:
  1) If the number of dimensions is not the same, all arrays with fewer than the maximum number of dimensions are "padded" with extra dimensions of size 1 at the beginning, until all arrays have the same number of dimensions.
  2) Along each dimension (axis) check the maximum size among the arguments, call it `n`. All arrays must have either size `n` or size `1` along that dimension. If an array has size `1`, its elements are repeated `n` times along that dimension. In this way all arrays end up having the same shape.
  3) The operation is finally applied element-wise.
- Example: you sum two arrays `a` and `b` with shapes `(3,2,1)` and `(2,4)`. Array `b` is treated as if its shape was `(1,2,4)` (rule 1). Then array `a`'s 3rd dimension is repeated as if it had shape `(3,2,4)` (rule 2) and array `b`'s 1st dimension is repeated as if the shape was also `(3,2,4)` (rule 2 again). Finally the sum is applied (rule 3).

Additional remarks on broadcasting

- Broadcasting rules apply even if only one argument is an array and the other is not. We've already seen the case of array and scalar, in which case the scalar is treated as if it was an array with shape `(1,)`. You can even operate with an array and a list, for example, and the list is going to be treated like a 1-d array.
- Broadcasting also sort-of-works in assignment. It's slightly more complicated. A couple of easy cases are as follows (suppose that `a` is a 2-d array):

  -
    ```
    a[:,:] = 2.0     # this is essentially like the fill() method,
                     # sets all elements of a
    ```

  -
    ```
    a[0:2, 0:3] = [1,2,3] # this sets a 2x3 sub-array of a,
                          # and [1,2,3] gets broadcasted
    ```

Other differences between numpy and math functions (aside from broadcasting)

- You would expect that the natural logarithm of a given number is always the same, right? Mathematically, it's like that. In programming, however, there can be subtle differences.
- One important example is the behavior outside of function domains. Observe these examples:

```
In [3]: math.log(0) # THIS GIVES AN ERROR

In [4]: np.log(0)   # this doesn't (by default)
__main__:1: RuntimeWarning: divide by zero encountered in log
Out[4]: -inf

In [5]: (-2.0)**0.5            # this returns a complex number
Out[5]: (8.659560562354934e-17+1.4142135623730951j)

In [6]: np.float64(-2.0)**0.5  # this returns nan instead
__main__:1: RuntimeWarning: invalid value encountered in
double_scalars
Out[6]: nan

In [7]: math.sqrt(-2.0)  # THIS GIVES AN ERROR

In [8]: np.sqrt(-2.0)    # this doesn't
__main__:1: RuntimeWarning: invalid value encountered in sqrt
Out[8]: nan
```

More puzzling differences (mostly a curiosity)

- Even when you evaluate a function in its domain, there can be very small differences due to floating point approximations and differences in the underlying code.
- And even when the results are actually identical, they can be printed differently!

```
In [3]: x, y = math.log(0.54234), np.log(0.54234)

In [4]: x == y              # the result is the same
Out[4]: True

In [5]: x.hex(), y.hex()    # identical down to the last bit
Out[5]: ('-0x1.3945ff81f4cc0p-1', '-0x1.3945ff81f4cc0p-1')

In [6]: x, y                # but if you print them...
Out[6]: (-0.6118621679437624, -0.61186216794376236)

In [7]: type(x), type(y)    # the reason is the different types
Out[7]: (float, numpy.float64)
```

In the above example, the internal number is the same, but since the types are different (`float` vs `np.float64`) they use different printing function when they are converted from hexadecimal to decimal, leading to two (equally valid, but slightly different) strings printed. This has zero consequences, except until you rely on the decimal representation (e.g. if you print the thing on a file and then compare results "as strings", or something like that).

A tip about those differences

- Rather than memorizing those differences, and the behavior of each function in all possible cases, memorize this rule instead:

A tip about those differences

- Rather than memorizing those differences, and the behavior of each function in all possible cases, memorize this rule instead:

NEVER IGNORE WARNINGS

Miscellanea

- The programming style/technique of using arrays and their built-in operations as much as possible (rather than more basic language constructs like for loops) is known as "vectorization". It is common to other languages where arrays are highly optimized (e.g. MATLAB).
  - Often, vectorizing the code can improve both the speed and the clarity (once you're accustomed to it; it's generally less lines of code anyway). Sometimes though the code gets faster but confusing, and somewhat unnatural.
  - Vectorization (avoiding for loops) is not always possible.
- There are many more performance tricks that you can use on top of vectorization. One example is recycling the storage of intermediate results of computations (see the optional argument "`out`" of all numpy functions).
- The behavior of numpy functions in case of problems (overflow, invalid arguments etc.) can be changed, i.e. you can for example turn warnings into errors, or ignore them, etc. This is done with `np.seterr()`. Don't ignore warnings though, unless you know what you're doing (hint: you don't).
- There is a mechanism to add the "broadcasting behavior" to any function you want (with `frompyfunc`). You are unlikely to need it though. It will also be unlikely to be fast.
- In `matplotlib.pyplot`, apart from the `plot()` function, you should have a look at least at two additional useful functions: `hist()` (computes and plots a histogram of the input data) and `imshow()` (gets a 2-d input, treats it as an image and displays it)