

Introduction to NumPy arrays

Gert-Ludwig Ingold

 <https://github.com/gertingold/euroscipy-numpy-tutorial.git>

Python comes with batteries included

→ extensive Python standard library

What about batteries for scientists (and others as well)?

→ scientific Python ecosystem



NumPy
Base N-dimensional
array package



SciPy library
Fundamental
library for scientific
computing



Matplotlib
Comprehensive 2D
Plotting



IPython
Enhanced
Interactive Console



Sympy
Symbolic
mathematics



pandas
Data structures &
analysis

from: www.scipy.org

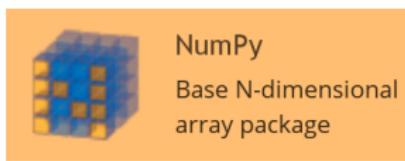
+ SciKits and many other packages

Python comes with batteries included

→ extensive Python standard library

What about batteries for scientists (and others as well)?

→ scientific Python ecosystem



SciPy library
Fundamental library for scientific computing



Matplotlib
Comprehensive 2D Plotting



IPython
Enhanced Interactive Console



Sympy
Symbolic mathematics

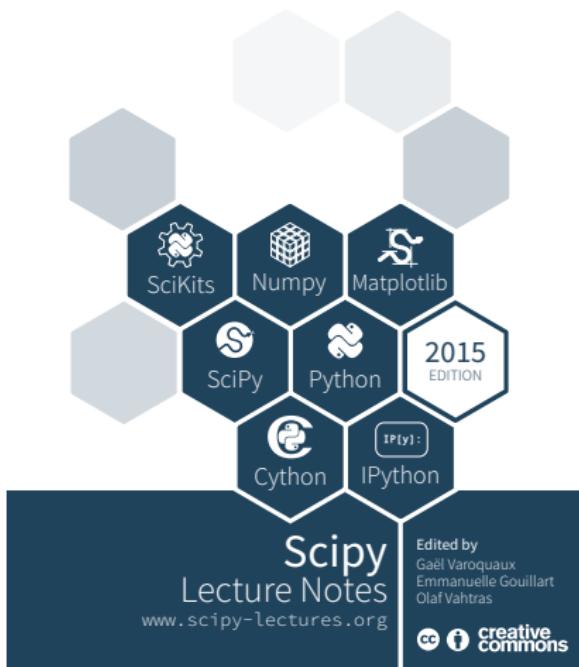


pandas
Data structures & analysis

from: www.scipy.org

+ SciKits and many other packages

www.scipy-lectures.org



Gaël Varoquaux • Emmanuelle Gouillart • Olav Vahtras
Valentin Haenel • Nicolas P. Rougier • Ralf Gommers
Fabian Pedregosa • Zbigniew Jędrzejewski-Szmek • Pauli Virtanen
Christophe Combelle • Didrik Pinte • Robert Cimrman
André Espaze • Adrian Chauve • Christopher Burns

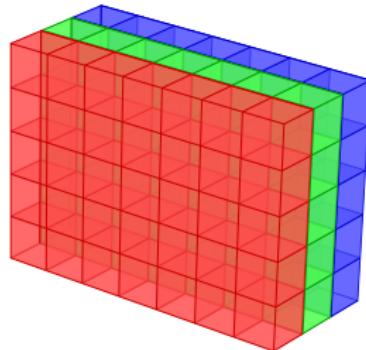
docs.scipy.org/doc/numpy/

The screenshot shows the "NumPy v1.13 Manual" documentation page. At the top, there's a header with the SciPy.org logo and a "Sponsored by ENTHOUGHT" link. Below the header, there are navigation links for "Scipy.org" and "Docs", and a "index" link. The main content area has a large title "NumPy v1.13 Manual" and a welcome message: "Welcome! This is the documentation for NumPy 1.13.0, last updated Jun 10, 2017." A section titled "Parts of the documentation:" lists several guides: "NumPy User Guide" (start here), "NumPy Reference" (reference documentation), "F2Py Guide" (f2py documentation), and "NumPy Developer Guide" (contributing to NumPy). Below these are sections for "Indices and tables:", "General Index" (all functions, classes, terms), "Glossary" (the most important terms explained), and "Meta information:" which includes links to "Reporting bugs", "About NumPy", "NumPy Enhancement Proposals", "Release Notes", and "License of NumPy".

