CEng 240 – Spring 2021
Week 13

Scientific and Engineering Libraries

Part 1: NumPy and SciPy

Sinan Kalkan

Error Types

Previously on CENG240!

METU Computer Engineering

- Syntax errors
- Type errors
- Run-time errors
- Logical errors

```
>>> for i in range(10)
...     print(i)

File "<ipython-input-1-12d72cac235a>", line 1
    for i in range(10)
                      ^
SyntaxError: invalid syntax
```

```
>>> x = float(input())
>>> a = ((x+5)*12+4

File "<ipython-input-2-dead5b360d91>", line 2
  a = ((x+5)*12+4
                ^
SyntaxError: invalid syntax
```

```
>>> s = 0
>>> for i in range(10):
...   s += i
...     print(i)

File "<ipython-input-3-c3ef5d622e47>", line 4
  print(i)
  ^
IndentationError: unexpected indent
```

```
>>> while x = 4:
...     s += x

File "<ipython-input-4-befcf7769cec>", line 1
  while x = 4:
          ^
SyntaxError: invalid syntax
```

Error Types

Previously on CENG240!

- Syntax errors
- Type errors
- Run-time errors
- Logical errors

```
>>> print(astr ** 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
>>> print(bflt[1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object is not subscriptable
>>> print(cdict * 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'dict' and 'int'
>>> cdict < astr
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'dict' and 'str'
```

Previously on CENG240!

Error Types

- Syntax errors
- Type errors
- Run-time errors
- Logical errors

```
def divisible(m, n):
  return m % n == 0

def count(m):
  sum = 0
  for i in range(1,1000):
    if divisible(i, m):
      sum += 1

  return sum

value = int(input())
print('input value is:', value)
print(value,' divides ', count(value), ' many integers in range [1, 1000]')
```

```
input value is: 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

 in ()
     12 value = int(input())
     13 print('input value is:', value)
---> 14 print(value,' divides ', count(value), ' many integers in range [1, 1000]')

 in count(m)
      5   sum = 0
      6   for i in range(1,1000):
----> 7     if divisible(i,m):
      8       sum += 1
      9

 in divisible(m, n)
      1 def divisible(m, n):
----> 2   return m % n == 0
      3
      4 def count(m):
      5   sum = 0

ZeroDivisionError: integer division or modulo by zero
```

Previously on CENG240!

METU Computer Engineering

Exceptions

Exception	Reason
KeyboardInterrupt	User presses `Ctrl-C`; not an error but user intervention
ZeroDivisionError	Right-hand side of `/` or `%` is 0
AttributeError	Object/class does not have a member
EOFError	`input()` function gets End-of-Input by user
IndexError	Container index is not valid (negative or larger than length)
KeyError	`dict` has no such key
FileNotFoundError	The target file of `open()` does not exist
TypeError	Wrong operand or parameter types, or wrong number of parameters for functions
ValueError	The given value has correct type but the operation is not supported for the given value

Previously on CENG240!

Exception Examples

```
b = 0
a = a / b                  # ZeroDivisionError

x = [1,2,3]
print(x.length)            # AttributeError: lists does not have a length attribute (use len(x))

print(x[4])                # IndexError: last valid index of list x is 2

person = { 'name' : 'Han', 'surname': 'Solo'}
print(person['Name'])      # KeyError: person does not have 'Name' key but 'name'

fp = open("example.txt") # FileNotFoundError: file "example.txt" does not exist

print([1,2,3] / 12)        # TypeError: Division is not defined for lists

def f(x, y):
  return x*x+y

print(f(5))                # TypeError: Only one element is supplied instead of 2.

print(int('thirtytwo'))  # ValueError: string value does not represent an integer

a,b,c = [1,2,3,4]          # ValueError: too many values on the right hand side
```

Error Types

Previously on CENG240!

METU Computer Engineering

- Syntax errors
- Type errors
- Run-time errors
- Logical errors

```
y = x / x+1              # you meant y = x / (x+1), forgetting about precedence

lastresult = 0
def missglobal(x):
  result = x*x+1                  # you intend to update the global variable
  if lastresult != result:        # but you assign a local variable instead
    lastresult = result           # you should have used "global lastresult"


def returnsnothing(x, y):
  y = x*x+y*y
  if x <= y:
    return x                      # if x > y, the function returns nothing
print(returnsnothing(0.1, 0.1))   # does not have any value. prints "None"

s = 1
while i < n:                      # you forgot incrementing i as i+=1
   s += s*x/i                     # loop will run forever. "infinite loop"
```

Previously on CENG240!

