Python: numpy

Intro. to computing

`https://mandarm.github.io/computing.html`

Background / motivation

- Many datasets can be represented as arrays (possibly multi-dimensional) of numbers.
- NumPy arrays (`ndarray`s) provide efficient storage and data operations (esp. compared to Python lists).
- Minor tradeoff: `ndarray`s are homogeneous, i.e., can only contain data of the same type.

Agenda

- Creating simple `ndarray`s
- Types supported by NumPy
- Some attributes of `ndarray`s
- More ways of creating `ndarray`s
- Operations on `ndarray`s

Descriptions of many functions have been simplified in the following slides. For a complete description, see the NumPy documentation.

NumPy arrays

Creating NumPy arrays from Python sequences

```
import numpy as np
# General pattern: X = np.array(x) where x is any Python sequence
A = np.array([1, 4, 2, 5, 3])
B = np.array([1, 4, 2, 5, 3], dtype=float)
C = np.array([1.1, 4.2, 2.9, 5.4, 3.7], dtype=np.int32) # truncates!
D = np.array([range(i, i + 3) for i in [2, 4, 6]])

# All entries are ``promoted'' to the ``highest'' type required
E = np.array([1, 4, 2, 5, 3.14])      # integers promoted to floating
    points
C[1] = 10.1                           # C.dtype is integer => truncated
```

NumPy types I

- Types: Booleans, integers, floating points, complex numbers, strings
- Similar to Python's types, but some **important differences**

  Python integers can be arbitrarily large but NumPy integers are *fixed width*

- **Signed integers**
  - `numpy.byte` / `numpy.int8`
  - `numpy.short` / `numpy.int16`
  - `numpy.intc` / `numpy.int32`
  - `numpy.int_` / `numpy.long` / `numpy.int64`
- **Unsigned integers:** for non-negative integers only
  - `numpy.ubyte` / `numpy.uint8`
  - `numpy.ushort` / `numpy.uint16`
  - `numpy.uintc` / `numpy.uint32`
  - `numpy.uint` / `numpy.ulong` / `numpy.uint64`

NumPy types II

- **Floats**
  - `numpy.single` / `numpy.float32`
  - `numpy.double` / `numpy.float64`
- **Strings**: `numpy.str_`
- NumPy maps standard Python type names to its own equivalents
  - `bool` → `numpy.bool` / `numpy.bool_`
  - `int` → `numpy.int_`
  - `float` → `numpy.double` / `numpy.float64`
  - `complex` → `numpy.complex128`

Terminology I

- **axis** / **axes**: non-negative serial numbers assigned to dimensions
  e.g., `axis=0` for rows, `axis=1` for columns, `axis=2` for 3rd dimension
- **shape**: tuple of N non-negative integers specifying size of each dimension

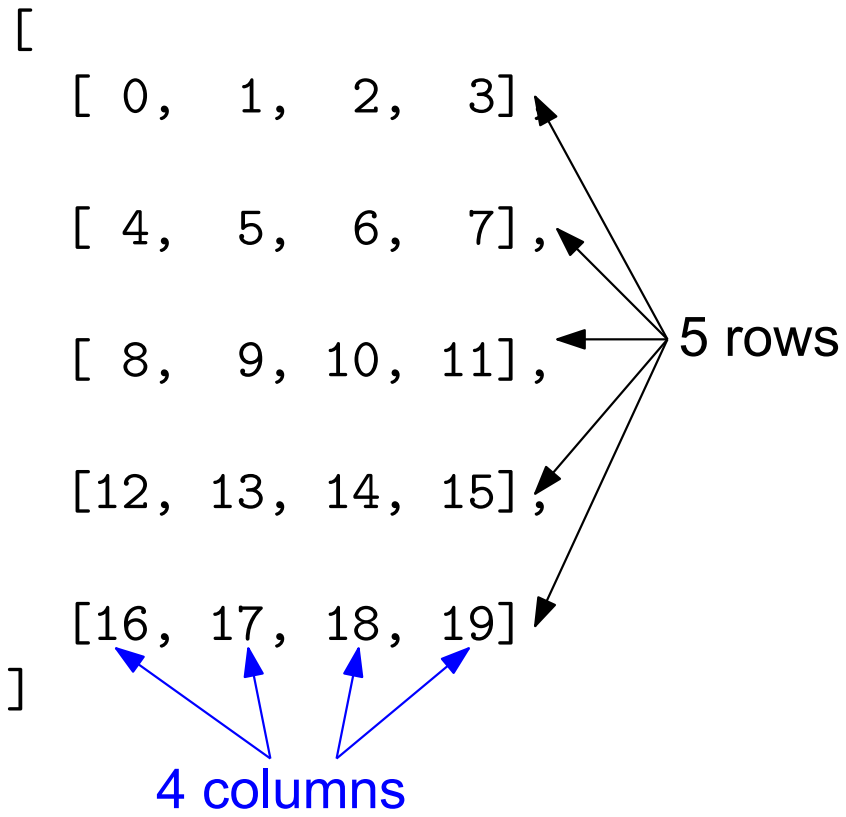
Making sense of multi-dimensional arrays

