ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ Η/Υ
σε Python

Διάλεξη 10: Η βιβλιοθήκη NumPy

Μάρκος Ζάμπογλου, mzampoglou@aegean.gr

Βιβλιοθήκες

- Μια βιβλιοθήκη της Python είναι ένα σύνολο *δομοστοιχείων* (**modules**) τα οποία περιέχουν **συναρτήσεις** και **κλάσεις** (με τις **μεθόδους** τους).
  - Πρόκειται δηλαδή για επιπλέον κώδικα που εμπλουτίζει την εφαρμογή μου με νέους τύπους δεδομένων και νέες δυνατότητες
- Κάποιες βιβλιοθήκες έρχονται προεγκατεστημένες με την Python
  - Π.χ. **`random`**, **`math`**, **`copy`**
- Τις περισσότερες πρέπει να τις κατεβάσω και να της εγκαταστήσω για να μπορώ να τις χρησιμοποιήσω

Βρίσκοντας βιβλιοθήκες

- Αν δουλεύω σε Google Colab, οι περισσότερες βιβλιοθήκες που θα χρειαστώ είναι **ήδη εγκατεστημένες στο σύστημα**
  - Μένει μόνο να τις κάνω import στον κώδικά μου
- Αν δουλεύω τοπικά όμως, θα χρειαστεί να **κατεβάσω** και να **εγκαταστήσω** τις βιβλιοθήκες πριν τις **εισαγάγω** στον κώδικά μου

Το σωστό είναι να εγκαθιστώ βιβλιοθήκες μέσα σε **εικονικά περιβάλλοντα (virtual environments)**

Εικονικά Περιβάλλοντα της Python

- Ένα **εικονικό περιβάλλον (virtual environment)** στην Python είναι ένα αυτοτελές τμήμα του δέντρου φακέλων
  - Έχει *ξεχωριστή* έκδοση της Python εγκατεστημένη
  - Έχει *δικές του* εγκατεστημένες βιβλιοθήκες
- Ο λόγος που χρησιμοποιούμε εικονικά περιβάλλοντα είναι ότι, αν εγκαταστήσουμε πολλές βιβλιοθήκες ταυτόχρονα, ενδέχεται να υπάρξουν **συγκρούσεις** μεταξύ τους (**conflicts**)
  - Είναι πιο ασφαλές **κάθε εφαρμογή να αναπτύσσεται σε διαφορετικό εικονικό περιβάλλον**, το οποίο θα περιέχει μόνο τις απαραίτητες βιβλιοθήκες

Δημιουργώντας εικονικά περιβάλλοντα στο Anaconda Powershell

- Για να δημιουργήσουμε environment:

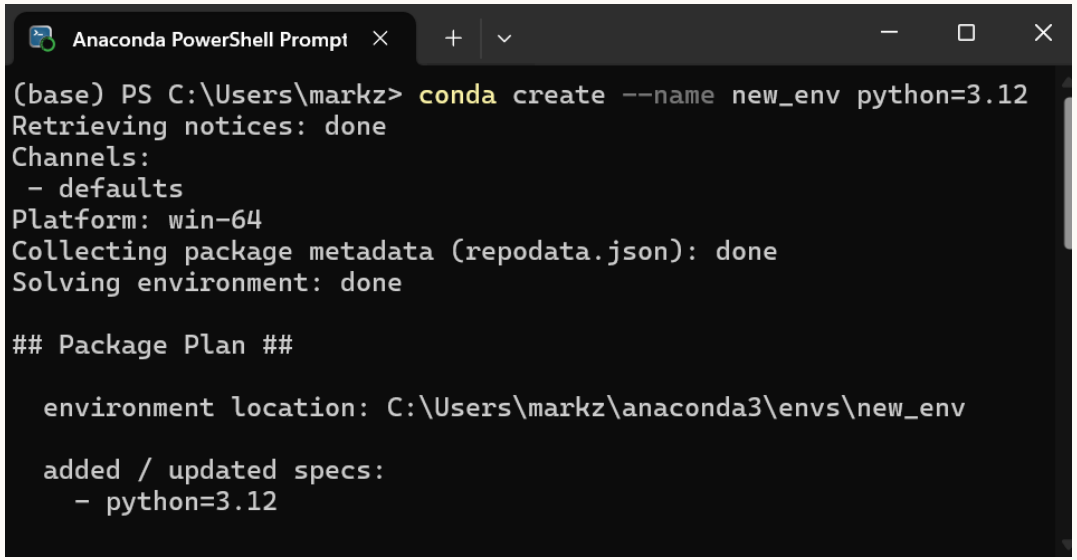
conda create --name <my-env> python=<version>

