Python and NumPy

Aniello Panariello, Lorenzo Bonicelli, Matteo Boschini, Angelo Porrello, Simone Calderara

September 23, 2022

University of Modena and Reggio Emilia

Agenda

Python

- Data Types
- Containers
- Functions and Classes

NumPy

Exercise

Python

Python

Python is a high-level, **dynamically** typed multiparadigm programming language. Python code allows you to express very **powerful ideas in very few lines of code** while being very readable.

As of Janurary 1, 2020, Python has officially dropped support for python2. For this class all code will use Python 3.8.

Python

We recommend using the python distribution from **anaconda3**:
https://www.anaconda.com/products/individual

Make sure to set *"Add Anaconda to my PATH environment variable"* to use the conda command from any console.

Setup. Once installed run `conda init` from any console, then restart it.
If using *PowerShell* you might need to run `Set-ExecutionPolicy RemoteSigned` as Administrator.

Virtual environments.

- `conda create -n <venv_name>` — create a new conda environment.
- `conda activate <venv_name>` — activate (use) the environment.
- `conda deactivate` — deactivate the environment.

Once your environment is active, you can install new packages with the *pip* package installer. **Numpy** can be installed with `pip install numpy` from your terminal.

Like most languages, Python has a number of basic types including integers, floats, booleans, and strings. These data types behave in ways that are familiar from other programming languages.

```python
x = 3

print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"

print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.

```
x = 3

x += 1
print(x)   # Prints "4"
x *= 2
print(x)   # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

Booleans: Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (&&, ||, etc.):

```
t = True
f = False

print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```

Strings: Python has great support for strings.

```python
hello = 'hello'     # String literals can use single quotes
world = "world"     # or double quotes; it does not matter.

print(hello)        # Prints "hello"
print(len(hello))   # String length; prints "5"

hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"

hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
print(hw12)  # prints "hello world 12"
```

String objects have a bunch of useful methods; for example:

```python
s = "hello"

print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces;
                       # prints "  hello"
print(s.replace('l', '(ell)'))  # Replace all instances of one substring
                                # with another; prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace;
                           # prints "world"
```

Containers

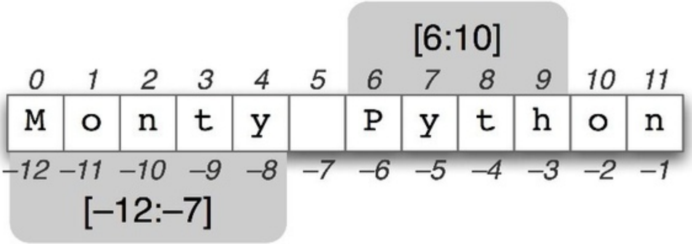
Python includes several built-in container types: lists, dictionaries, sets, and tuples.

A **list** is the Python equivalent of an array, but is resizeable and can contain elements of different types:

```
xs = [3, 1, 2]      # Create a list

print(xs, xs[2])    # Prints "[3, 1, 2] 2"

xs[2] = 'foo'       # Lists can contain elements of different types
print(xs)           # Prints "[3, 1, 'foo']"
xs.append('bar')    # Add a new element to the end of the list
print(xs)           # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()        # Remove and return the last element of the list
print(x, xs)        # Prints "bar [3, 1, 'foo']"
```

A **list** is the Python equivalent of an array, but is resizeable and can contain elements of different types:

```
xs = [3, 1, 2]       # Create a list
print(xs[-1])        # Negative indices count from the
                     # end of the list; prints "2"
```



Slicing: In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing.

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

Start position *End position* *The increment*

```python
nums = list(range(5))     # range is a built-in function
                          # that creates a list of integers

print(nums)               # Prints "[0, 1, 2, 3, 4]"

print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive);
```

Slicing: In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing.

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

Start position   End position   The increment

```
nums = list(range(5))     # range is a built-in function
                          # that creates a list of integers

print(nums[2:])           # Get a slice from index 2 to the end;
                          # prints "[2, 3, 4]"

print(nums[:2])           # Get a slice from the start to
                          # index 2 (exclusive); prints "[0, 1]"
```

```
nums = list(range(5))

print(nums[:])          # Get a slice of the whole list;
                        # prints "[0, 1, 2, 3, 4]"

print(nums[:-1])        # Slice indices can be negative;
                        # prints "[0, 1, 2, 3]"

nums[2:4] = [8, 9]      # Assign a new sublist to a slice
print(nums)             # Prints "[0, 1, 8, 9, 4]"
```

Loops: You can loop over the elements of a list like this.