A wish list

- ▶ we want to work with vectors and matrices

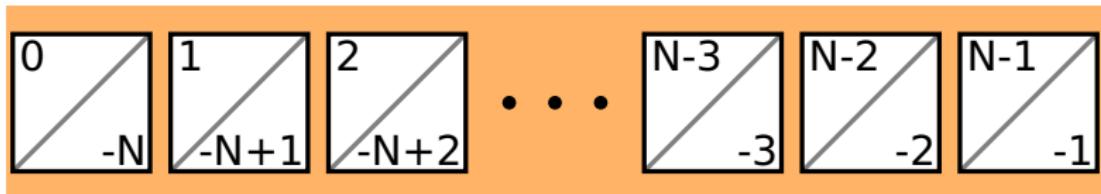
$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$



colour image as
 $N \times M \times 3$ -array

- ▶ we want our code to run fast
- ▶ we want support for linear algebra
- ▶ ...

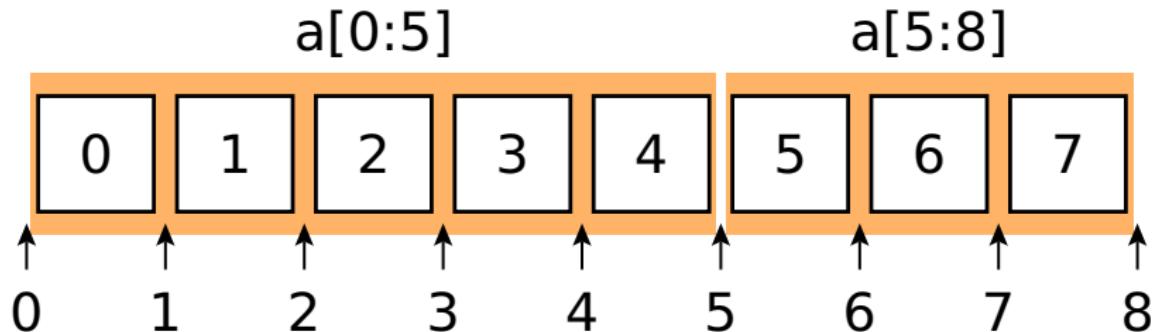
List indexing



- ▶ indexing starts at 0
- ▶ negative indices count from the end of the list to the beginning

List slicing

basic syntax: [start:stop:step]



- ▶ if `step=1`
 - ▶ slice contains the elements `start` to `stop-1`
 - ▶ slice contains `stop-start` elements
- ▶ default values:
 - ▶ `start` 0 (first element)
 - ▶ `stop` $N-1$ (last element)
 - ▶ `step` 1

Let's do some slicing



Matrices and lists of lists

Can we use lists of lists to work with matrices?

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
matrix = [[0, 1, 2],  
          [3, 4, 5],  
          [6, 7, 8]]
```

- ▶ How can we extract a row?
- ▶ How can we extract a column?

Matrices and lists of lists

Can we use lists of lists to work with matrices?

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
matrix = [[0, 1, 2],  
          [3, 4, 5],  
          [6, 7, 8]]
```

- ▶ How can we extract a row?
- ▶ How can we extract a column?

Let's do some experiments

Your turn

Matrices and lists of lists

Can we use lists of lists to work with matrices?

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

```
matrix = [[0, 1, 2],  
          [3, 4, 5],  
          [6, 7, 8]]
```

- ▶ How can we extract a row? 😊
- ▶ How can we extract a column? 😞

Lists of lists do not work like matrices

Problems with lists as matrices

- ▶ different axes are not treated on equal footing
- ▶ lists can contain arbitrary objects
matrices have a homogeneous structure
- ▶ list elements can be scattered in memory

Applied to matrices ...

... lists are conceptually inappropriate

... lists have less performance than possible

We need a new object

ndarray

multidimensional, homogeneous array of fixed-size
items

Getting started

Import the NumPy package:

```
from numpy import *
```

Getting started

Import the NumPy package:

```
from numpy import *
from numpy import array, sin, cos
```

Getting started

Import the NumPy package:

```
from numpy import *
from numpy import array, sin, cos
import numpy
```

Getting started

Import the NumPy package:

```
from numpy import *
from numpy import array, sin, cos
import numpy
import numpy as np ←
```

Your turn

Data types

Some important data types:

integer int8, int16, int32, int64, uint8, ...

float float16, float32, float64, ...

complex complex64, complex128, ...

boolean bool8

Unicode string

Default: Python float



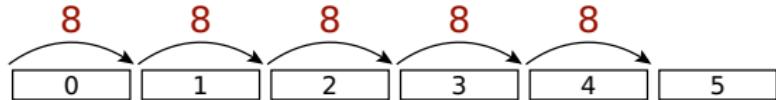
Beware of overflows!