How to work with errors

1. Program with care
2. Place controls in your code
3. Handle exceptions
4. Write verification code & raise exceptions
5. Debug your code
6. Write test cases

How to work with errors:

(2) Place controls in your code

```
# CASE 1 with sanitization
n = int(input())
if 0 <= n < len(a):
  print(a[n])
else:
  print("n is not valid:", n)

# CASE 2 with sanitization
name = input()
if name in age:       # membership test for dictionaries
  print(age[name])
else:
  print("dictionary does not have member:", name)

# CASE 3 with sanitization
x = float(input())
if x >= 0:
  y = math.sqrt(x)
else:
  print("invalid for sqrt operation: ", x)

if x != 0:
  y = 1 / x
else:
  print("divisor cannot be 0")
```

How to work with errors:

(3) Handle Exceptions

```
try:
  ......    # a block with possible errors
  ......    # if there are function calls here
  ......    # and error occurs in the function, we can handle error here
except exceptionname:     # exceptionname is optional
  .....     # this is error handling block.
  .....     # when there is an error, execution jumps here
```

Previously on CENG240:

How to work with errors:

(3) Handle Exceptions

```python
import math

a = [1,2,3]
age = {'Han': 30, 'Leia': 20, 'Luke': 20}

try:
  n = int(input())
  print(a[n])           # will fail for n > 2 or n < -2

  name = input()
  print(age[name])      # will fail names other than 'Han', 'Leia', 'Luke'

  x = float(input())
  y = math.sqrt(x)      # will fail for x < 0
  y = 1 / x             # will fail for x == 0
except IndexError:
  print('List index is not valid')
except KeyError:
  print('Dictionary does not have such key')
except ValueError:
  print('Invalid value for square root operation')
except ZeroDivisionError:
  print('Division by zero does not have value')
except:
  print('None of the known errors. Something happened even if nothing happened')
```

Previously on CENG240:

How to work with errors:

(4) Write verification code and raise exception

- You can raise exceptions
- “raise Exception” => raise a generic exception

```
try:
  if !cond1:
    raise Error

  ..1..

  if !cond2:
    raise Error

  ..2..

  if !cond3:
    raise Error

  ..3..
  ..4..  # success
except :
  ... Error handling
```

```
def solvesecond(a,b,c):
  det = b*b - 4*a*c
  # the following is the verification code
  if det < 0:
    print("Equation has no real roots for", a, b, c)
    raise ValueError
  ....
  ...
```

How to work with errors:

(6) Write test cases

Previously on CENG240:

METU Computer Engineering

```
(x1, x2) = findrootsecond(a,b,c)

if a*x1*x1 + b*x1 + c != 0 or a*x2*X2 + b*x2 +c != 0:
   print('test failed for', a, b, c, 'roots', x1, x2)
```

Previously on CENG240!

Debugging

- Using debugging outputs
- Handling exception and getting more info
- Using debugger

Debugging:
Using debugging outputs

- The following code has a bug, how can we find it?

```
def sum_and_delete(L):
        sum = 0
        for i in range(len(L)):
                sum += L[i]
                del L[i]

        return sum

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

Debugging:

Handling Exception to Get More Info

```
def sum_and_delete(L):
        sum = 0
        try:
                for i in range(len(L)):
                        sum += L[i]
                        del L[i]
        except:
                print(f"i: {i} len(L): {len(L)}")

        return sum

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

Debugging:
Using debugger

```
import pdb
```

```
pdb.set_trace()
```

```
> <ipython-input-14-110393975fb5>(7)startswith()
-> for i in range(len(srcstr)): # check all characters of srcstr
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF     c          d         h         list      q        rv       undisplay
a       cl         debug     help      ll        quit     s        unt
alias   clear      disable   ignore    longlist  r        source   until
args    commands   display   interact  n         restart  step     up
b       condition  down      j         next      return   tbreak   w
break   cont       enable    jump      p         retval   u        whatis
bt      continue   exit      l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb
```

Previously on CENG240!