- Number of dimensions, $k \equiv$ levels of nesting, i.e., number of open brackets ([) you have to cross before getting to first data item
- axis = 0 corresponds to topmost level of nesting
  axis = 1 corresponds to next (2nd) level of nesting
  ...
  axis = $k - 1$ corresponds to innermost level of nesting

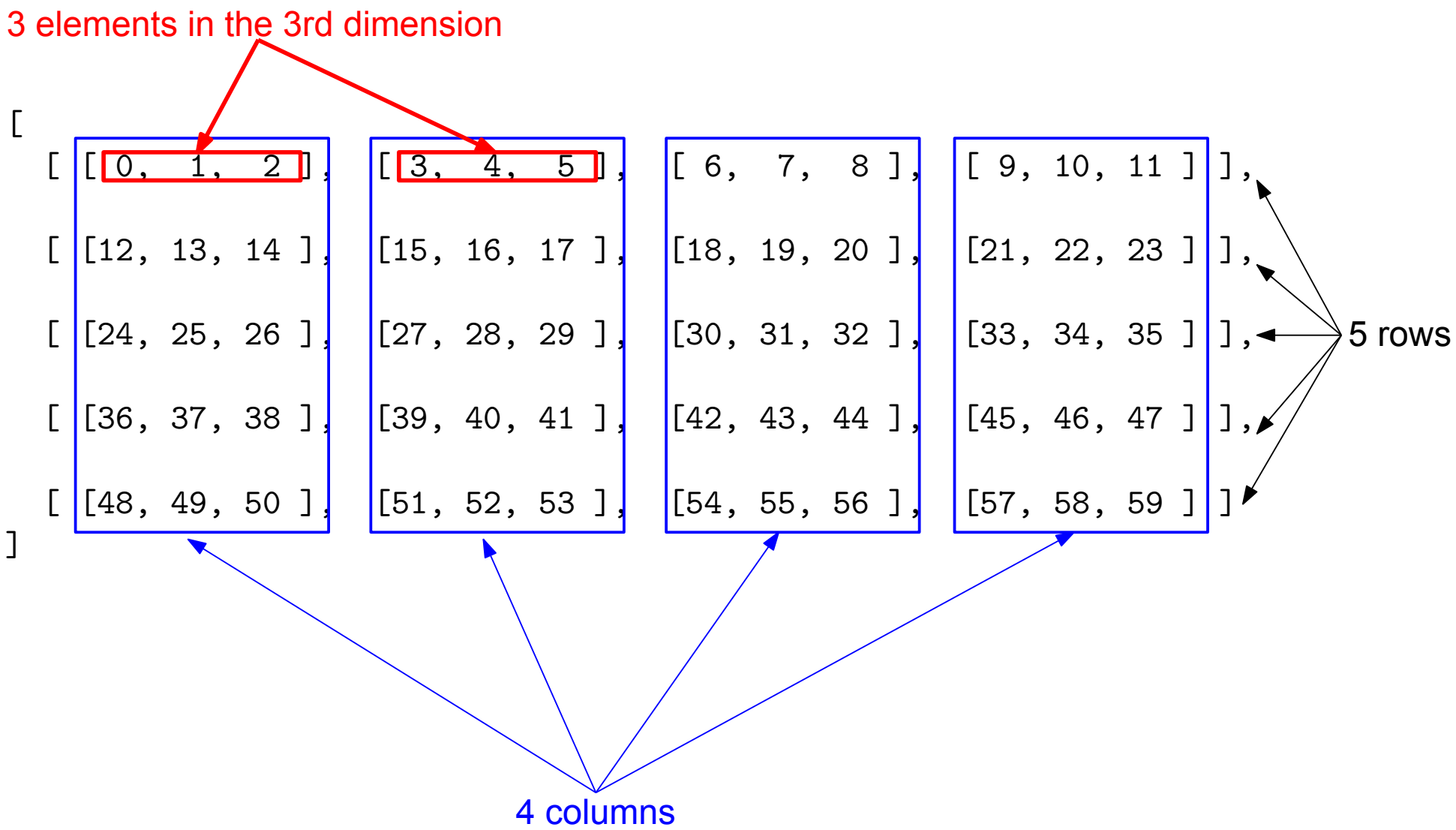
Terminology II

# dim.	Shape	# elements	Command
1	(5,)	5	`A1d = np.arange(5)`
2	(5,4)	20	`A2d = np.arange(20).reshape((5,4))`
3	(5,4,3)	60	`A3d = np.arange(60).reshape((5,4,3))`
4	(5,4,3,2)	120	`A4d = np.arange(120).reshape((5,4,3,2))`

