

# *Python Advance Course via Astronomy street*



Sérgio Sousa (CAUP)

ExoEarths Team (<http://www.astro.up.pt/exoearths/>)

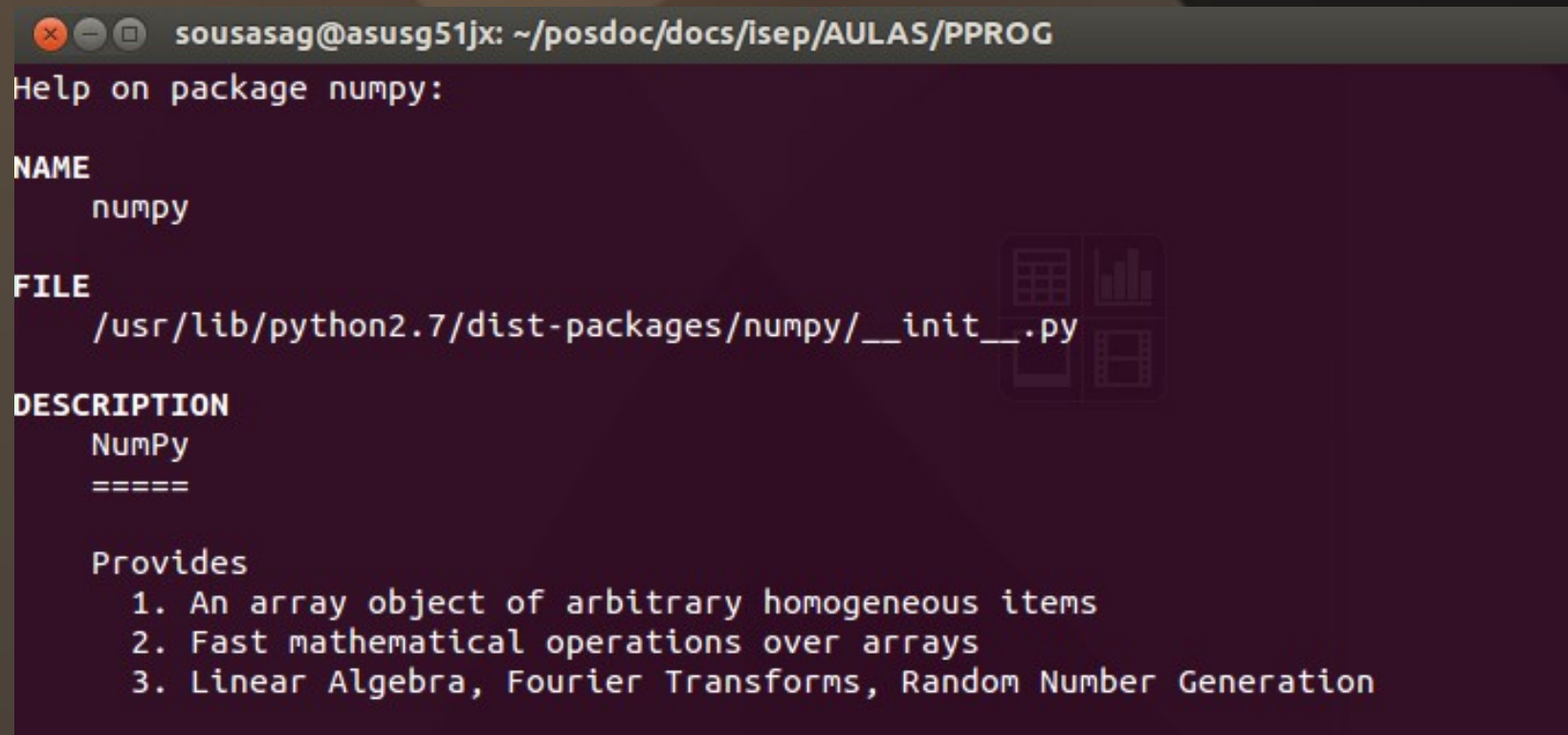
# *Python Advance Course via Astronomy street*