METU Computer Engineering

Debugging:
Using debugger

```
import pdb
```

```
pdb.set_trace()
```

```
import pdb

def sum_and_delete(L):
        sum = 0
        pdb.set_trace()
        for i in range(len(L)):
                sum += L[i]
                del L[i]

        return sum

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

This Week

- Scientific and Engineering Libraries
  - NumPy for numerical computing
  - SciPy for scientific computing
- Next week:
  - Pandas for data handling and analysis
  - Matplotlib for plotting

Administrative Notes

- Lab 8
- ~~Midterm: 1 June, Tuesday, 17:40~~
- Final: 8 July, 9:30


NumPy

Outline

- Overview
- Installation
- Arrays and their properties
- Working with arrays
- Linear algebra
- Why use NumPy?

Overview

- A Python library for arrays & matrices and mathematical functions that work on them.
- Combination of two ancestor libraries in 2005: Numeric and Numarray.

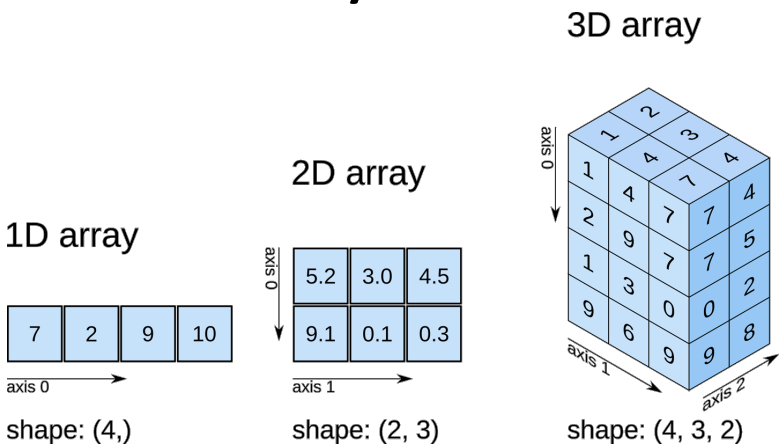


Figure: https://www.oreilly.com/library/view/elegant-scipy/9781491922927/ch01.html

Installation

https://numpy.org/install/

- On your Linux environment:

  $ pip install numpy

  or

  $ conda install numpy

- On Windows/Mac: install anaconda first
- On Colab, it is already installed

Data types in NumPy

- Integers:
  - np.int8, np.int16, np.int32, np.int64, np.uint8, ...
- Float:
  - np.float16, np.float32, np.float64, ...
- Complex:
  - np.complex64, np.complex128, ..
- Boolean:
  - np.bool8
- Default:
  - np.float64

For a full list, see:
https://numpy.org/devdocs/user/basics.types.html

Arrays and their properties

- In essence, NumPy is a library for n-dimensional arrays:

$$array1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

$$array2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

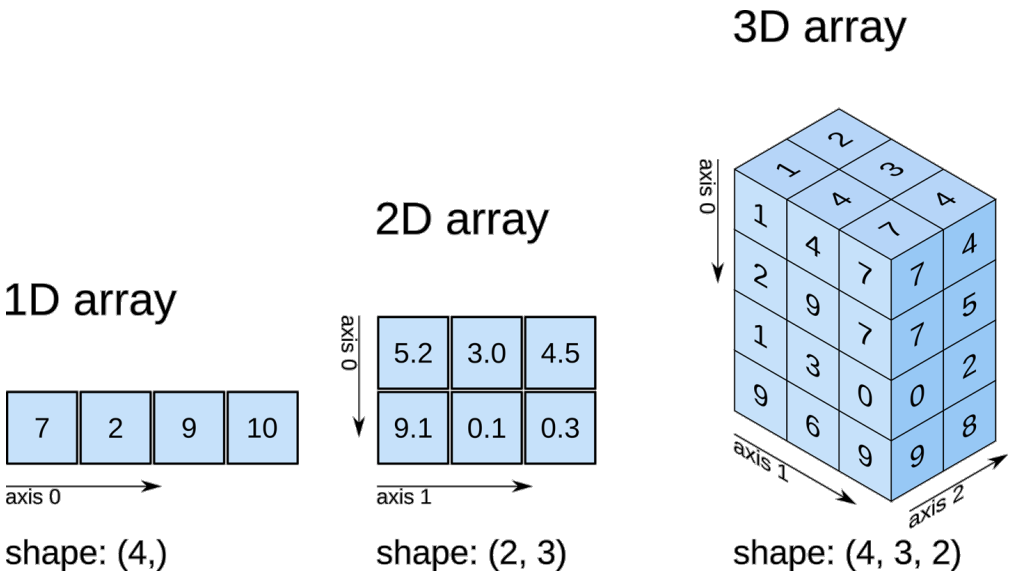


Figure: https://www.oreilly.com/library/view/elegant-scipy/9781491922927/ch01.html

Arrays and their properties

Basic array creation

$$array1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

$$array2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
>>> import numpy as np          # Import the NumPy library
>>> array1 = np.array([1, 2, 3])
>>> array2 = np.array([[1, 2, 3], [4, 5, 6]])
>>> type(array1)
<class 'numpy.ndarray'>
>>> type(array2)
<class 'numpy.ndarray'>
>>> array1
array([1, 2, 3])
>>> array2
array([[1, 2, 3],
       [4, 5, 6]])
>>> print(array2)
[[1 2 3]
 [4 5 6]]
```