- Για να μεταβούμε σε αυτό:

conda activate <my-env>

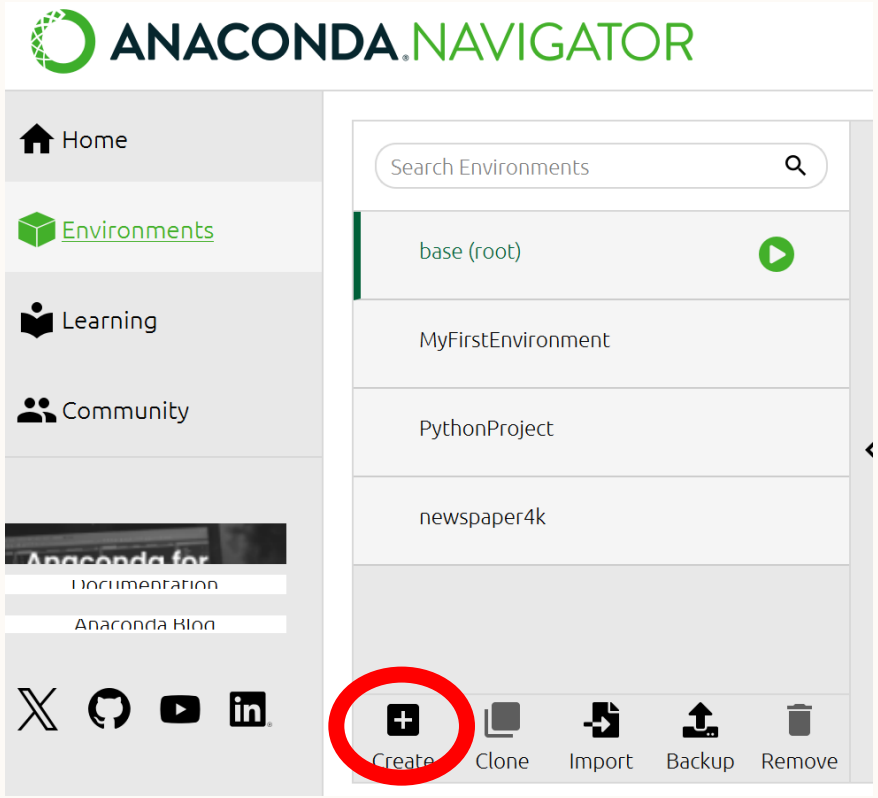
- Για να εγκαταστήσουμε βιβλιοθήκες:

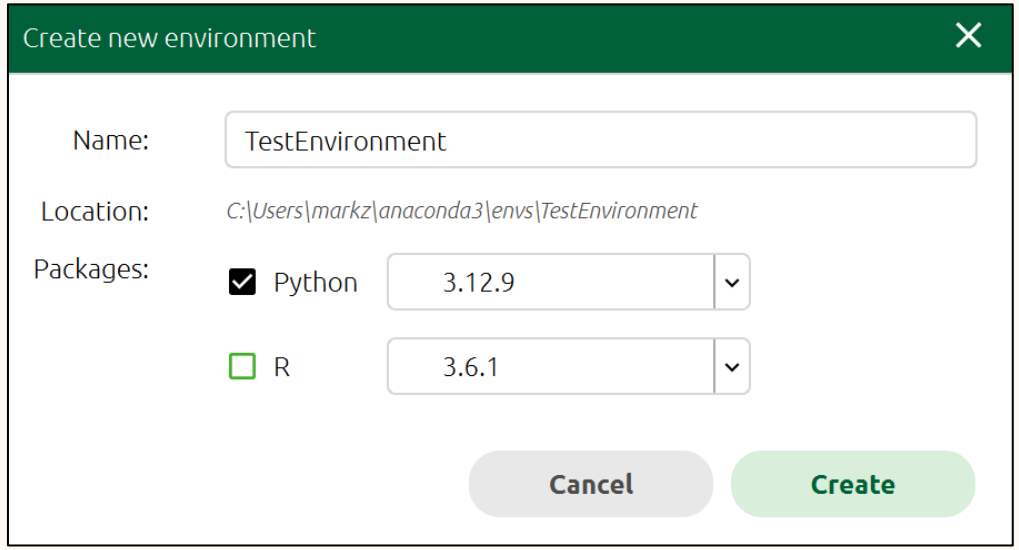
conda install <library-name>



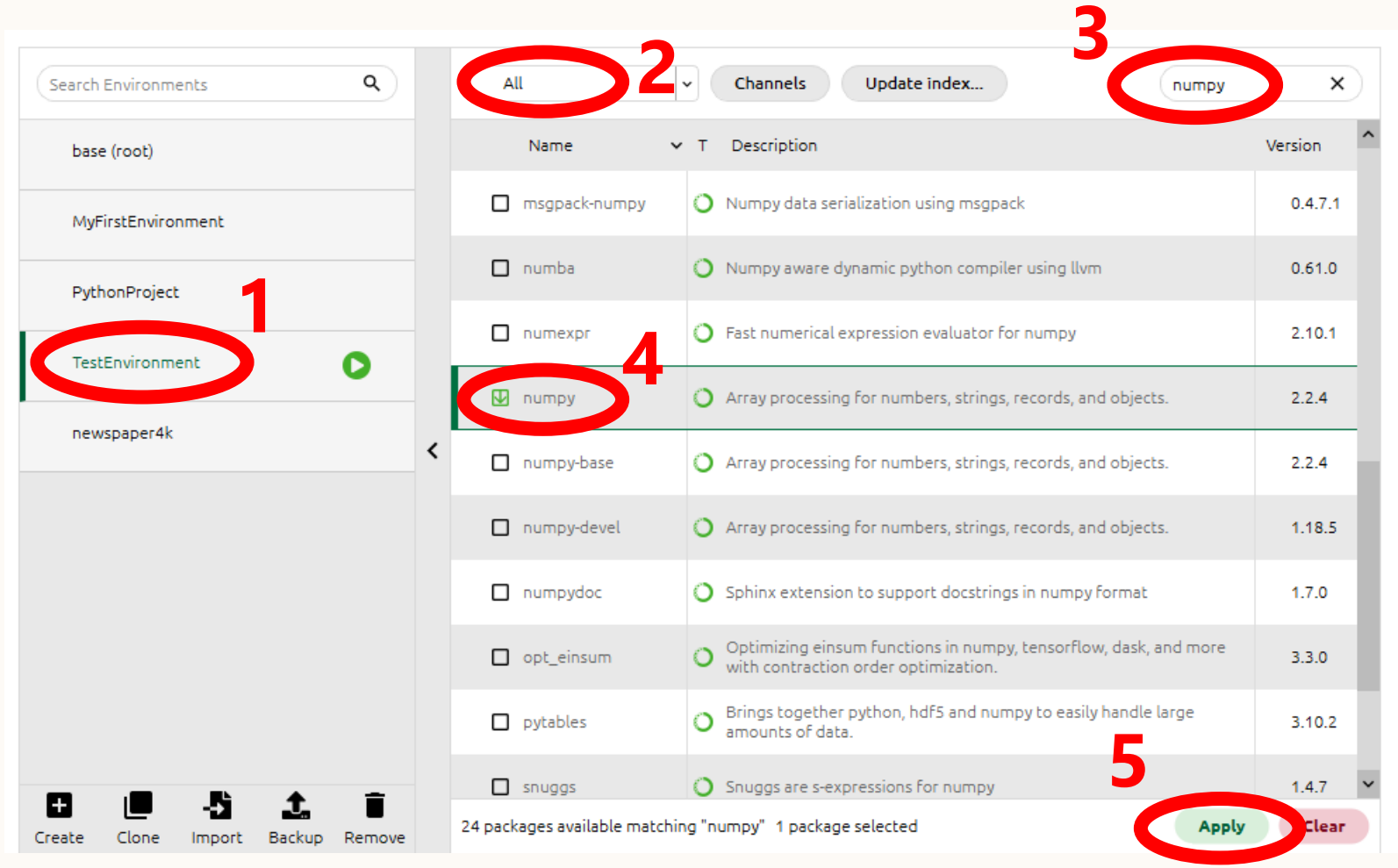
https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

Εικονικά περιβάλλοντα στο Anaconda GUI





Εγκαθιστώντας βιβλιοθήκες στο Anaconda GUI



Τρέχοντας Jupyter Notebook σε συγκεκριμένο environment

- Για να μπορεί να "δει" το Jupyter Notebook τις βιβλιοθήκες, πρέπει να εκτελεστεί μέσα από περιβάλλον που τις περιέχει
- Αντί να το εκτελέσουμε από το μενού των Windows, το εκτελούμε μέσα από το Anaconda Powershell
  - Μπαίνουμε μέσα σε ένα environment που έχει εγκατεστημένο το Jupyter και τις βιβλιοθήκες που θέλουμε, και τρέχουμε **`jupyter notebook`**

```
(my_environment) PS C:\Users\markz> jupyter notebook
```

Βιβλιοθήκες

Το να εισάγω μια βιβλιοθήκη στην Python είναι απλό:

```
import math #εισάγει τη βιβλιοθήκη random
a = math.sin(3.141) #εκτελεί τη συνάρτηση sin της math
import numpy #εισάγει τη βιβλιοθήκη numpy
a = numpy.zeros(2) #εκτελεί τη συνάρτηση zeros της numpy
```

Ή, αν προτιμάω (*συνήθως το προτιμάω*), με **ψευδώνυμο** (***alias***):

```
import numpy as np
a = np.zeros(2)
```