## Advance Course Outline:

- Lesson 1: Python basics (T)
- Lesson 2: Python with Numpy and Matplotlib (T)
- Lesson 3: Science modules (Scipy) (T)

# *Numpy*

```
>>> import numpy
>>> help(numpy)
```



```
sousasag@asusg51jx: ~/posdoc/docs/isep/AULAS/PPROG
Help on package numpy:

NAME
    numpy

FILE
    /usr/lib/python2.7/dist-packages/numpy/__init__.py

DESCRIPTION
    NumPy
    =====

    Provides
    1. An array object of arbitrary homogeneous items
    2. Fast mathematical operations over arrays
    3. Linear Algebra, Fourier Transforms, Random Number Generation
```

# *Numpy Array vs. Python list*

```
example_timeit_numpy.py x
1 #!/usr/bin/python
2 ## My first python code
3
4 ##imports:
5 import numpy as np
6 import timeit
7
8 ## My functions:
9
10 def normal_list():
11     L = range(1000000)
12     L2 = [i**2 for i in L]
13
14
15 def numpy_array():
16     a = np.arange(1000000)
17     a2 = a**2
18
19 ### Main program:
20 def main():
21     print "Normal List:"
22     timer = timeit.Timer("normal_list()", setup="from __main__ import normal_list")
23     print timer.timeit(1)*1000. , 'mili seconds'
24     print "Numpy Array:"
25     timer = timeit.Timer("numpy_array()", setup="from __main__ import numpy_array")
26     print timer.timeit(1)*1000. , 'mili seconds'
27
28
29 if __name__ == "__main__":
30     main()
```

```
$ ./example_timeit_numpy.py
Normal List:
140.730142593 mili seconds
Numpy Array:
7.87591934204 mili seconds
```

# *Things that you can do with Numpy*

- Create Arrays: direct definition

```
>>> import numpy as np
>>> a = np.array([0,1,2,3]) # 1D array
>>> b = np.array([[0, 1, 2], [3, 4, 5]]) # 2D array: 2 x 3 array
>>> c = np.array([[[1], [2]], [[3], [4]]]) # 3D Array
>>> # evenly spaced:
```

```
...
>>> d = np.arange(10) ← Same as range in built in Python
```

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

```
>>> e = np.arange(1,9,2) # start, end (exclusive), step
```

```
>>> e
array([1, 3, 5, 7])
```

```
>>> # by a number of points
```

```
...
>>> f = np.linspace(0, 1, 6) # start, end, num-points ← Defining a linear space array
```

```
>>> f
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

```
>>> a
array([0, 1, 2, 3])
```

```
>>> a.ndim
1
```

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

```
>>> b.ndim
2
```

```
>>> len(a)
4
```

```
>>> b.shape
(2, 3) ← This is a tuple
```

} Some properties of the array object

# *Things that you can do with Numpy*

- Create Arrays: automatic definitions

```
>>> # creation of predefined arrays:
...
>>> a = np.ones((3, 3))
>>> # reminder: (3, 3) is a tuple
... a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>>
>>> b = np.zeros((2,2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>>
>>> c = np.eye(3)
>>> c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>>
>>> d = np.diag([1,2,3,4])
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
...
```

ones and zeros also work for 1D

Defining an Identity matrix

Defining a diagonal matrix

# *Things that you can do with Numpy*

- Create Arrays: setting different data types

```
>>> #creating with different data types
```

```
...
```

```
>>> a = np.array([1, 2, 3])
```

```
>>> a.dtype
```

```
dtype('int64')
```

```
>>>
```

```
>>> a = np.array([1., 2., 3.])
```

```
>>> a.dtype
```

```
dtype('float64')
```

```
>>>
```

```
>>> a = np.array([1, 2, 3], dtype=float) #bool, complex
```

```
>>> a.dtype
```

```
dtype('float64')
```

```
>>> a = np.array([1, 2, 3], dtype=complex) #bool, complex
```

```
>>> a.dtype
```

```
dtype('complex128')
```

} automatic definition

} Forcing the types



# *Things that you can do with Numpy*

- Indexing in Arrays – getting the values that you need