Arrays and their properties

Array Shapes, Dimensions, and Elements

```
>>> array1.shape
(3,)
>>> array2.shape
(2,3)
```

$$array1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

$$array2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
>>> array1.reshape((3,1))
array([[1],
       [2],
       [3]])
>>> array2.reshape((1,6))
array([[1, 2, 3, 4, 5, 6]])
```

```
>>> array1.size
3
>>> array2.size
6
```

```
>>> array1.ndim
1
>>> array2.ndim
2
```

Arrays and their properties

Array Shapes, Dimensions, and Elements

Same indexing mechanisms of Python:

```
>>> array1[-1]
3
>>> array2[1][2]
6
>>> array2[-1]
array([4, 5, 6])
```

$$array1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

$$array2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

`array2[1, 2]` is also possible.

Arrays and their properties

Creating arrays

- With the constructor:
  - `np.array(<List>)`

```
>>> array1 = np.array([1, 2, 3])
>>> array2 = np.array([[1, 2, 3], [4, 5, 6]])
```

- Arrays with constant values:
  - `np.zeros(<shape>)`
  - `np.ones(<shape>)`

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones((2,6))
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

Arrays and their properties

Creating arrays

- Array with a range of values:
  - `np.arange(start, stop, step)`

```
>>> np.arange(1,10)      # 1: starting value. 10: ending value (excluded)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(1,10,2)
array([1, 3, 5, 7, 9])
>>> np.arange(1,10).reshape((3,3))
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

- Random array:
  - `np.random.randn(<shape>)`

Working with arrays

- Arithmetic, relational and membership operations
- Mathematical functions
- Splitting and combining arrays
- Iterations with arrays

Working with arrays

Arithmetic, relational and membership operations

- Arithmetic operations:
  - +, -, *, /, **

Be careful about the shapes of the arrays

```
>>> A = np.arange(4).reshape((2,2))
>>> A
array([[0, 1],
       [2, 3]])
>>> B = np.arange(4, 8).reshape((2,2))
>>> B
array([[4, 5],
       [6, 7]])
>>> print(B-A)
[[4 4]
 [4 4]]
>>> print(B+A)
[[ 4  6]
 [ 8 10]]
>>> A
array([[0, 1],
       [2, 3]])
>>> B
array([[4, 5],
       [6, 7]])
```

Working with arrays

Arithmetic, relational and membership operations

- Relational operations:
  - `==, <, <=, >, >=`
- Membership operations:
  - `in, not in`

Be careful about the shapes of the arrays

```
>>> A = np.arange(4).reshape((2,2))
>>> A
array([[0, 1],
       [2, 3]])
>>> B = np.arange(4, 8).reshape((2,2))
>>> B
array([[4, 5],
       [6, 7]])
>>> A < B
array([[ True,  True],
       [ True,  True]])
>>> B > A
array([[ True,  True],
       [ True,  True]])
>>> A > B
array([[False, False],
       [False, False]])
>>> 4 in B
True
>>> 10 in B
False
```

Working with arrays

Useful Functions

- `np.sqrt(<array>)`
- `np.exp(<array>)`
- `np.sin(<array>)`
- `np.cos(<array>)`
- `<array>.min()`
- `<array>.max()`
- `<array>.sum()`
- `<array>.mean()`
- `<array>.std()`

Working with arrays

Useful Functions

You may apply some of these operations on the whole array or along an axis

```
>>> A
array([[0, 1],
       [2, 3]])
>>> A.sum()
6
>>> A.sum(axis=0)
array([2, 4])
>>> A.sum(axis=1)
array([1, 5])
>>> A.sum(axis=2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.7/site-packages/numpy/core/_methods.py", line 47, in _sum
    return umr_sum(a, axis, dtype, out, keepdims, initial, where)
numpy.AxisError: axis 2 is out of bounds for array of dimension 2
```