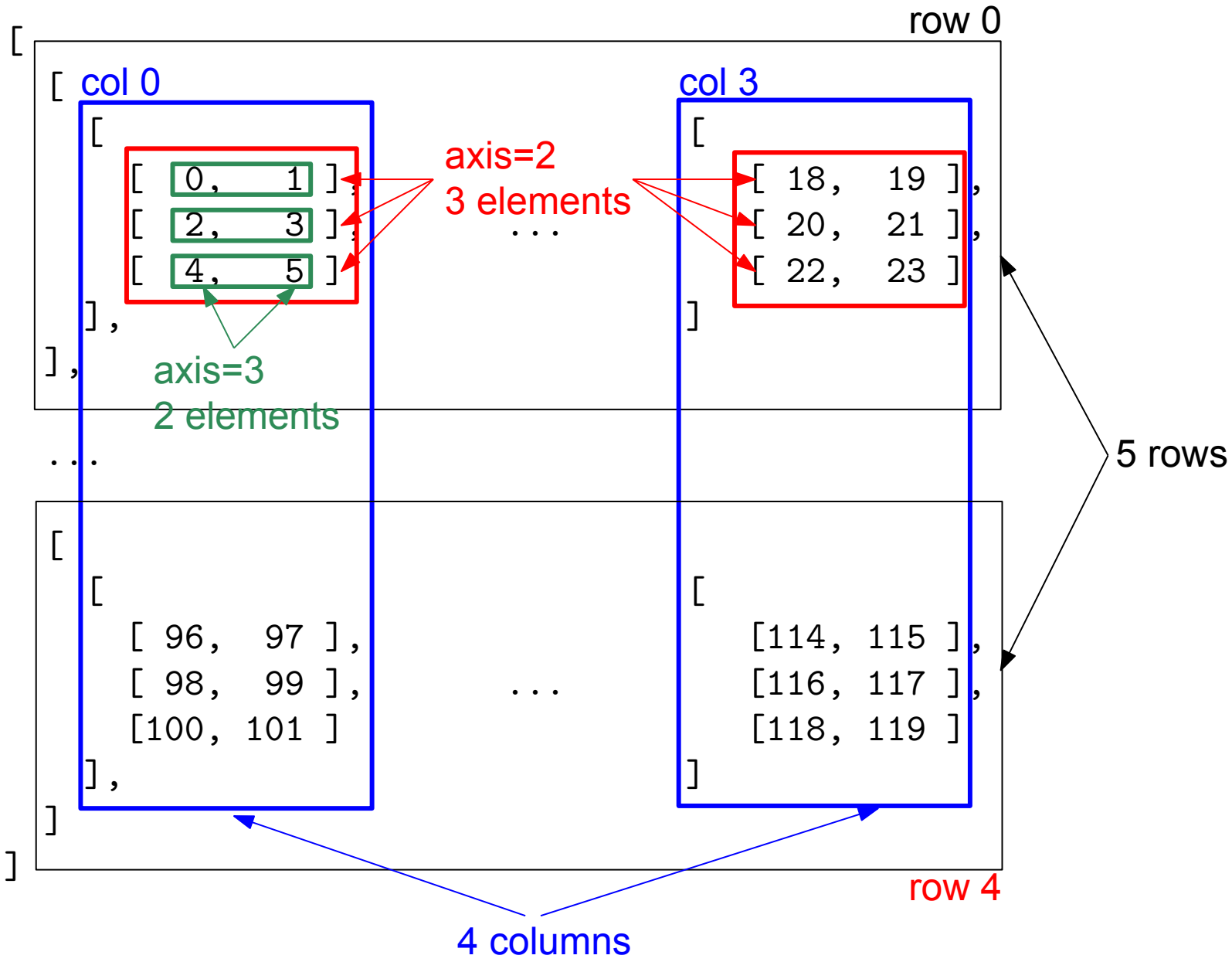
Terminology III



Terminology IV



Terminology V



Functions for creating `ndarrays` I

From Python sequences: see above

Converting existing arrays

- `np.copy(a)`: create a copy
- `np.astype(x, dtype)`: creates a copy with the specified type
- `np.fromiter(iterable, dtype, count)`: create a new 1-dimensional array from `iterable`

optional, but recommended so that required memory can be pre-allocated

Functions for creating `ndarrays` II

Creating special arrays: can optionally specify `dtype` in all of the following

- `np.empty(shape)`: create new *uninitialised* array of given `shape` and type (optional)
- `np.ones(shape)`: create new array of given shape, filled with ones
- `np.zeros(shape)`: create new array of given shape, filled with zeros
- `np.full(shape, fill_value)`: create new array of given shape, filled with `fill_value`
- `np.identity(n)`: create `n` $\times$ `n` identity matrix

  `np.eye(N, M, k)`: generalised version of above

See `numpy-creating-arrays.py` for examples.

Functions for creating `ndarrays` III

Numerical ranges (1-D arrays): can optionally specify `dtype` in all of the following

- `np.arange(start, stop, step)`: create array with evenly spaced values within given interval (like `range()`)
  - safest to use integers as `start`, `stop`, and `step` values
  - `stop` value may sometimes be included due to roundoff error
  - **Warning:** actual step value used to populate the array is `dtype(start + step) - dtype(start)`, rather than `step`, leading to counter-intuitive results.