```
>>> #indexing and Slicing
...
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
>>>
>>> #index in numpy 2D: 2ways:
... a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1][1]
1
>>> a[1,1]      #using a tuple
1
>>> a[(1,1)]
1
>>> a[1,2] = 26
>>> a
array([[ 0,  0,  0],
       [ 0,  1, 26],
       [ 0,  0,  2]])
>>> a[1]# geting a line
array([ 0,  1, 26])
>>> a[:,2]# getting a row
array([ 0, 26,  2])
```

Indexing arrays is the same as for lists

You can also use tuples for arrays,  
simpler notation for n>1 Dimensional arrays

Getting lines and/or rows from a matrix



# Things that you can do with Numpy

- Create Arrays: Slicing and Views

```
>>> #slicing:
... a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # start;end;step
array([2, 5, 8])
>>> a[:4]
array([0, 1, 2, 3])
>>> a[2:]
array([2, 3, 4, 5, 6, 7, 8, 9])
>>> a[::2]
array([0, 2, 4, 6, 8])
```

Sintaxe for slicing:  
**array[start:end:step]**

Some useful variations

```
>>> #slicing create views, using the same memory... Careful
...
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]
>>> b
array([0, 2, 4, 6, 8])
>>> b[0] = 26
>>> b
array([26, 2, 4, 6, 8])
>>> a
array([26, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> # to copy use:
... b = a[::2].copy() # this force a copy
>>> b[0] = 12
>>> b
array([12, 2, 4, 6, 8])
>>> a
array([26, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

**Important to be aware:**

Slicing create views that share the same memory of the original array;

If you want to copy you need to force the copy;

# *Things that you can do with Numpy*

- Create Arrays: Applying masks (can be used as where in IDL)

Next slide for random numbers (wait for it...)

Defining a mask  
(which will be an array of booleans)

Applying the mask

Nice for tagging  
(check example)

Can also be used to  
assign a value to  
different places at once

```
>>> #masks (similar to where in IDL)
...
>>> a = np.random.random_integers(0, 20, 15)
>>> a
array([ 6, 18,  6,  3,  5, 14, 15, 16,  8,  1, 20, 20,  5,  7,  8])
>>> mask = (a % 2 == 0)
>>> mask
array([ True,  True,  True, False, False,  True, False,  True,  True,
        False,  True,  True, False, False,  True], dtype=bool)
>>> even_a = a[mask] # this create a copy, not a view
>>> even_a
array([ 6, 18,  6, 14, 16,  8, 20, 20,  8])
>>>
>>> # useful for tagging
... a[a % 2 == 0] = -1
>>> a
array([-1, -1, -1,  3,  5, -1, 15, -1, -1,  1, -1, -1,  5,  7, -1])
>>>
>>> a[[0,1,3]] = 26 # use a list to change n values in one line
>>> a
array([26, 26, -1, 26,  5, -1, 15, -1, -1,  1, -1, -1,  5,  7, -1])
```

# *Things that you can do with Numpy*

- Create Arrays: Applying masks (can be used as where in IDL)

Using np.where()

```
>>> import numpy as np
>>> x = np.arange(15)+12
>>> x
array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26])
>>> ix = np.where(x % 2 == 0)
>>> ix
(array([ 0,  2,  4,  6,  8, 10, 12, 14]),)
>>> x2 = x[ix]
>>> x2
array([12, 14, 16, 18, 20, 22, 24, 26])
>>> np.where(x % 2 == 0, x, -1)
array([12, -1, 14, -1, 16, -1, 18, -1, 20, -1, 22, -1, 24, -1, 26])
>>> x
array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26])
>>> x = np.where(x % 2 == 0, x, -1)
>>> x
array([12, -1, 14, -1, 16, -1, 18, -1, 20, -1, 22, -1, 24, -1, 26])
>>>
```