Your turn

Strides

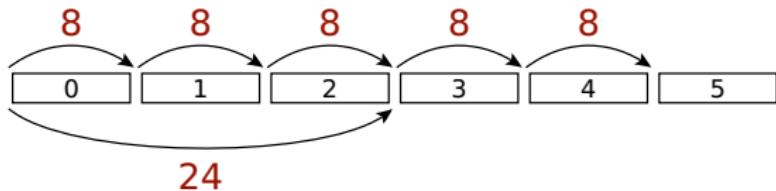
$(0 \ 1 \ 2 \ 3 \ 4 \ 5)$

$(8,)$



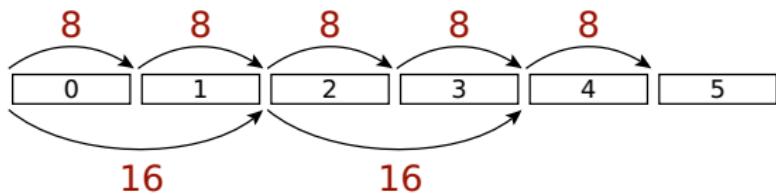
$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$

$(24, 8)$



$\begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{pmatrix}$

$(16, 8)$



Views

For the sake of efficiency, NumPy uses views if possible.

- ▶ Changing one or more matrix elements will change it in all views.
- ▶ Example: transposition of a matrix $a.T$
No need to copy the matrix and to create a new one



Some array creation routines

- ▶ numerical ranges: arange, linspace, logspace
- ▶ homogeneous data: zeros, ones
- ▶ diagonal elements: diag, eye
- ▶ random numbers: rand, randint

 Numpy has an append()-method. Avoid it if possible.



Indexing and slicing in one dimension

1d arrays: indexing and slicing as for lists

- ▶ first element has index 0
- ▶ negative indices count from the end
- ▶ slices: `[start:stop:step]`
without the element indexed by stop
- ▶ if values are omitted:
 - ▶ start: starting from first element
 - ▶ stop: until (and including) the last element
 - ▶ step: all elements between start and stop-1



Indexing and slicing in higher dimensions

- ▶ usual slicing syntax
- ▶ difference to lists:
slices for the various axes separated by comma

$a[2, -3]$

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

Indexing and slicing in higher dimensions

- ▶ usual slicing syntax
- ▶ difference to lists:
slices for the various axes separated by comma

`a[:3, :5]`

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

Indexing and slicing in higher dimensions

Your turn

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

Indexing and slicing in higher dimensions

Your turn

`a[-3:, -3:]`

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

Indexing and slicing in higher dimensions

Your turn

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

Indexing and slicing in higher dimensions

Your turn

$a[:, 3]$

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

Indexing and slicing in higher dimensions

Your turn

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

Indexing and slicing in higher dimensions

Your turn

$a[1, 3:6]$

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

Indexing and slicing in higher dimensions

Your turn

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

Indexing and slicing in higher dimensions

Your turn

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

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

Fancy indexing – Boolean mask

$a[a \% 3 == 0]$

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

Fancy indexing – array of integers

`a[(1, 1, 2, 2, 3, 3), (3, 4, 2, 5, 3, 4)]`

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

Application: sieve of Eratosthenes

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	45	46	47	48	49

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	45	46	47	48	49

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	45	46	47	48	49

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	45	46	47	48	49

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	45	46	47	48	49

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	45	46	47	48	49

Axes

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

axis 1

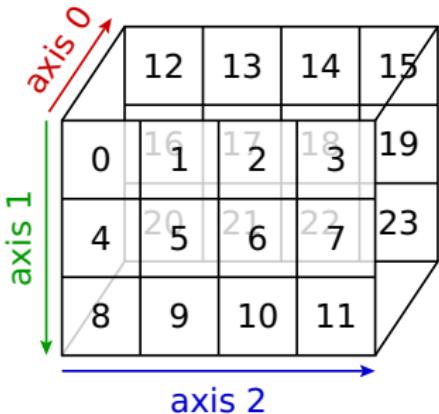
The diagram illustrates a 3x3 matrix. A vertical red arrow labeled "axis 0" points downwards from the top row to the bottom row. A horizontal blue arrow labeled "axis 1" points to the right from the first column to the third column. The matrix elements are labeled with indices: a[0, 0], a[0, 1], a[0, 2] in the first row; a[1, 0], a[1, 1], a[1, 2] in the second row; and a[2, 0], a[2, 1], a[2, 2] in the third row.