```
np.arange(0, 5, 0.5, dtype=np.int_)
# -> array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
np.arange(-3, 3, 0.5, dtype=np.int_)
# -> array([-3, -2, -1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

Functions for creating `ndarrays` IV

- `np.linspace(start, stop, num=50)`: create array with `num` evenly spaced values in [`start`, `stop`]
- `np.logspace(start, stop, num=50, base=10.0)`: as above, but numbers are evenly spaced on a logarithmic scale in the interval `[base ** start, base ** stop]`
- `np.geomspace(start, stop, num)`: endpoints are `start` and `stop`, instead of `base ** start`, `base ** stop`

Reshaping

`A.reshape(new-shape-tuple)`

- Parentheses around tuple may be omitted.
- Sizes of initial array and reshaped array must match.

Functions for creating `ndarrays` V

Random matrices: returns single sample if `size` (shape) is omitted

- `np.random.random(size=None)`: create matrix of random floats from continuous uniform distribution over [0.0, 1.0)
- `np.random.uniform(low=0.0, high=1.0, size=None)`
- `np.random.normal(loc=0.0, scale=1.0, size=None)`
  `loc` $\equiv$ mean; `scale` $\equiv$ standard deviation
- `np.random.randint(low, high=None, size)`: create matrix of random integers from discrete uniform distribution over [`low`, `high`), or [0, `low`) (if high is omitted)

See `numpy-creating-arrays.py` for examples.

Functions for creating `ndarrays` VI

Recommended modern usage