```
animals = ['cat', 'dog', 'monkey']

for animal in animals:
    print(animal)

# Prints "cat", "dog", "monkey", each on its own line.
```

If you want access to the **index** of each element within the body of a loop, use the built-in `enumerate` function:

```
animals = ['cat', 'dog', 'monkey']

for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))

# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

List comprehensions

When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []

for x in nums:
    squares.append(x ** 2)

print(squares)    # Prints [0, 1, 4, 9, 16]
```

You can make this code simpler using a **list comprehension**:

```python
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]

print(squares)    # Prints [0, 1, 4, 9, 16]
```

List comprehensions

List comprehensions can also contain **conditions**:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]

print(even_squares)  # Prints "[0, 4, 16]"
```

A dictionary stores (key, value) pairs, similar to a Map in Java or an object in Javascript. You can use it like this:

```
d = {'cat': 'cute',
     'dog': 'furry'}  # Create a new dictionary with some data

print(d['cat'])       # Get an entry from a dictionary;
                      # prints "cute"

print('cat' in d)     # Check if a dictionary has a given key;
                      # prints "True"
```

A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

```
d = {'cat': 'cute',
     'dog': 'furry'}  # Create a new dictionary with some data

d['fish'] = 'wet'     # Set an entry in a dictionary

print(d['fish'])      # Prints "wet"
```

A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

```
print(d['monkey'])  # KeyError: 'monkey' not a key of d

print(d.get('monkey', 'N/A'))  # Get an element with a default;
                               # prints "N/A"

print(d.get('fish', 'N/A'))    # Get an element with a default;
                               # prints "wet"
```

UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

It is easy to iterate over the keys in a dictionary.

```
d = {'person': 2, 'cat': 4, 'spider': 8}

for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))

# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

If you want access to keys and their corresponding values, use the `items` method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}

for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))

# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

```
nums = [0, 1, 2, 3, 4]

even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}

print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"
```

A **set** is an unordered collection of distinct elements. As a simple example, consider the following:

```
animals = {'cat', 'dog'}

print('cat' in animals)    # Check if an element is in a set;
                           # prints "True"
print('fish' in animals)   # prints "False"

animals.add('fish')        # Add an element to a set
print('fish' in animals)   # Prints "True"
```

A **set** is an unordered collection of distinct elements. As a simple example, consider the following:

```
animals = {'cat', 'dog', 'fish'}

animals.add('cat')          # Adding an element that is already in
                            # the set does nothing
print(len(animals))         # Prints "3"

animals.remove('cat')       # Remove an element from a set
print(len(animals))         # Prints "2"
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

Set comprehensions: we can easily construct sets using set comprehensions.

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Prints "{0, 1, 2, 3, 4, 5}"
```

A **tuple** is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

```
t = (5, 6)          # Create a tuple

d = {(x, x + 1): x for x in range(10)}  # Create a dictionary
                   # with tuple keys -> { (0, 1): 0; (1, 2): 1; ...}

print(type(t))      # Prints "<class 'tuple'>"
print(d[t])         # Prints "5"
print(d[(1, 2)])    # Prints "1"
```

You can refer to this page for the differences between lists and tuples.

Python functions are defined using the `def` keyword. For example:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]: # Prints "negative", "zero", "positive"
    print(sign(x))
```

Function

We will often define functions to take **optional keyword arguments**, like this:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

The syntax for defining classes in Python is straightforward:

```
class AbstractGreeter(object):

    # Constructor
    def __init__(self, name):
        pass

    # Instance method
    def greet(self, loud=False):
        pass
```

UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)
```

```python
g = Greeter('Fred')  # Construct an instance of the Greeter class

g.greet()            # Call an instance method;
                     # prints "Hello, Fred"

g.greet(loud=True)   # Call an instance method;
                     # prints "HELLO, FRED!"
```

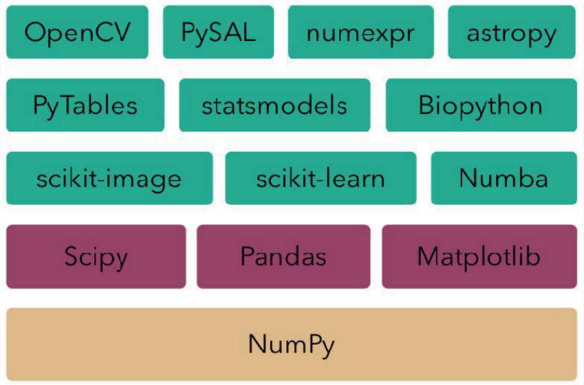
NumPy

NumPy is a Python C extension library for array-oriented computing:

- Efficient
- In-memory
- Contiguous (or Strided)
- Homogeneous (but types can be algebraic)

Array come in c hanno tipo

NumPy is the foundation of the Python scientific stack:



Quick Start

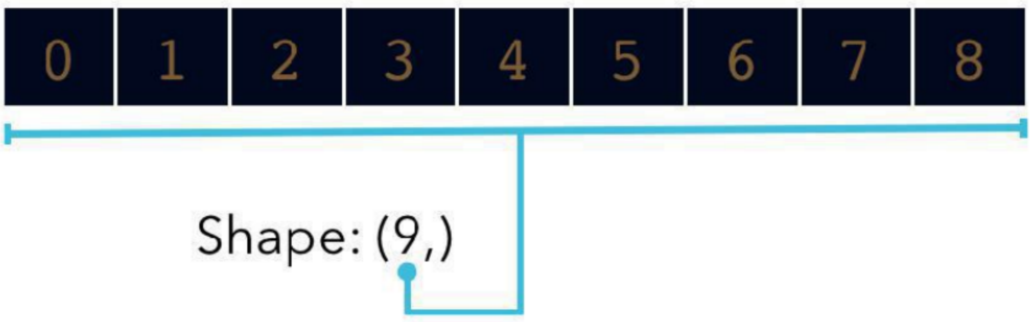
```
In [1]: import numpy as np

In [2]: A = np.array([1,2,3,4])

In [3]: A.dtype #type of what is stored in the array - NOT python types!
Out [3]:dtype('int64')

In [4]: A.ndim #number of dimensions (axes in numpy speak)
Out [4]: 1

In [5]: A.shape #size of the dimensions as a tuple
Out [5]: (4,)

In [6]: A.reshape((4,1)).shape #a column vector
Out [6]: (4, 1)
```

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

In [2]: a = np.array([1,2,3,4,5,6,7,8,9])

In [3]: a
Out[3]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [4]: b = a.reshape((3,3))

In [5]: b
Out[5]:
array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
```

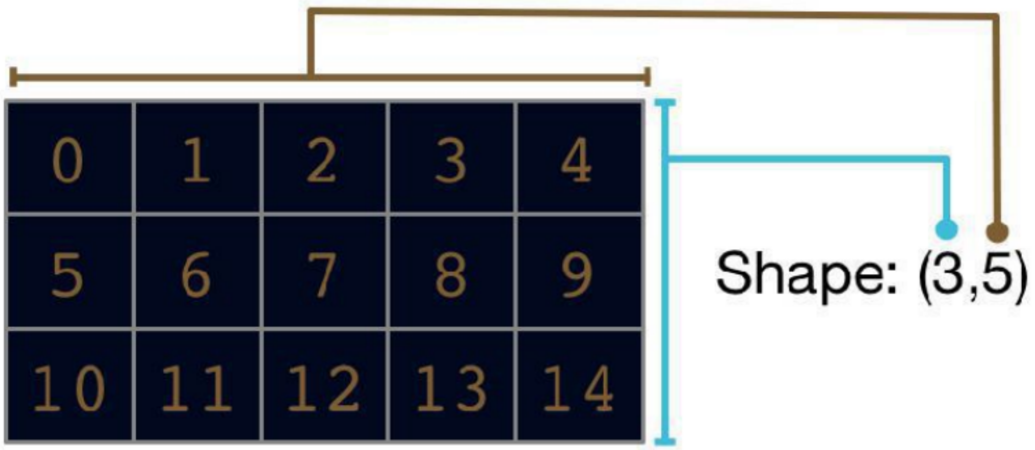
```
In [6]: b * 10 + 4
Out[6]:
array([[14, 24, 34],
[44, 54, 64],
[74, 84, 94]])
```