Η βιβλιοθήκη NumPy

- Η NumPy είναι μια από τις πιο διαδεδομένες βιβλιοθήκες της Python
- Η βασική δομή δεδομένων της, ο **πίνακας NumPy** (**NumPy array**) χρησιμοποιείται σε πάρα πολλές άλλες βιβλιοθήκες, και είναι πολύ σημαντικός στο οικοσύστημα της Python

NumPy

- Το όνομά της προέρχεται από το Numerical Python
- Το βασικό της πλεονέκτημα είναι ότι οι πίνακες NumPy επιτρέπουν πολύ γρηγορότερη επεξεργασία από τις λίστες
  - Και διαθέτουν ένα μεγάλο πλήθος ενσωματωμένων μεθόδων
- Είναι εξαιρετικά διαδεδομένη και αποτελεί τη βάση σχεδόν **όλων** των άλλων επιστημονικών βιβλιοθηκών της Python

Πίνακες NumPy

Ένας πίνακας NumPy είναι ένας πίνακας N διαστάσεων, όπως αυτοί της γραμμικής άλγεβρας

Διαφορές πίνακα NumPy και λίστας:

- Όλα τα στοιχεία του πίνακα πρέπει να είναι ίδιου τύπου
  - Η NumPy έχει τους δικούς της τύπους δεδομένων που λέγονται **dtypes**
    https://www.w3schools.com/python/numpy/numpy_data_types.asp
    - Πχ float32, int64, bool
- Οι πίνακες μπορεί να είναι πολυδιάστατοι (οι λίστες όχι)
  - Το κοντινότερο πράγμα που έχουμε είναι οι λίστες μέσα σε λίστες, όμως δεν είναι το ίδιο
- Οι πίνακες δεν έχουν δυναμικό μέγεθος.
  - Η μέθοδοι που τροποποιούν το μέγεθος ενός πίνακα, **επιστρέφουν έναν νέο πίνακα** (*δεν εφαρμόζονται in-place*)

Δημιουργία NumPy array 1/2

Μπορώ να δημιουργήσω έναν πίνακα Numpy από μια λίστα Python με τη συνάρτηση **`np.array()`**

- Μία λίστα από στοιχεία επιστρέφει μονοδιάστατο/διάνυσμα

```
arr = np.array([4, 3, 7])
```

- Μια λίστα από εσωτερικές λίστες επιστρέφει δισδιάστατο

```
matrix = np.array([[1, 2], [3, 4]])
```

- κλπ...

Προσοχή! Στη δεύτερη περίπτωση όλες οι εσωτερικές λίστες πρέπει να έχουν όλες ίδιο μήκος!

Δημιουργία NumPy array 2/2

Από συναρτήσεις της NumPy:

`z = np.zeros((2, 3, 2))`: 2×3×2 πινακας μηδενικών (0)

```
[[[0. 0.]
  [0. 0.]
  [0. 0.]]

 [[0. 0.]
  [0. 0.]
  [0. 0.]]]
```

Η παράμετρος μπορεί να είναι είτε μεμονωμένος αριθμός (π.χ. **`3`**) είτε tuple με όλες τις διαστάσεις (π.χ. **`(2,3,2)`**)

`o = np.ones(3)`: διάνυσμα 3 τιμών με μονάδες (1)

```
[1. 1. 1.]
```

`id = np.eye(3)`: 3×3 μοναδιαίος ($\mathcal{I}$)

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Ο **`eye`** είναι πάντα δισδιάστατος. Αν δώσουμε μια παράμετρο βγαίνει τετραγωνικός, αν δώσουμε δύο βγαίνει στις διαστάσεις που θα ορίσουμε

Άλλες μέθοδοι δημιουργίας πινάκων

- Συναρτήσεις που παράγουν ακολουθίες:
  - **`np.arange(start=0, stop, step=1)`**: λειτουργεί όπως και η **`range()`**, αλλά επιστρέφει πίνακα NumPy
  - **`np.linspace(start, stop, num=50)`**: παράγει **`num`** αριθμούς σε ίσες αποστάσεις από start έως stop
- Συναρτήσεις τυχαίων αριθμών (παράδειγμα):

```
#Πρώτα δημιουργώ μια γεννήτρια τυχαίων αριθμών από τη numpy.random
#Αν δώσω κάποιον ακέραιο ως παράμετρο, αρχικοποιώ τη γεννήτρια
# σε συγκεκριμένη κατάσταση
rng = np.random.default_rng(1981)
#Πίνακας 2×3 πραγματικών αριθμών από ομοιόμορφη κατανομή στο [0.0, 1.0)
randfl = rng.random((2,3))
#Πίνακας 2×4 ακέραιων αριθμών από ομοιόμορφη κατανομή στο [2, 4)
randint = rng.integers(10,15,(2,4))
```

Στο παρελθόν, χρησιμοποιούσαμε τις συναρτήσεις **`numpy.random.rand()`** και **`numpy.random.randint()`**. Σε επόμενες εκδόσεις της Python αυτές οι συναρτήσεις θα καταργηθούν, όμως ακόμα χρησιμοποιούνται συχνά.