$$\left| \begin{array}{ccc} a[0, 0] & a[0, 1] & a[0, 2] \\ a[1, 0] & a[1, 1] & a[1, 2] \\ a[2, 0] & a[2, 1] & a[2, 2] \end{array} \right|$$

Your turn

`np.sum(a)`
`np.sum(a, axis=...)`

Axes in more than two dimensions



```
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],
      [[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```



create this array and produce 2d arrays by cutting perpendicular to the axes 0, 1, and 2

Matrix multiplication

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 6 & 7 \\ 26 & 31 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 6 & 7 \\ 26 & 31 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 6 & 7 \\ 26 & 31 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 6 & 7 \\ 26 & 31 \end{pmatrix}$$

Your turn

try `np.dot(•, •)`
`•.dot(•)`
`• @ • *`)

*) Python≥3.5, NumPy≥1.10

Mathematical functions in NumPy

Universal functions (ufuncs) take ndarrays as argument

Trigonometric functions

`sin, cos, tan, arcsin, arccos, arctan, hypot, arctan2,`
`degrees, radians, unwrap, deg2rad, rad2deg`

Hyperbolic functions

`sinh, cosh, tanh, arcsinh, arccosh, arctanh`

Rounding

`around, round_, rint, fix, floor, ceil, trunc`

Sums, products, differences

`prod, sum, nanprod, cumprod, cumsum, diff,`
`ediff1d, gradient, cross, trapz`

Exponents and logarithms

`exp, expm1, exp2, log, log10, log2, log1p,`
`logaddexp, logaddexp2`

Other special functions

`i0, sinc`

Floating point routines

`signbit, copysign, frexp, ldexp`

Arithmetic operations

`add, reciprocal, negative, multiply, divide, power,`
`subtract, true_divide, floor_divide, fmod, mod,`
`modf, remainder`

Handling complex numbers

`angle, real, imag, conj`

Miscellaneous

`convolve, clip, sqrt, square, absolute, fabs, sign,`
`maximum, minimum, fmax, fmin, nan_to_num,`
`real_if_close, interp`

Many more special functions are provided as ufuncs by SciPy

Rules for broadcasting

Arrays can be broadcast to the same shape if one of the following points is fulfilled:

1. The arrays all have exactly the same shape.