```
# Create numpy.random.Generator instance; call diff. methods to obtain
# samples from diff. distributions
rng = np.random.default_rng()
```

Traditional name	Recommended usage
np.random.random(size=None)	rng.random
np.random.uniform(low=0.0, high=1.0, size=None)	rng.uniform
np.random.normal(loc=0.0, scale=1.0, size=None)	rng.normal
np.random.randint(low, high=None, size)	rng.integers

NOTE new name!

- Calling conventions remain the same for the modern equivalents
- **LATER**: using seeds for reproducibility

Some attributes of `ndarrays`

`T`	view of the transposed array (**see examples below**)
`dtype`	data-type of the array's elements
`itemsize`	size of one array element in bytes
`nbytes`	total bytes consumed by array elements
`ndim`	number of dimensions
`shape`	sizes of the dimensions of the array as a tuple
	(length of the shape tuple == number of axes, `ndim`)
`size`	number of elements in the array

Shape and transpose

Difference with conventional Linear Algebra

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

x.shape      # (3,)
x.T.shape    # (3,)

X = np.array([x])     # array([[1, 2, 3]])
X.shape               # (1, 3)
X.T.shape             # (3, 1)
```

`x` and `y` are *1-D* arrays; they have 1 row (with 3 elements), but **NO** columns.
`x` and `x.T` have the **same** shape.

`X` is a *2-D* array, with 1 row and 3 columns.

Digression: The comma is important, even in 1-tuples.

```
print(type((3)))      # int
print(type((3,)))     # tuple
```

```
print((3,) == (3))          # False
print((3,) == tuple([3]))   # True
```

Views vs. copies

- **Indexing and slicing:** as for Python lists, except indexes / slices for multidimensional arrays can be separated by commas

  Examples:

```
A[i, :] # i-th row
A[i]    # i-th row
A[:, j] # j-th column
```

- **BUT** NumPy array slices are *always views*, **NOT** copies

  Examples:

```
a = np.array([1, 2, 3, 4, 5])
s = a[:]
s[2] = 499  # a changes
s = [0]     # Re-assignment!
s = a[:].copy()
s[2] = -999
```

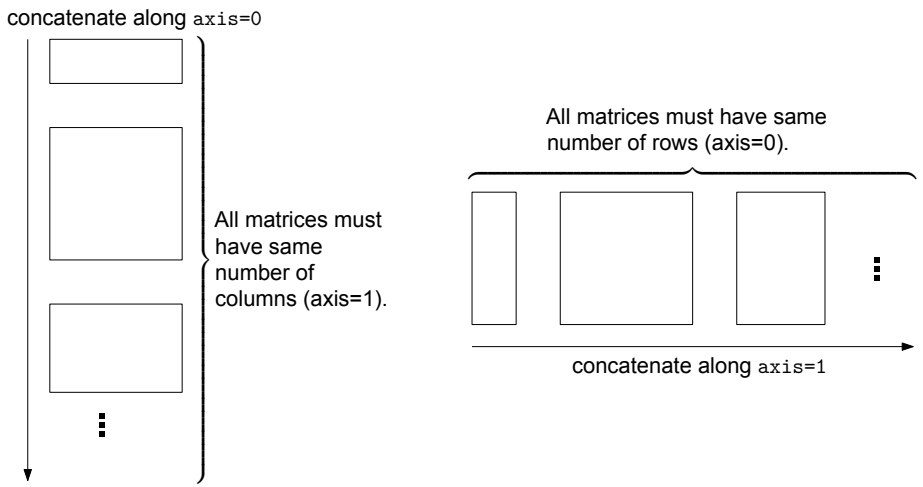
```
l = [1, 2, 3, 4, 5]
l[:2] = []    # VIEW
s = l[:]      # COPY
```

Reshaping also (generally) returns no-copy view of initial array

Concatenating arrays I

See `numpy-concat-split.py` for examples.