Define some array with numbers

Using where.  
Similar as a mask

You can also tag with where

Can also be used to assign a value to different places at once

# *Things that you can do with Numpy*

- Create Arrays: Random numbers generation

```
>>> # creation of random numbers:
...
>>> a = np.random.rand(4) # uniform in [0, 1]
>>> a
array([ 0.292304 ,  0.66154467,  0.88626098,  0.67117646])
>>>
>>> b = np.random.randn(4) # Gaussian
>>> b
array([-1.07921219, -0.62811542,  0.29934704,  0.12727723])
>>>
>>> np.random.seed(1234) # Setting the random seed
>>>
>>> c = np.random.random_integers(1,50,5) # 5 random integers
>>> c
array([48, 20, 39, 13, 25])
>>>
>>> s = np.random.poisson(5, 10) # see help(np.random) for more distributions
>>> s
array([ 5,  1,  4,  2,  6,  6,  6, 10,  2,  3])
```

← The standard random common to many languages

← A gaussian random number generation.  $G(0,1)$

← **Important:** Careful with the seed to generate the random numbers

← To generate the next Euromillions numbers

You can also obtain random numbers following many other statistical distributions. ex. poisson

# *Things that you can do with Numpy*

- Create Arrays: Random numbers generation – The statistical distributions available

```
=====
Univariate distributions
=====
beta          Beta distribution over ``[0, 1]``.
binomial      Binomial distribution.
chisquare     :math:`\chi^2` distribution.
exponential   Exponential distribution.
f            F (Fisher-Snedecor) distribution.
gamma        Gamma distribution.
geometric     Geometric distribution.
gumbel       Gumbel distribution.
hypergeometric Hypergeometric distribution.
laplace      Laplace distribution.
logistic     Logistic distribution.
lognormal    Log-normal distribution.
logseries    Logarithmic series distribution.
negative_binomial Negative binomial distribution.
noncentral_chisquare Non-central chi-square distribution.
noncentral_f Non-central F distribution.
normal       Normal / Gaussian distribution.
pareto       Pareto distribution.
poisson      Poisson distribution.
power        Power distribution.
rayleigh     Rayleigh distribution.
triangular   Triangular distribution.
uniform      Uniform distribution.
vonmises     Von Mises circular distribution.
wald         Wald (inverse Gaussian) distribution.
weibull      Weibull distribution.
zipf         Zipf's distribution over ranked data.
=====
```

>>> help(np.random)



# Things that you can do with Numpy

- Arrays: Numerical Operations

```
>>> a = np.array([1,2,3,4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2,  4,  8, 16])
>>>
>>> # arrays are objects, operators were overloaded
... b = np.ones(4) + 1
>>> b
array([ 2.,  2.,  2.,  2.])
>>> a - b
array([-1.,  0.,  1.,  2.])
>>> a * b
array([ 2.,  4.,  6.,  8.])
>>>
>>> # warning: NOT matrix multiplication
...
>>> c = np.ones((3,3))
>>> c
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> c * c
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>>
>>> # to multiply matrices:
...
>>> c.dot(c)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
```

Numpy arrays are objects.

Most of the operators were overloaded to represent what we would like.

(One of the advantages of OO programming)

We can do straightforward operations with arrays (similar to IDL)

**Careful with the multiplication of arrays:**

Use `array.dot(array)` to do a proper algebra multiplication of arrays

# Things that you can do with Numpy

- Create Arrays: comparisons and reductions

```
>>> # comparisons
...
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False,  True, False,  True], dtype=bool)
>>> a > b
array([False, False,  True, False], dtype=bool)
>>> #logical operations
... np.all([True, True, False])
False
>>> np.any([True, True, False])
True
```

The comparisons operators are also overloaded. You can use it to compare directly arrays.