Ιδιότητες της κλάσης `numpy.array`

```
my_arr = np.zeros(5,5,2)
```

Διαστάσεις πίνακα:

```
print(my_arr.shape)
(5,5,2)
```

Πλήθος στοιχείων:

```
print(my_arr.size)
50
```

Τύπος δεδομένων:

```
print(my_arr.dtype)
float64
```

Διαστάσεις

Οι διαστάσεις καταγράφονται στην ιδιότητα **`.shape`**

1D

0	1	2	3	4

shape: (5,)

2D

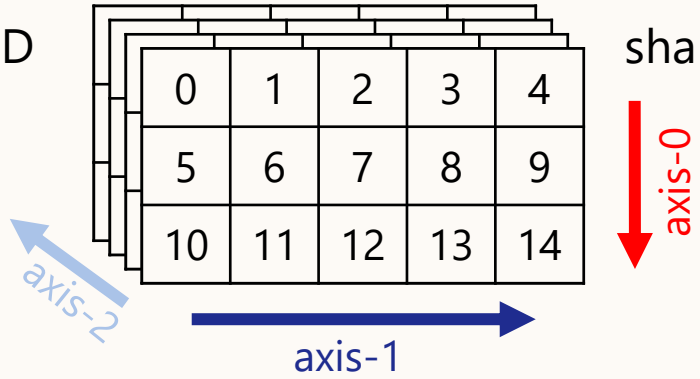
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

shape: (3,5)

shape: (3,1)

4
9
14

3D

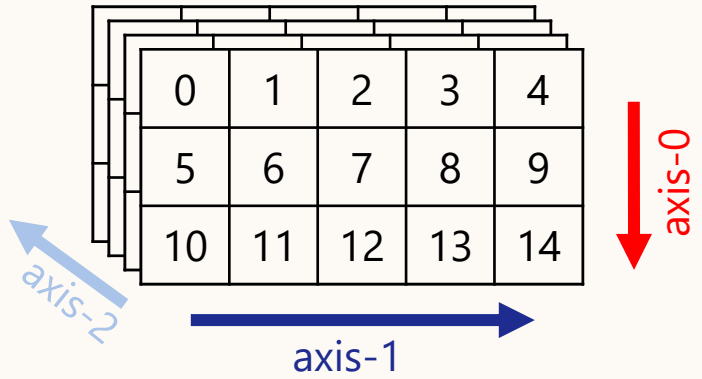


shape: (3,5,4)

Προσοχή στη διαφορά ανάμεσα στο μονοδιάστατο και τον δισδιάστατο με μία στήλη!

Προσοχή στην απεικόνιση!

Μαθηματικά αλλά και σχεδιαστικά, ένας τρισδιάστατος πίνακας 3×5×4 απεικονίζεται ως 4 πίνακες 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]]

 [[40 41 42 43]
  [44 45 46 47]
  [48 49 50 51]
  [52 53 54 55]
  [56 57 58 59]]]
```

Η numpy όμως, έναν τέτοιο πίνακα τον αντιμετωπίζει ως 3 πίνακες 5×4

Χρειάζεται προσοχή για να αποφύγουμε σφάλματα!

Αλλαγή διαστάσεων πίνακα

- Μπορώ να αλλάξω τις διαστάσεις ενός πίνακα με τη μέθοδο **.reshape()**

```
a=np.arange(24)
print(a)
```

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

```
a2=a.reshape((3,2,4))
print(a2)
```

```
[[[0  1  2  3]
  [4  5  6  7]]

 [[8  9 10 11]
  [12 13 14 15]]

 [[16 17 18 19]
  [20 21 22 23]]]
```

- Αν το πλήθος των στοιχείων του νέου πίνακα διαφέρει από του παλιού, προκαλείται σφάλμα

Indexing και slicing

Έστω ο δισδιάστατος πίνακας:

```
matrix = np.array([[1, 2, 3, 4, 5],
                   [6, 7, 8, 9, 10],
                   [11, 12, 13, 14, 15]])
```

Το indexing γίνεται όπως οι λίστες, όμως με όλες τις διαστάσεις μέσα στην ίδια αγκύλη:

```
print(matrix[0,3])
```

```
4
```

Το slicing επίσης, με τη γνωστή δομή **από:μέχρι:βήμα**:

```
print(matrix[:, 1:3])
```

```
[[ 2  3]
 [ 7  8]
 [12 13]]
```

```
print(matrix[1, ::-1])
```

```
[10 9 8 7 6]]
```

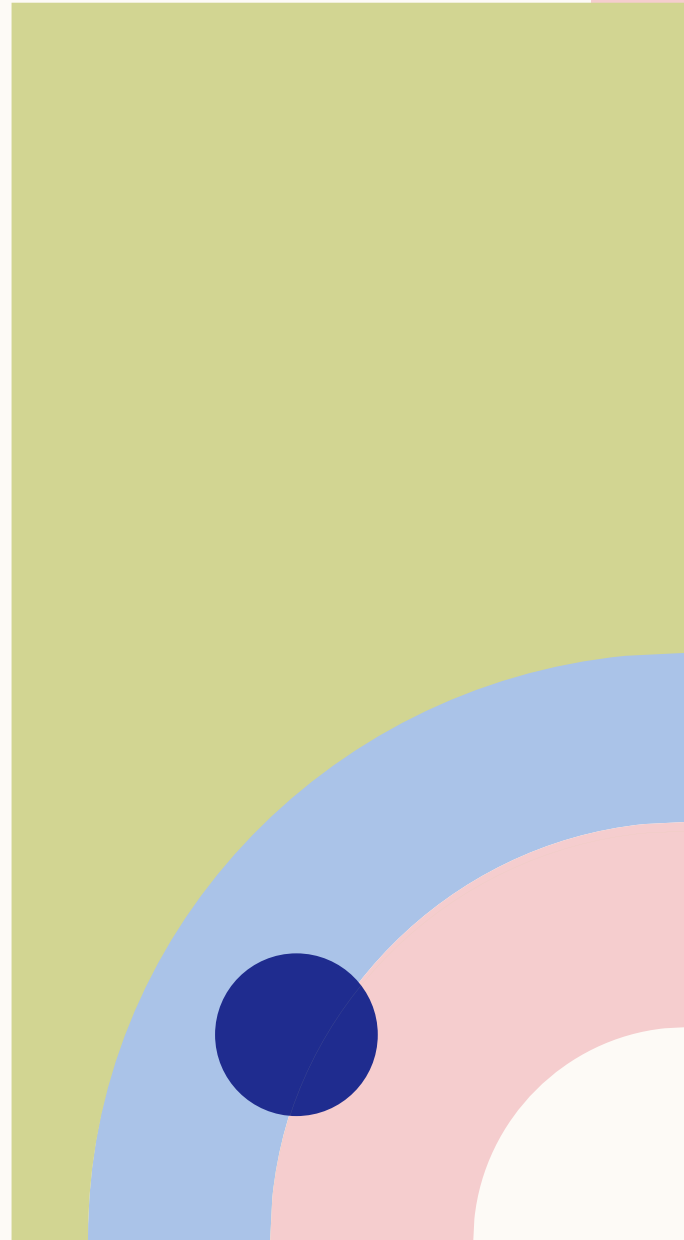
```
print(matrix[1, 0:-2:2])
```

```
[6  8]
```

Παράδειγμα 10.1

Να δημιουργηθεί ένας πίνακας NumPy με όνομα **`arr`** διαστάσεων 4×6 με τυχαίους ακέραιους από 11 έως 20

Στη συνέχεια, να δημιουργηθεί ένας νέος πίνακας 2×3 που θα περιέχει τα στοιχεία της 3^{ης} γραμμής του **`arr`**.



Παράδειγμα 10.1

Να δημιουργηθεί ένας πίνακας NumPy με όνομα **arr** διαστάσεων 4×6 με τυχαίους ακέραιους από 11 έως 20

Στη συνέχεια, να δημιουργηθεί ένας νέος πίνακας 2×3 που θα περιέχει τα στοιχεία της 3ης γραμμής του **arr**.

```
rng = np.random.default_rng(42)
arr = rng.integers(11,21,(4,6))
print(arr)
arr2 = arr[2,:].reshape(2,3)
print(arr2)
```

Boolean indexing

Αντί για αριθμητικές τιμές συντεταγμένων, μπορούμε να χρησιμοποιήσουμε πίνακες ή λίστες από boolean για indexing:

```
a = np.array([8, 7, 6, 5, 4, 3])
b = [True, False, True, True, False, False]
print(a[b])
```

```
[8 6 5]
```

Αυτή η λειτουργία μας παρέχει ένα πολύ δυνατό εργαλείο. Θα δούμε στη συνέχεια γιατί.

Indexing με λίστες

Αν δε θέλω να ορίσω κάποιο range με αρχή τέλος και βήμα αλλά θέλω να ορίσω αυθαίρετες τιμές, μπορώ να τις βάλω σε λίστα

Π.χ. μόνο η τέταρτη και δέκατη έβδομη γραμμή, όλες οι στήλες:

```
print(a[[3,16],:])
```

...σε περισσότερες διαστάσεις, τα πράγματα γίνονται πιο περίπλοκα:
https://numpy.org/doc/stable/user/basics.indexing.html 🤓

Views και Copies

- Οι περισσότερες μέθοδοι της NumPy που επιστρέφουν πίνακα, επιστρέφουν ένα **View** αυτού του πίνακα
  - Το ίδιο και το slicing
- Ένα *view* ενός πίνακα αποτελεί ένα υποσύνολο των στοιχείων του σε μια νέα διάταξη/οπτική, το οποίο όμως **συνδέεται με τον αρχικό πίνακα**
  - Με μια έννοια, έχει το ίδιο reference
  - ...που σημαίνει ότι **τροποποιώντας το view, τροποποιείται και ο πίνακας**
- Υπάρχουν συγκεκριμένες μέθοδοι ή περιπτώσεις, όπου επιστρέφεται ένα **αντίγραφο** (Copy) του πίνακα, με τιμές ανεξάρτητες από τις αρχικές
  - Και υπάρχει και η μέθοδος `.copy()` (βλ. κώδικα)

Πράξεις NumPy arrays

- Οι πράξεις πίνακα με αριθμό εκτελούνται σε κάθε στοιχείο ξεχωριστά
  - `np.array([1,2,3])+2` επιστρέφει `[3,4,5]`
  - `np.array([1,2,3])>2` επιστρέφει `[False False True]`
- Οι πράξεις μεταξύ πινάκων εκτελούνται στοιχείο προς στοιχείο
  - Επιτρέπονται όλες οι γνωστές πράξεις (*,+,%,>,== κλπ)
  - `np.array([3,4])+np.array([1,2])` επιστρέφει `[4 6]`
  - `np.array([3,2])==np.array([1,2])` επιστρέφει `[False True]`
  - Αρκεί οι διαστάσεις των πινάκων να είναι συμβατές (*όχι απαραίτητα ίδιες*)
    - *(...γιατί υπάρχει και το broadcasting)*

🤓 Συμβατές διαστάσεις (*broadcasting*)

- Πίνακας 1×3 με πίνακα 2×3:

```
a = np.ones((1,3))
b = np.array([[10,20,30],[40,50,60]])
print(a + b)
```

```
[[11. 21. 31.]
 [41. 51. 61.]]
```

Η NumPy:
α) προσθέτει επιπλέον διαστάσεις με τιμή 1 στο μικρότερο πίνακα όσες φορές χρειάζεται ώστε να "εφαρμόσει" η διάσταση που έχει ίση με το μεγαλύτερο, και
β) αντιγράφει τον πίνακα που προκύπτει ώστε να αποκτήσει ακριβώς ίσες διαστάσεις με το μεγαλύτερο.

- Πίνακας 4×3×2 με διάνυσμα 2 (το οποίο μετατρέπεται σε (1,1,2) και στη συνέχεια σε (4,3,2) :

```
a = np.ones((4, 3, 2))
b = np.array([1, 2])
print(a + b)
```

…πινακας 4×3×2 με τα αποτελέσματα

Το broadcasting είναι πολύ χρήσιμο όταν θέλουμε π.χ. να αφαιρέσουμε μια γραμμή από ένα πίνακα (π.χ. τη μέση τιμή κάθε στήλης από όλες τις γραμμές)

Ξανά για το Boolean indexing

- Το γεγονός πως οι συγκρίσεις πίνακα με αριθμό ή πίνακα επιστρέφουν πίνακες από boolean, συνδυάζεται πολύ βολικά με το boolean indexing:

```
a = np.array([1,-4,8,9,2)
print(a>2)
[False False True True False]
print(a[a>5])
[8 9]
```

- Η πράξη **a>5** επιστρέφει έναν πίνακα Boolean
  - Έχει ίσες διαστάσεις με τον **a**
  - Μπορώ να τον χρησιμοποιήσω για να επιλέξω συγκεκριμένα στοιχεία του **a**.
  - Η πρακτική αυτή λέγεται **Boolean Masking** και ο **a>5** είναι μια **Boolean Mask**

Παράδειγμα 10.2

Να γραφεί κώδικας που θα δημιουργεί δυο πίνακες **a** και **b** διαστάσεων **(4,5)** με τυχαίους ακεραίους από **1** έως **20** και θα τους πολλαπλασιάζει ανά στοιχείο. Στη συνέχεια θα εμφανίζει τον πίνακα των γινομένων, όμως στις θέσεις που ο **b** έχει άρτιο αριθμό να βάζει **-1**.

Παράδειγμα 10.2

Να γραφεί κώδικας που θα δημιουργεί δυο πίνακες **a** και **b** διαστάσεων **(4,5)** με τυχαίους ακεραίους από **1** έως **20** και θα τους πολλαπλασιάζει ανά στοιχείο. Στη συνέχεια θα εμφανίζει τον πίνακα των γινομένων, όμως στις θέσεις που ο **b** έχει άρτιο αριθμό να βάζει **-1**.