- `np.concatenate(arrays, /, axis=0, out=None)`, `np.concat(arrays, /, axis=0, out=None)`: join elements of `arrays` (has to be a *sequence* of arrays) along an existing axis
  - all elements of `arrays` must have same shape, except in dimension corresponding to `axis`
  - `axis` is None $\Rightarrow$ arrays are flattened before joining
  - `out`: existing array of appropriate shape in which result can be stored



Concatenating arrays II

- `np.vstack(arrays)`: stack elements of `arrays` (has to be a *sequence* of arrays) vertically (row wise)
  - similar to `concat` with `axis=0`, but 1-D arrays of shape `(N,)` are first reshaped to `(1,N)`
  - all elements of `arrays` must have same shape, except along first axis
  - 1-D arrays must have same length
  - if `arrays` is a single array (instead of a sequence of arrays), it is treated as a sequence of arrays, i.e., each element along zeroth axis is treated as separate array

```
np.vstack(x) # array([[1], [2], [3]])
```

Concatenating arrays III

- `np.hstack(arrays)`: stack elements of `arrays` (has to be a *sequence* of arrays) horizontally (column wise)
  - similar to `concat` with `axis=1`, but 1-D arrays are concatenated along `axis=0`
  - all elements of `arrays` must have same shape, except along second axis
  - 1-D arrays can be of any length
  - if `arrays` is a single array (instead of a sequence of arrays), it is treated as a sequence of arrays, i.e., each element along zeroth axis is treated as separate array

- `np.stack(arrays, axis=0, out=None)`: join a *sequence* of arrays along a **new** axis (SKIP)
- np.block(arrays): (SKIP)

Splitting arrays

- `numpy.split(A, indices, axis=0)`: split `A` into multiple sub-arrays
  - `indices`: may be single integer or list of indices
    - single integer $\equiv$ # of equal sized splits (`indices` must divide `A.shape[axis]`)
    - (sorted) list $\equiv$ split points (N split points $\Rightarrow N+1$ subarrays) each split point $\equiv$ starting index of subarray
  - resulting sub-arrays are views into `A`, not copies

weird things happen with unsorted list!

- `np.hsplit(A, indices)`: as above, with `axis=1` (column-wise) (for 1-D arrays, `axis` is still 0)
- `np.vsplit(A, indices)`: as above, with `axis=0` (column-wise)

Universal functions (ufuncs)

Universal functions (ufuncs)

- Iterating over lists, lists of lists, etc. to operate on each element is very slow.
- NumPy redefines operations and functions to work element-wise very efficiently.

Examples:

```
rng = np.random.default_rng(seed=987654321)
values = rng.integers(1, 10, size=5)
print(1/values)  # works element-wise
print(2 * values)  # works element-wise
x = np.arange(9).reshape((3, 3))
2 ** x           # works element-wise
```

Arithmetic operators (re)implemented in NumPy

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Operands must have same shape.

LATER: Broadcasting.

Warning: matrix multiplication

`A*B` – element-wise multiplication (Hadamard product)

`A@B` or `np.dot(A,B,out=C)` or `A.dot(B)` – 'normal' matrix multiplication

Math functions (work element-wise)

`np.sin(x)`	`np.arcsin(x)`	`np.exp(x)`	`np.log(x)`
`np.cos(x)`	`np.arccos(x)`	`np.exp2(x)` ($\equiv 2^x$)	`np.log2(x)`
`np.tan(x)`	`np.arctan(x)`	`np.power(x)`	`np.log10(x)`
`np.abs(x)`			

and many more.