Array Shape

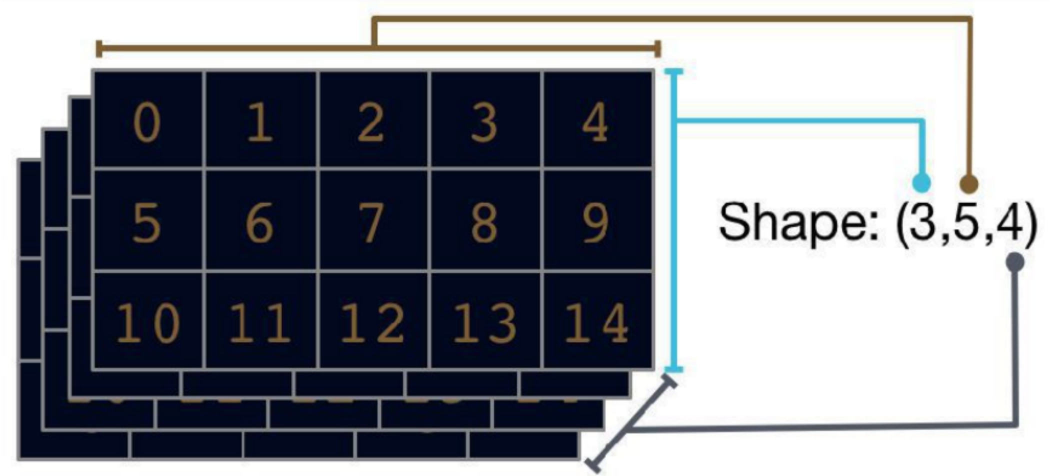
One dimensional arrays have a 1-tuple for their shape:



Two dimensional arrays have a 2-tuple:



...and so on:



Array Element Type (dtype)

NumPy arrays comprise elements of a single data type. The type object is accessible through the `.dtype` attribute.

Here are a few of the most important attributes of dtype objects

- `dtype.byteorder` — big or little endian
- `dtype.itemsize` — element size of this dtype
- `dtype.name` — a name for this dtype object
- `dtype.type` — type object used to create scalars

There are many others...

Array Element Type (dtype)

Array dtypes are usually inferred automatically:

```
In [16]: a = np.array([1,2,3])

In [17]: a.dtype
Out[17]: dtype('int64')

In [18]: b = np.array([1,2,3,4.567])
In [19]: b.dtype
Out[19]: dtype('float64')

In [20]: a = np.array([1,2,3], dtype=np.float32)

In [22]: a
Out[22]: array([ 1., 2., 3.], dtype=float32)
```

Array Creation

Explicitly from a list of values:

```
In [2]: np.array([1,2,3,4])
Out[2]: array([1, 2, 3, 4])
```

As a range of values:

```
In [3]: np.arange(10)
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

By specifying the number of elements:

```
In [4]: np.linspace(0, 1, 5)
Out[4]: array([ 0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Array Creation

Zero-initialized:

```
In [4]: np.zeros((2,2))
Out[4]:
array([[ 0., 0.],
[ 0., 0.]])
```

One-initialized:

```
In [5]: np.ones((1,5))
Out[5]: array([[ 1., 1., 1., 1., 1.]])
```