2. The arrays all have the same number of dimensions and the length of each dimension is either a common length or 1.
3. The arrays that have too few dimensions can have their shapes prepended with a dimension of length 1 to satisfy property 2.

Broadcasting

shape=(3, 4)

0	1	2	3
4	5	6	7
8	9	10	11

shape=(1,)

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

shape=(4,)

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

shape=(3,)

1	1	1	

shape=(3, 1)

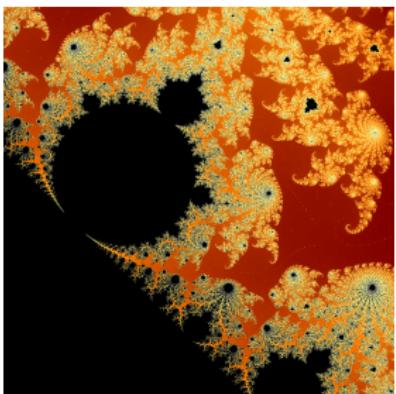
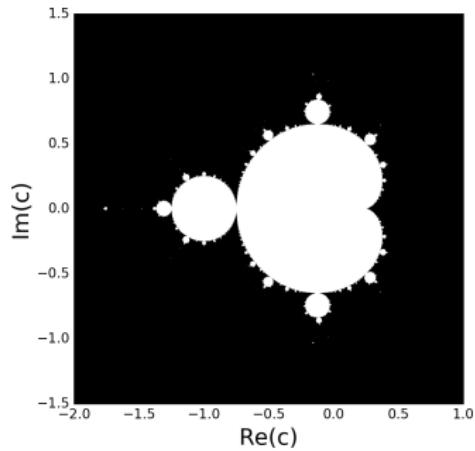
1	1	1	1
1	1	1	1
1	1	1	1

Your turn

Application: Mandelbrot set

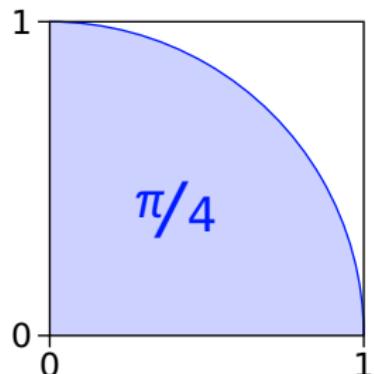
$$z_{n+1} = z_n^2 + c, \quad z_0 = 0$$

Mandelbrot set contains the points for which z remains bounded.



Your turn

Application: π from random numbers

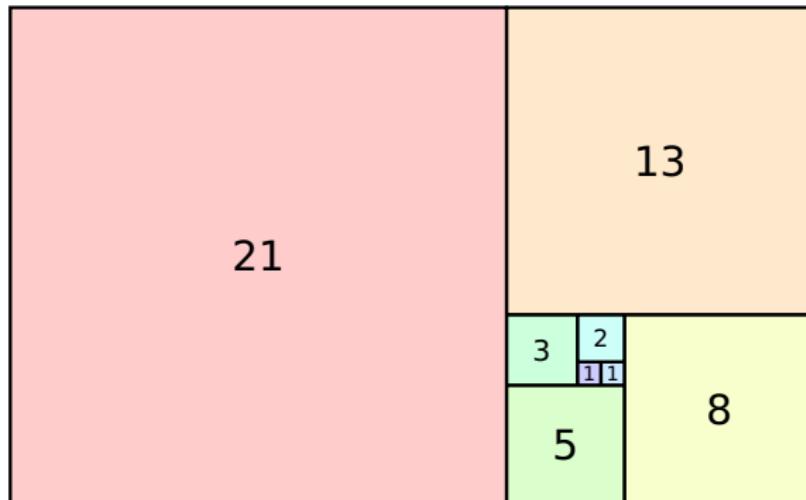


1. Create pairs of random numbers and determine the fraction of pairs which has a distance from the origin less than one.
2. Multiply the result by four to obtain an approximation of π .

hint: `count_nonzero(a)` counts the number of non-zero values in the array `a` and also works for Boolean arrays. Remember that `np.info(...)` can be helpful.



Fibonacci series and linear algebra



Fibonacci series:

1, 1, 2, 3, 5, 8, 13, 21, ...

$$F_{n+1} = F_n + F_{n-1}, \quad F_1 = F_2 = 1$$

$$\text{or: } \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

What is the limit of F_{n+1}/F_n for large n ?

Eigenvalue problems

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} v_1^{(k)} \\ \vdots \\ v_n^{(k)} \end{pmatrix} = \lambda^{(k)} \begin{pmatrix} v_1^{(k)} \\ \vdots \\ v_n^{(k)} \end{pmatrix} \quad k = 1, \dots, n$$

eigenvalue $\lambda^{(k)}$

eigenvector $\begin{pmatrix} v_1^{(k)} \\ \vdots \\ v_n^{(k)} \end{pmatrix}$

for our Fibonacci problem:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \lambda \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

We are looking for the eigenvalue larger than one.

Linear algebra in NumPy