In addition you also have some useful logical operations in array

There are several reductions that you can do directly from the arrays.

- sum all elements;
- get the mean;
- standard deviation;
- median (is a class method);
- get extremes in array;
- get index of extremes;

Another useful method is the sort(). The default algorithm is the quicksort, but you can choose others. (check with >>>help(np.sort))

```
>>> # some basic reductions:
...
>>> x = np.array([3, 10, 5, 1, 2, 7, 4])
>>> np.sum(x)
32
>>> x.sum()
32
>>> x.mean()
4.5714285714285712
>>> x.std()
2.8713930346059686
>>> np.median(x) # there is no x.median()...
4.0
>>> x.min()
1
>>> x.max()
10
>>> x.argmin() # index of minimum
3
>>> x.argmax() # index of maximum
1
>>> x.sort()
>>> x
array([ 1,  2,  3,  4,  5,  7, 10])
```



# *Things that you can do with Numpy*

- Create Arrays: reductions in 2D

```
>>> #notes on 2D or higher
...
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>>
>>> x.sum()
6
>>>
>>> x.sum(axis=0)
array([3, 3])
>>>
>>> x.sum(axis=1)
array([2, 4])
>>>
>>> x[:, 0].sum()
3
>>>
>>> x.T # transpose
array([[1, 2],
       [1, 2]])
```

When you have 2D arrays (or with more dimensions) you can also use the methods.

You can also select the axis that you want to reduce using the option inside the method.

Or alternatively you use what you learn on the indexing/slicing

Special note:

How to quickly transpose a matrix

# Things that you can do with Numpy

- Create Arrays: data types casting and rounding

```
>>> # data types and Casting
... np.array([1, 2, 3]) + 1.5 # bigger type win
array([ 2.5,  3.5,  4.5])
>>>
>>> # assignment don't change type
... a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')
>>> a[0] = 1.9 # float will be truncated
>>> a
array([1, 2, 3])
>>>
>>> #forced cast:
... a = np.array([1.7, 1.2, 1.6])
>>> b = a.astype(int) # <-- truncates to integer
>>> b
array([1, 1, 1])
>>>
>>> # Rounding
...
>>> a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])
>>> b = np.around(a)
>>> b
array([ 1.,  2.,  2.,  2.,  4.,  4.])
>>> c = np.around(a).astype(int)
>>> c
array([1, 2, 2, 2, 4, 4])
```

**Casting:** is the transformation of a variable between different data types

When making operations with different data types, the output will be the bigger data type involved.

**Careful with Down - casting**

Example: Floats will be truncated when passing to integers.

You can also force a cast of an array.

You can use the `around()` to round values.

The result is casted, and you may want to force the cast to a different data-type.

# *Things that you can do with Numpy*

- Polynomials

```
>>> #polynomials
... #3x**2 + 2*x - 1:
... p = np.poly1d([3, 2, -1])
>>> p(0)
-1
>>> p.roots
array([-1.          ,  0.33333333])
>>> p.order
2
>>> p.c
array([ 3,  2, -1])
>>>
>>> q = p * p
>>> q
poly1d([ 9, 12, -2, -4,  1])
>>> q.order
4
```

Polynomials with 1 variable can also be represented by a numpy object (poly1d).

Definition of a polynomial can be done using its coefficients in a list:  
Example:  $3x^2 + 2x - 1$ :  
List:  $[3, 2, -1] \rightarrow [a_2, a_1, a_0]$

There are some useful methods:

- getting the roots
- getting the order
- getting the coefficients

You can also perform direct operations with polynomials.