Uninitialized:

```
In [4]: np.empty((1,3))
Out[4]: array([[ 2.12716633e-314, 2.12716633e-314, 2.15203762e-314]])
```

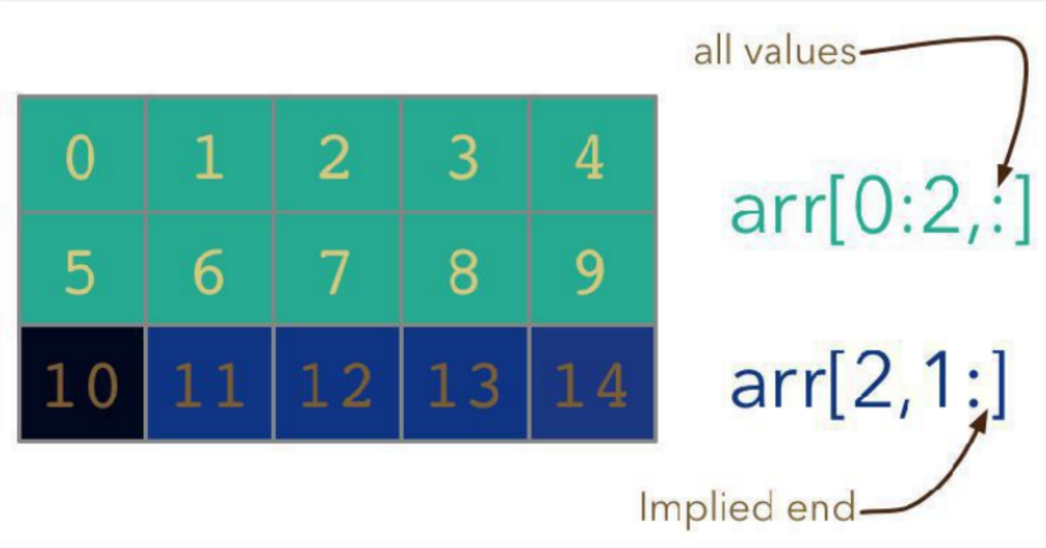
Constant diagonal value:

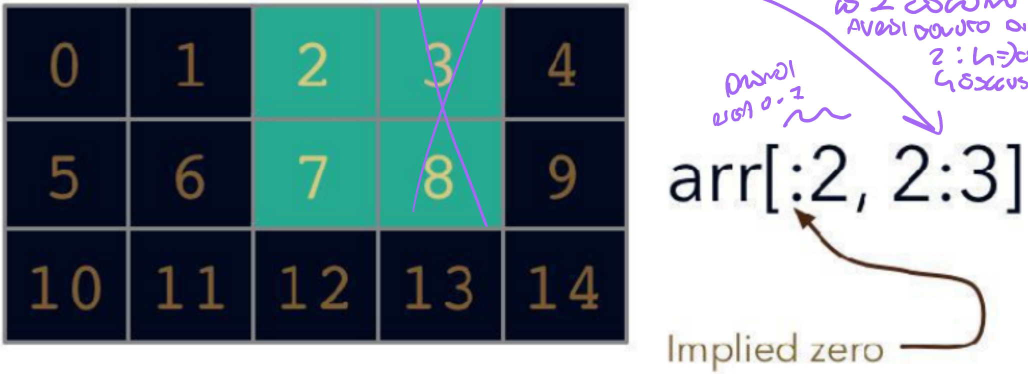
```
In [6]: np.eye(3)
Out[6]:
array([[ 1., 0., 0.],
[ 0., 1., 0.],
[ 0., 0., 1.]])
```

Multiple diagonal values:

```
In [7]: np.diag([1,2,3,4])
Out[7]:
array([[1, 0, 0, 0],
[0, 2, 0, 0],
[0, 0, 3, 0],
[0, 0, 0, 4]])
```