Aggregation functions

sum	min	mean	median	any
prod	max	std	percentile	all
	argmin	var		
	argmax			

- Can be used as `np.f(A)` or `A.f()`
- Generally much faster than equivalent Python built-ins
- Without `axis`: aggregation is done over *all* elements

  With (optional) `axis=k`: aggregation / collapsing is done along specified axis $\Rightarrow$ result has one dimension less than initial ndarray

Examples:

```
M = rng.integers(0, 10, (3, 4)) # 3 rows, 4 columns
M.min(axis=0)                   # aggregate along rows => 4 values
M.max(axis=1)                   # aggregate along rows => 3 values
```

Broadcasting

Broadcasting

Recall:

- For arithmetic operators re-implemented in NumPy, operands were required to have same shape for simplicity.
- Except: arithmetic op. on ndarray with scalar $\equiv$ pointwise operation.
- **Broadcasting:** rules generalising above to allow binary NumPy operations to be applied between arrays of different sizes and shapes

Broadcasting rules

1. If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Comparisons, masks, Boolean operations

Comparison operators

- Standard relational operators also work as element-wise ufuncs.
- Result $\equiv$ ndarray of Booleans

Operator	Equivalent ufunc	Operator	Equivalent ufunc
`==`	`np.equal`	`!=`	`np.not_equal`
`<`	`np.less`	`<=`	`np.less_equal`
`>`	`np.greater`	`>=`	`np.greater_equal`

Aggregation functions revisited

- Motivation
  - How many entries satisfy a condition?
  - How many entries in each row/column/dimension satisfy a condition?
  - Do any / all entries (possibly in each row/column/dimension) satisfy a condition?
- Functions: `count_nonzero()`, `sum()`, `any()`, `all()`
- Usage: `np.f(⟨Boolean condition⟩)`
  (takes optional `axis` argument, as before)

Elementwise Boolean operators

`~`	not	`&`	and
`|`	or	`^`	exclusive or

Warning

'Standard' Boolean operators (`and`, `or`, etc.) will give errors / unexpected results.
(They operate on object as a whole, **not** element-wise.)

Boolean conditions/arrays as masks

- Motivation: use Boolean conditions/arrays for *masking*, i.e., to *select* elements
- Usage: `A[⟨Boolean condition⟩]`
- Returns *one-dimensional array* filled with all elements that satisfy given condition, i.e., all elements in positions where mask array contains `True`

Examples:

```
x = rng.integers(100, size=(3, 4))
print(x[(x < 75) & (x > 25)]   # Don't forget the ()

middle\_part = (x < 75) & (x > 25)
print(x[middle\_part])
```

Structured NumPy arrays for general data

```
## Creating/accessing arrays with field names

# Define field names and formats
dtype = [('name', 'S10'),  # string of max length 10
         ('age', 'i4'),    # 4-byte integer
         ('weight', 'f8')]

# Provide data as list of tuples (each tuple is a record (row))
data = [('Alice', 25, 55.0), ('Bob', 35, 78.5), ('Cathy', 28, 61.2)]
structured_array = np.array(data, dtype=dtype)

# Access columns by field name
print(structured_array['name']) # [b'Alice' b'Bob' b'Cathy']
```

I/O: saving as text (e.g., .txt, .csv)

- `np.savetxt()`

```
# Save the array to a text file with space delimiter
np.savetxt('my_array.txt', my_array_txt)

# Save the array to a CSV file with comma delimiter
np.savetxt('my_array.csv', my_array_txt, delimiter=',')

# Save with custom formatting (e.g., specific precision)
np.savetxt('my_array_formatted.txt', my_array_txt, fmt='%.2f', delimiter
    ='\t')
```

References

- *Python Data Science Handbook*, Jake VanderPlas, 2nd ed. (2023), O'Reilly Media; Part II, Chapters 4–9.
  https://jakevdp.github.io/PythonDataScienceHandbook/
- Official documentation
  Quick start: https://numpy.org/devdocs/user/quickstart.html
  User guide: https://numpy.org/doc/stable/user/index.html
- Cheatsheets
  www.datacamp.com/cheat-sheet/numpy-cheat-sheet-data-analysis-in-python
  https://www.dataquest.io/blog/numpy-cheat-sheet
  https://quickref.me/numpy
  https://intellipaat.com/blog/tutorial/python-tutorial/numpy-cheat-sheet
  blog.finxter.com/collection-10-best-numpy-cheat-sheets-every-python-coder-must-own