Working with arrays

Splitting and Combining Arrays

Splitting:

- horizontal split (`np.hsplit`)
- vertical split (`np.vsplit`)
- general splitting (`np.array_split`)

```
>>> L = np.arange(16).reshape(4,4)
>>> L
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> np.hsplit(L,2)  # Divide L into 2 arrays along the horizontal axis
[array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]]), array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])]
>>> np.vsplit(L,2)
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])]
```

Working with arrays

Splitting and Combining Arrays

Combining:

- horizontal stack (`np.hstack`)
- vertical stack (`np.vstack`)
- general stack (`np.stack`)

```
>>> A
array([[0, 1],
       [2, 3]])
>>> B
array([[4, 5],
       [6, 7]])
>>> np.hstack((A, B))
array([[0, 1, 4, 5],
       [2, 3, 6, 7]])
>>> np.vstack((A, B))
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

Working with arrays

Iterations with Arrays

- You can use NumPy arrays in iterations as we would be using other container data types

```
>>> L
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> for r in L:
...     print("row: ", r)
...
row:  [0 1 2 3]
row:  [4 5 6 7]
row:  [ 8  9 10 11]
row:  [12 13 14 15]
```

Working with arrays

Iterations with Arrays

```
>>> for element in L.flat:
...     print(element)
...
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
>>> for r in L:
...     for element in r:
...         print(element)
...
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Linear algebra with NumPy

Transpose

- `<array>.T` or `<array>.transpose()`

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$A^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

```
>>> A
array([[1, 2],
       [3, 4]])
>>> A.T
array([[1, 3],
       [2, 4]])
>>> A
array([[1, 2],
       [3, 4]])
```

Original array does not change!

Linear algebra with NumPy

Inverse

- `np.linalg.inv(<array>)`

$$A \times A^{-1} = I.$$

```
>>> A
array([[1, 2],
       [3, 4]])
>>> A_inv = np.linalg.inv(A)
>>> A_inv
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Let's check correctness:

```
>>> np.matmul(A, A_inv)
array([[1.0000000e+00, 0.0000000e+00],
       [8.8817842e-16, 1.0000000e+00]])
```

Linear algebra with NumPy

Determinant, norm, rank, condition number, trace

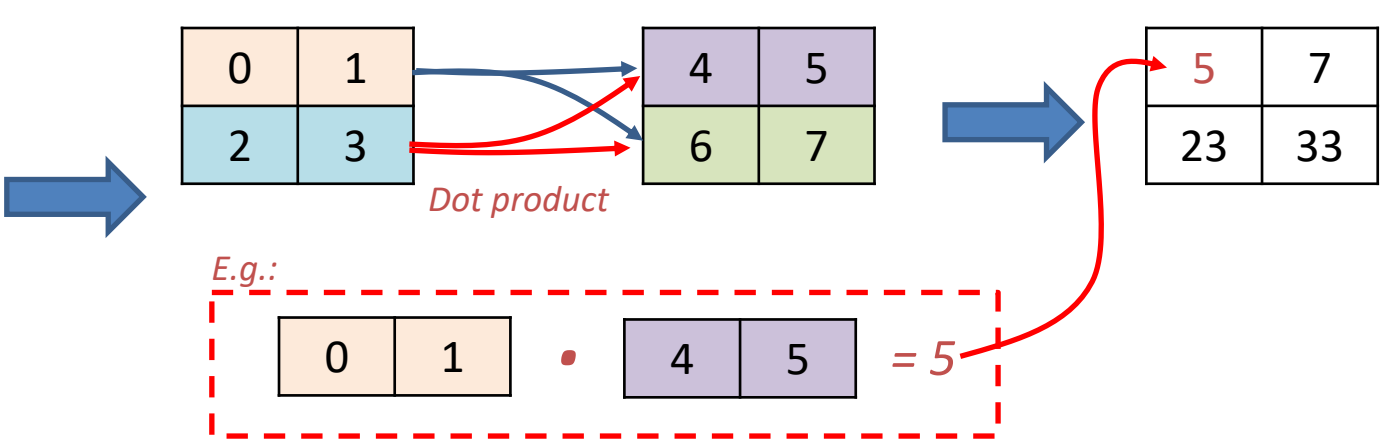
Matrix Property	How to Calculate with NumPy
Determinant ($\|A\|$ or $det(A)$)	`np.linalg.det(A)`
Norm ($\|\|A\|\|$)	`np.linalg.norm(A)`
Rank ($rank(A)$)	`np.linalg.matrix_rank(A)`
Condition number ($\kappa(A)$)	`np.linalg.cond(A)`
Trace ($tr(A)$)	`np.linalg.trace(A)`

Linear algebra with NumPy

Dot product, inner product, outer product, matrix multiplication

- `np.dot(a, b)`
  - For 1D arrays, this is dot product: $\sum_i \mathbf{a}_i \mathbf{b}_i$
  - For nD arrays, this is matrix multiplication (see below).
- `np.inner(a,b)`
  - For 1D arrays, this is dot product: $\sum_i \mathbf{a}_i \mathbf{b}_i$
  - For nD arrays, dot-product over the last axes:

```
>>> A
array([[0, 1],
       [2, 3]])
>>> B
array([[4, 5],
       [6, 7]])
>>> np.inner(A,B)
array([[ 5,  7],
       [23, 33]])
```



Linear algebra with NumPy

Dot product, inner product, outer product, matrix multiplication

- `np.matmul(a,b)`
  - $result_{ij} = \sum_k \mathbf{a}_{ik}\, \mathbf{b}_{kj}$

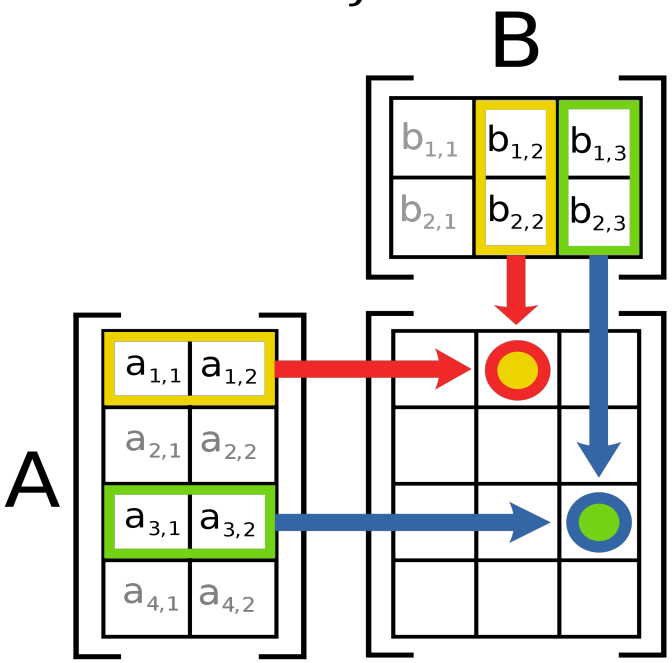


Figure source: Wikipedia

Linear algebra with NumPy

Dot product, inner product, outer product, matrix multiplication

- np.outer(a,b):
  - Defined over vectors (1D arrays)
  - equivalent to matrix multiplication with ab^{T}
  - $result_{ij} = a_i b_j$

$$\begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{bmatrix} [b_1\ b_2\ \dots b_m] = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \dots & a_1 b_m \\ a_2 b_1 & a_2 b_2 & \dots & a_2 b_m \\ \dots & \dots & \dots & \dots \\ a_n b_1 & a_n b_2 & \dots & a_n b_m \end{bmatrix}$$

Linear algebra with NumPy

Eigenvectors and Eigenvalues

$$Ax = \lambda x$$

matrix eigenvector eigenvalue

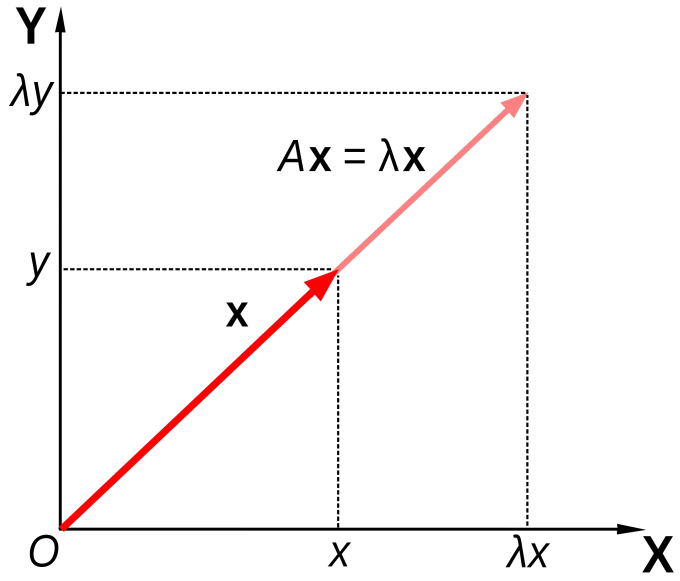


Figure source:
https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors

Linear algebra with NumPy

Matrix Decomposition

- Matrix decomposition is highly used in solving problems in linear algebra

Matrix Decomposition	How to Calculate with NumPy
Cholesky decomposition	`linalg.cholesky(A)`
QR factorization	`np.linalg.qr(A)`
Singular Value Decomposition	`linalg.svd(a)`

Linear algebra with NumPy

Solving a linear system of equations

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\
&\ldots \\
a_{n1}x_1 + a_{22}x_2 + \ldots + a_{nn}x_n &= b_n,
\end{aligned}
$$

which can be rewritten in matrix form as:

$$\mathbf{ax} = \mathbf{b},$$

We can use `np.linalg.solve(a,b)`, e.g.:

```
>>> A
array([[0, 1],
       [2, 3]])
>>> B
array([3, 4])
>>> np.linalg.solve(A,B)
array([-2.5,  3. ])
>>> X = np.linalg.solve(A,B)
>>> X
array([-2.5,  3. ])
```

We can verify the solution by checking whether AX=B:

```
>>> np.inner(A, X)
array([3., 4.])
```

Why Use NumPy?

- We can implement all functionalities of NumPy using Python datatypes and functions.
- E.g. matrix multiplication is just an iteration over the elements of two arrays.
- However, all such NumPy operations are implemented in C. Therefore, they are faster.

~1000 times difference in speed!

```python
import numpy as np
from time import time

def matmul_2D(M, N):
  """Custom defined matrix multiplication for two 2D matrices M and N"""
  (H_M, W_M) = M.shape
  (H_N, W_N) = N.shape
  if W_M != H_N:
    print("Dimensions of M and N mismatch!")
    return None

  result = np.zeros((H_M, W_N))
  for i in range(H_M):
    for j in range(W_N):
      for k in range(W_M):
        result[i][j] += M[i][k] * N[k][j]

  return result
```

```python
# Now let us measure the running-time performances
# Create two 2D large matrices
M = np.random.randn(100, 100)
N = np.random.randn(100, 100)

# Option 1: Use NumPy's matrix multiplication
t1 = time()
result = np.matmul(M, N)
t2 = time()
print("NumPy's matmul took ", t2-t1, "ms.")

# Option 2: Use our matmul_2D function
t1 = time()
result = matmul_2D(M, N)
t2 = time()
print("Our matmul_2D function took ", t2-t1, "ms.")
```

```
Our matmul_2D result:
 [[1.44819064 2.44425364 3.19640633 0.98131108]
 [1.91135161 2.66745824 4.09341735 1.37928184]]
Correct result:
 [[1.44819064 2.44425364 3.19640633 0.98131108]
 [1.91135161 2.66745824 4.09341735 1.37928184]]
NumPy's matmul took  0.008047342300415039 ms.
Our matmul_2D function took  1.2613885402679443 ms.
```


SciPy

Outline

- Overview
- Installation
- Modules

Overview

- A library for Scientific Computing.
  - Clustering, Fourier Transforms, Integration & differential equation solving, Linear Algebra, Optimization, Image processing ...
- It is closely linked with NumPy so much that NumPy needs to be imported first to be able to use SciPy.

Installation

- On your Linux environment:

  $ pip install scipy

  or

  $ conda install scipy

- On Windows/Mac: install anaconda first
- On Colab, it is already installed

Modules

SciPy

Module	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

Examples

- From the book:
  - “Define functions that work like the sum, mean, min and max operations provided by NumPy. These functions should take a single 2D array and return the result as a number. You can assume that the operation applies to the whole array and not to a single axis.”

Final Words: Important Concepts

- NumPy arrays and their properties: array shape, dimensions, sizes, elements.
- Accessing and modifying elements of a NumPy array.
- Simple algebraic functions on NumPy arrays.
- SciPy and its basic capabilities.

THAT'S ALL FOLKS!
STAY HEALTHY