Indexing and Slicing

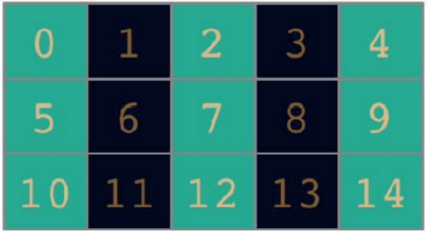




NON È INCLUSO
QUINDI PER OTTENERE
LE 2 COLONNE
AVREI DOVUTO DIRE
2:4 => CON
4 ESCLUSO

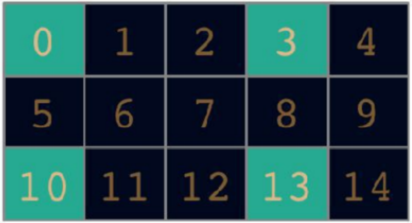
QUINDI
RIGA 0, 1

NumPy array indices can also take an optional stride:



:prendi tutto, , ::2 a step di due

`arr[:,::2]`



prendi le righe a step di due e le colonne a step di tre

`arr[::2,::3]`

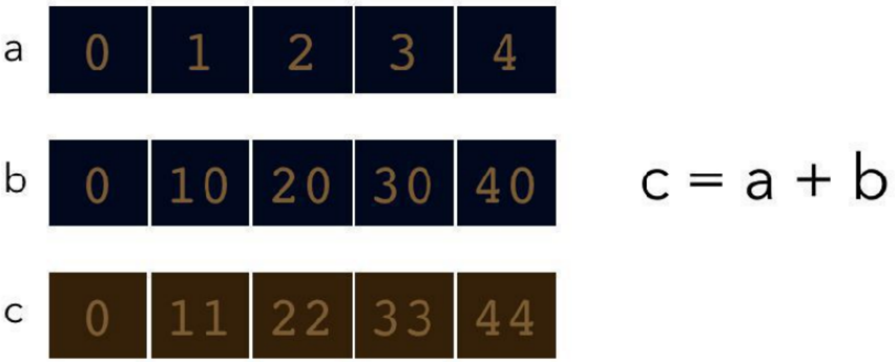
Simple assigments do not make copies of arrays (same semantics as Python). Slicing operations do not make copies either; they return views on the original array:

```
In [2]: a = np.arange(10)
In [3]: b = a[3:7]

In [4]: b
Out[4]: array([3, 4, 5, 6])

In [5]: b[:] = 0

In [6]: a
Out[6]: array([0, 1, 3, 0, 0, 0, 0, 7, 8, 9])

In [7]: b.flags.owndata
Out[7]: False
```

Array views contain a pointer to the original data, but may have different shape or stride values. Views always have `flags.owndata` equal to False.

cosi crea direttamente un array np
da 1 a 10, con np.array(range(10))
crea una lista da 1 a 10 e poi la
copia inun aray np

NumPy ufuncs are functions that operate element-wise on one or more arrays:



ufuncs dispatch to optimized C inner-loops based on array dtype.

Array Element Type (dtype)

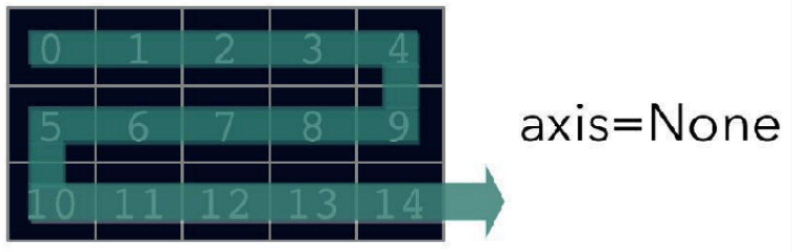
NumPy has many built-in ufuncs:

- **comparison**: `<`, `<=`, `==`, `!=`, `>=`, `>`
- **arithmetic**: `+`, `-`, `*`, `/`, `reciprocal`, `square`
- **exponential**: `exp`, `expm1`, `exp2`, `log`, `log10`, `log1p`, `log2`, `power`, `sqrt`
- **trigonometric**: `sin`, `cos`, `tan`, `acsin`, `arccos`, `atctan`
- **hyperbolic**: `sinh`, `cosh`, `tanh`, `acsinh`, `arccosh`, `atctanh`
- **bitwise operations**: `&`, `|`, `~`, `^`, `left_shift`, `right_shift`
- **logical operations**: `and`, `logical_xor`, `not`, `or`
- **predicates**: `isfinite`, `isinf`, `isnan`, `signbit`
- **other**: `abs`, `ceil`, `floor`, `mod`, `modf`, `round`, `sinc`, `sign`, `trunc`

There are many others...