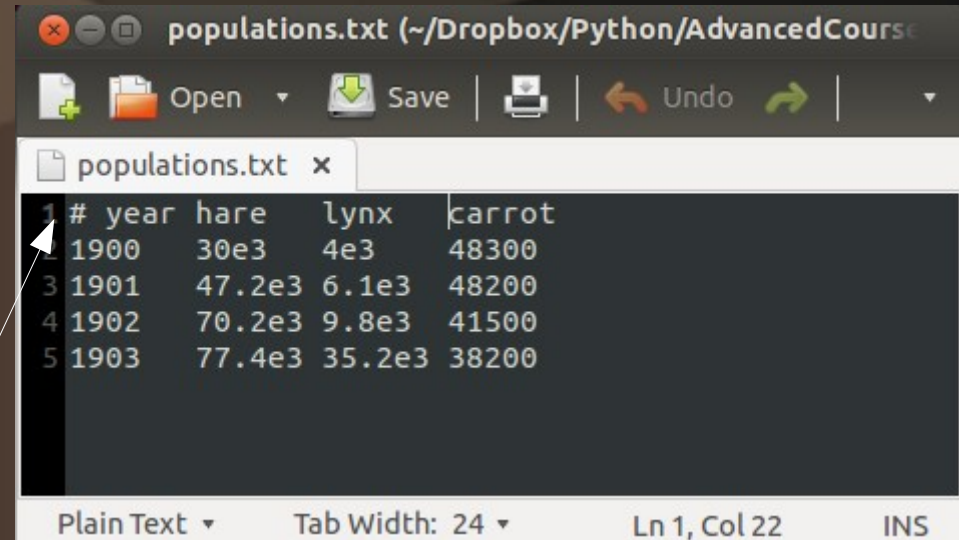
# Things that you can do with Numpy

- Reading and writing data to files with Numpy

```
>>> import numpy as np
>>> data = np.loadtxt('populations.txt')
>>> data
array([[ 1900.,  30000.,  4000.,  48300.],
       [ 1901.,  47200.,  6100.,  48200.],
       [ 1902.,  70200.,  9800.,  41500.],
       [ 1903.,  77400.,  35200.,  38200.]])
```

Using `loadtxt()` you can directly obtain the data.

**Careful** with header in files. They should start with the special character '#'



```
populations.txt (~/.Dropbox/Python/AdvancedCourse)
Open Save Undo
populations.txt x
1 # year hare lynx carrot
2 1900 30e3 4e3 48300
3 1901 47.2e3 6.1e3 48200
4 1902 70.2e3 9.8e3 41500
5 1903 77.4e3 35.2e3 38200
Plain Text Tab Width: 24 Ln 1, Col 22 INS
```

You can also save the data directly to a file with `savetxt()`.

```
>>> np.savetxt('pop2.txt', data)
>>>
sousasag@asusg51jx:~/Dropbox/Python/AdvancedCourse/examples$ more pop2.txt
1.9000000000000000e+03 3.0000000000000000e+04 4.0000000000000000e+03 4.8300000000000000e+04
1.9010000000000000e+03 4.7200000000000000e+04 6.1000000000000000e+03 4.8200000000000000e+04
1.9020000000000000e+03 7.0200000000000000e+04 9.8000000000000000e+03 4.1500000000000000e+04
1.9030000000000000e+03 7.7400000000000000e+04 3.5200000000000000e+04 3.8200000000000000e+04
```



# *Good Practices in Python*

- Explicit variable names (no need of a comment to explain what is in the variable)
- Style: spaces after commas, around =, etc. A certain number of rules for writing “beautiful” code (and, more importantly, using the same conventions as everybody else!) are given in the Style Guide for Python Code and the Docstring Conventions page (to manage help strings).
- Except some rare cases, variable names and comments in English.
- Good indentation (forced in Python)

# *Why Python?*

The background of the slide is composed of several overlapping, semi-transparent shapes in various shades of brown and dark grey. These shapes create a layered, organic effect, with some areas appearing lighter due to the overlap of lighter shades and others appearing darker where more layers overlap. The overall aesthetic is modern and minimalist.