```
import numpy.linalg as LA
```

Matrix and vector products

dot, vdot, inner, outer, matmul, tensordot, einsum,
LA.matrix_power, kron

Decompositions

LA.cholesky, LA.qr, LA.svd

Matrix eigenvalues

LA.eig, LA.eigh, LA.eigvals, LA.eigvalsh

Norms and other numbers

LA.norm, LA.cond, LA.det, LA.matrix_rank, LA.slogdet, trace

Solving equations and inverting matrices

LA.solve, LA.tensorsolve, LA.ltsq, LA.inv, LA.pinv, LA.tensorinv



hint: see also the methods for linear algebra in SciPy

Statistics in NumPy

Order statistics

amin, amax, nanmin, nanmax, ptp, percentile, nanpercentile

Averages and variances

median, average, mean, std, var, nanmedian, nanmean,
nanstd, nanvar

Correlating

corrcoef, correlate, cov

Histograms

histogram, histogram2d, histogramdd, bincount, digitize

Application: Brownian motion



1. Simulate several trajectories for a one-dimensional Brownian motion
hint: `np.random.choice`
2. Plot the mean distance from the origin as a function of time
3. Plot the variance of the trajectories as a function of time



Sorting, searching, and counting in NumPy

Sorting

`sort, lexsort, argsort, ndarray.sort, msort, sort_complex,
partition, argpartition`

Searching

`argmax, nanargmax, argmin, nanargmin, argwhere, nonzero,
flatnonzero, where, searchsorted, extract`

Counting

`count_nonzero`

Application: identify entry closest to 1/2

$$\begin{pmatrix} 0.05344164 & \textcolor{red}{0.37648768} & 0.80691163 & 0.71400815 \\ \textcolor{red}{0.60825034} & 0.35778938 & 0.37393356 & 0.32615374 \\ 0.83118547 & 0.33178711 & 0.21548027 & \textcolor{red}{0.42209291} \end{pmatrix}$$



$$\begin{pmatrix} \textcolor{red}{0.37648768} \\ \textcolor{red}{0.60825034} \\ \textcolor{red}{0.42209291} \end{pmatrix}$$



hint: use `np.argsort`

Polynomials in NumPy

Power series: `numpy.polynomial.polynomial`

Polynomial Class

`Polynomial`

Basics

`polyval`, `polyval2d`, `polyval3d`, `polygrid2d`, `polygrid3d`,
`polyroots`, `polyfromroots`

Fitting

`polyfit`, `polyvander`, `polyvander2d`, `polyvander3d`

Calculus

`polyder`, `polyint`

Algebra

`polyadd`, `polysub`, `polymul`, `polymulx`, `polydiv`, `polypow`

Miscellaneous

`polycompanion`, `polydomain`, `polyzero`, `polyone`, `polyx`,
`polytrim`, `polyline`

also: Chebyshev, Legendre, Laguerre, Hermite polynomials

Some examples

P.Polynomial([24, -50, 35, -10, 1])

$$p_4(x) = x^4 - 10x^3 + 35x^2 - 50x + 24 = (x-1)(x-2)(x-3)(x-4)$$

p4.deriv()

$$\frac{dp_4(x)}{dx} = 4x^3 - 30x^2 + 70x - 50$$

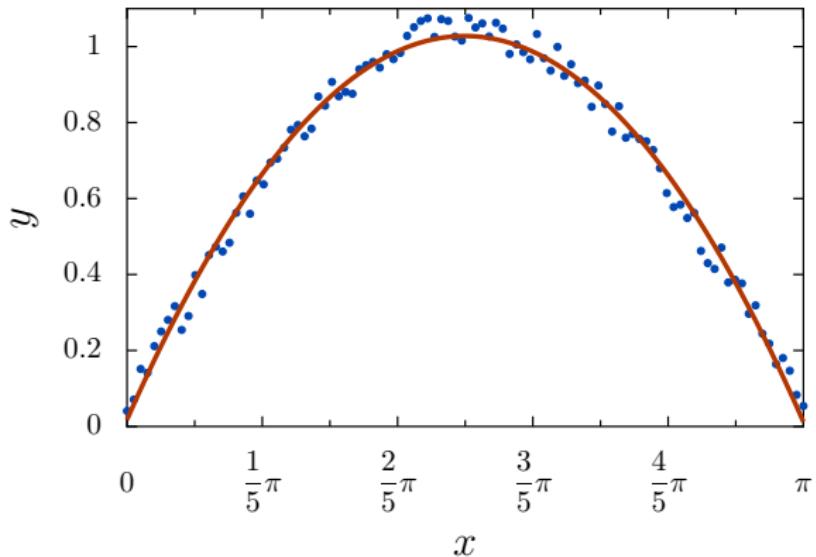
p4.integ()

$$\int p_4(x) dx = \frac{1}{5}x^5 - \frac{5}{2}x^4 + \frac{35}{3}x^3 - 25x^2 + 24x + C$$

p4.polydiv()

$$\frac{p_4(x)}{2x+1} = \frac{1}{2}x^3 - \frac{21}{4}x^2 + \frac{161}{8}x - \frac{561}{16} + \frac{945}{16p_4(x)}$$

Application: polynomial fit

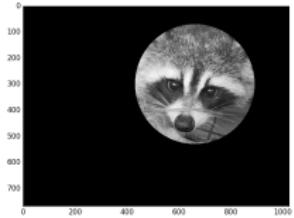
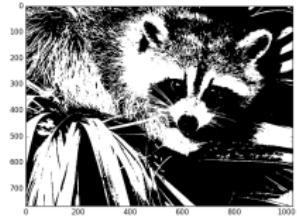


add some noise to a function and
fit it to a polynomial

see `scipy.optimize.curve_fit` for general fit functions

Application: image manipulation

```
from scipy import misc  
face = misc.face(gray=True)
```



Your turn