Array method reductions take an optional axis parameter that specifies over which axes to reduce (`axis=None` reduces into a single scalar):

```
In [7]: a.sum()
Out[7]: 105
```



`axis=None` is the default

riduce le righe a quelle indicate

`axis=0` reduces into the zeroth dimension:

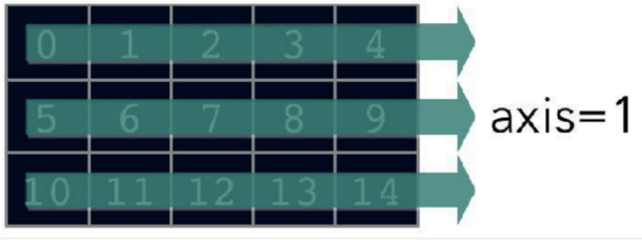
```
In [8]: a.sum(axis=0)
Out[8]: array([15, 18, 21,
24, 27])
```

Reducing an axis

`axis=1` reduces into the first dimension:

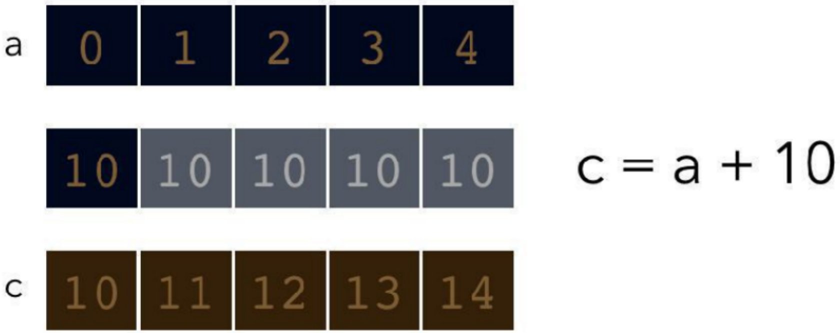
riduce la matrice a una singola colonna

```
In [9]: a.sum(axis=1)
Out[9]: array([10,35,60])
```

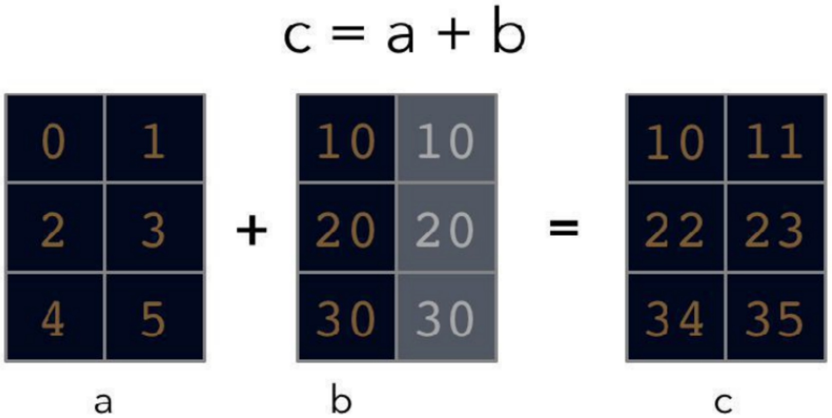


se indico axis=2 dovrebbe compattare i
due layer della matrice a 1 layer
sommando cella a cella

A key feature of NumPy is broadcasting, where arrays with different, but compatible shapes can be used as arguments to ufuncs:



In this case an array scalar is broadcast to an array with shape `(5,)` A slightly more involved broadcasting example in two dimensions:



Here an array of shape `(3, 1)` is broadcast to an array with shape `(3, 2)` vedere documentazione numpy per avere tutto più chiaro

If the dimensions do not match up, `np.newaxis` may be useful:

```
In [16]: a = np.arange(6).reshape((2, 3)) # shape (2, 3)
In [17]: b = np.array([10, 100]) # shape (2,)
In [18]: a * b
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
in ()
----> 1 a * b
ValueError: operands could not be broadcast together with shapes (2,3) (2)

In [20]: a * b[:,np.newaxis] # (2, 3) * (2, 1)
Out[20]:
array([[ 0, 10, 20],
[300, 400, 500]])
```

Array Methods

- **Predicates**: `a.any()`, `a.all()`
- **Reductions**: `a.mean()`, `a.argmin()`, `a.argmax()`,`a.trace()`, `a.cumsum()`, `a.cumprod()`
- **Manipulation**: `a.argsort()`, `a.transpose()`, `a.reshape(...)`, `a.ravel()`, `a.fill(...)`, `a.clip(...)`
- **Complex Numbers**: `a.real`, `a.imag`, `a.conj()`

Boolean arrays can also be used as indices into other arrays:

```
In [3]: a
Out[3]:
array([[ 0, 1, 2, 3, 4],
[ 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]])

In [4]: b = (a % 3 == 0)
In [5]: b
Out[5]:
array([[ True, False, False, True, False],
[False, True, False, False, True],
[False, False, True, False, False]], dtype=bool)

In [6]: a[b]
Out[6]: array([ 0, 3, 6, 9, 12])
```

- **Data I/O**: `fromfile`, `genfromtxt`, `load`, `loadtxt`, `save`, `savetxt`
- **Mesh Creation**: `mgrid`, `meshgrid`, `ogrid`
- **Manipulation**: `einsum`, `hstack`, `take`, `vstack`

Other Subpackages:

- `numpy.fft` — Fast Fourier transforms
- `numpy.polynomial` — Efficient polynomials
- `numpy.linalg` — Linear algebra `cholesky`, `det`, `eig`, `eigvals`, `inv`, `lstsq`, `norm`, `qr`, `svd`
- `numpy.math`: — C standard library math functions
- `numpy.random` — Random number generation `beta`, `gamma`, `geometric`, `hypergeometric`, `lognormal`, `normal`, `poisson`, `uniform`, `weibull`

Exercise

- Write a function that takes a 1d numpy array and computes its reverse vector (last element becomes the first).
- Given the following square array, compute the product of the elements on its diagonal. `[[1 3 8] [-1 3 0] [-3 9 2]]`
- Create a random vector of size (3, 6) and find its mean value.
- Given two arrays a and b, compute how many time an item of a is higher than the corresponding element of b.

```
a: [[1 5 6 8] [2 -3 13 23] [0 -10 -9 7]]
b: [[-3 0 8 1] [-20 -9 -1 32] [7 7 7 7]]
```

- Create and normalize the following matrix (use min-max normalization). `[[0.35 -0.27 0.56] [0.15 0.65 0.42] [0.73 -0.78 -0.08]]`

Let's run a little benchmark! Given a matrix $\mathbf{A} \in \mathbb{R}^{NxM}$ and a vector $\mathbf{b} \in \mathbb{R}^{M}$, compute the euclidean distance between $\mathbf{b}$ and each row $\mathbf{A}_{i,:}$ of $\mathbf{A}$:

$$d(\mathbf{a}, \mathbf{b}) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_M - b_M)^2} = \sqrt{\sum_{j=1}^{M} (a_i - b_i)^2}.$$

By filling in the blanks in *eucl_distance.py*, implement this simple function **twice**:

- with *vanilla Python operators*,
- with *optimized Numpy operations*.

Which one runs faster? Read the provided code carefully and watch out for mistakes ;)

Final Disclaimer

If you're one of the lucky ones with access to the GitHub Copilot beta,
please **disable it now**!

Resources

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- **Reference**: https://numpy.org/doc/stable/reference/
- **User guide**: https://numpy.org/doc/stable/user/index.html
- **Basics tutorial**: https://numpy.org/doc/stable/user/quickstart.html
- **Examples**: https://numpy.org/doc/stable/reference/routines.html