```
rng = np.random.default_rng()
a = rng.integers(1,21,(4,5))
b = rng.integers(1,21,(4,5))
c = a*b
c[b%2==0]=-1
print(c)
```

Συναρτήσεις NumPy
(και μέθοδοι πινάκων NumPy)

Αριθμητικές πράξεις:

Στρογγυλοποίηση (π.χ. στα 2 δεκαδικά):
```
print(np.round(a,2))
```
Απόλυτη τιμή όλων των στοιχείων:
```
print(np.abs(a))
```
Λογάριθμος:
```
print(np.log(a))
```
Ημίτονο:
```
print(np.sin(a))
```

🤓 Οι περισσότερες συναρτήσεις που θα δούμε υπάρχουν και ως μέθοδοι. Π.χ. αντί για `b=np.round(a,2)` μπορώ να γράψω `b=a.round(2)`. Οι διαφορές μεταξύ των δύο εκδοχών δε θα μας απασχολήσουν εδώ.

Υπάρχουν πράξεις που είτε εφαρμόζονται σε όλα τα στοιχεία του πίνακα, είτε λειτουργούν ως πράξεις μεταξύ πινάκων (ίσων διαστάσεων):

Δύναμη:
```
print(np.power(a,3)) ή print(np.power(a,a))
```
Modulo:
```
print(np.mod(a,3)) ή print(np.mod(a,a))
```
(Στην πραγματικότητα κάνουν broadcasting)

Συναρτήσεις NumPy
(και μέθοδοι πινάκων NumPy)

Συναρτήσεις συγκέντρωσης πληροφορίας:

- …"aggregate information"
- Δέχονται ως πρώτη παράμετρο έναν πίνακα numpy, και ως προαιρετική δεύτερη παράμετρο τη διάσταση επί της οποίας θα γίνει ο υπολογισμός
- Π.χ. μέγιστη τιμή όλου του πίνακα:

```
np.max(a)
```

- Π.χ. μέγιστη τιμή κάθε στήλης:

```
np.max(a,0)
```

- Αντίστοιχα `np.min, np.sum, np.mean, np.median, np.count_nonzero`

Αν ο a έχει διαστάσεις `(4,10)` τότε ο `np.max(a,0)` έχει διαστάσεις `(4,1)`, δηλαδή είναι μια γραμμή με ένα στοιχείο για κάθε στήλη.

Συναρτήσεις NumPy
(και μέθοδοι πινάκων NumPy)

Συναρτήσεις μετασχηματισμού πινάκων

- `.flatten()` (μέθοδος πίνακα): μετατροπή σε μονοδιάστατο
- `.ravel()`: το ίδιο, όμως επιστρέφει *view* αντί για *copy*
- `.reshape(d1,d2,d3,…)`: αλλαγή διαστάσεων
- `numpy.concatenate((a,b,c,…),d)`: συνένωση των πινάκων **a**,**b**,**c**,... κατά μήκος της διάστασης **d**
  - Παρόμοια λειτουργία έχουν και οι `np.stack()`, `np.hstack()`, `np.vstack()` που δε θα δούμε εδώ

Συναρτήσεις NumPy
(και μέθοδοι πινάκων NumPy)

Λογικές πράξεις και επιλογή

- `np.logical_and()`, `np.logical_or()`, `np.logical_not()`: Λογικές πράξεις ανά στοιχείο
- `np.nonzero(a)`: επιστρέφει τις συντεταγμένες όπου οι τιμές του **a** είναι μη-μηδενικές

Συναρτήσεις NumPy
(και μέθοδοι πινάκων NumPy)

- Όλα αυτά αγγίζουν μόνο την επιφάνεια…

Πλήρης λίστα: https://numpy.org/devdocs/reference/routines.math.html

Παράδειγμα 10.3

Να γραφεί κώδικας Python που θα δημιουργεί δυο πίνακες τυχαίων ακεραίων από 1 έως 100 διαστάσεων (20,10) και θα τους αθροίζει. Στη συνέχεια θα υπολογίζει:

Α) Πόσα στοιχεία έχουν τιμή πάνω από 100

Β) Ποιες γραμμές έχουν μέση τιμή πάνω από 100

Παράδειγμα 10.3

Να γραφεί κώδικας Python που θα δημιουργεί δυο πίνακες τυχαίων ακεραίων από 1 έως 100 διαστάσεων (20,10) και θα τους αθροίζει. Στη συνέχεια θα υπολογίζει:

Α) Πόσα στοιχεία έχουν τιμή πάνω από 100

Β) Ποιες γραμμές έχουν μέση τιμή πάνω από 100

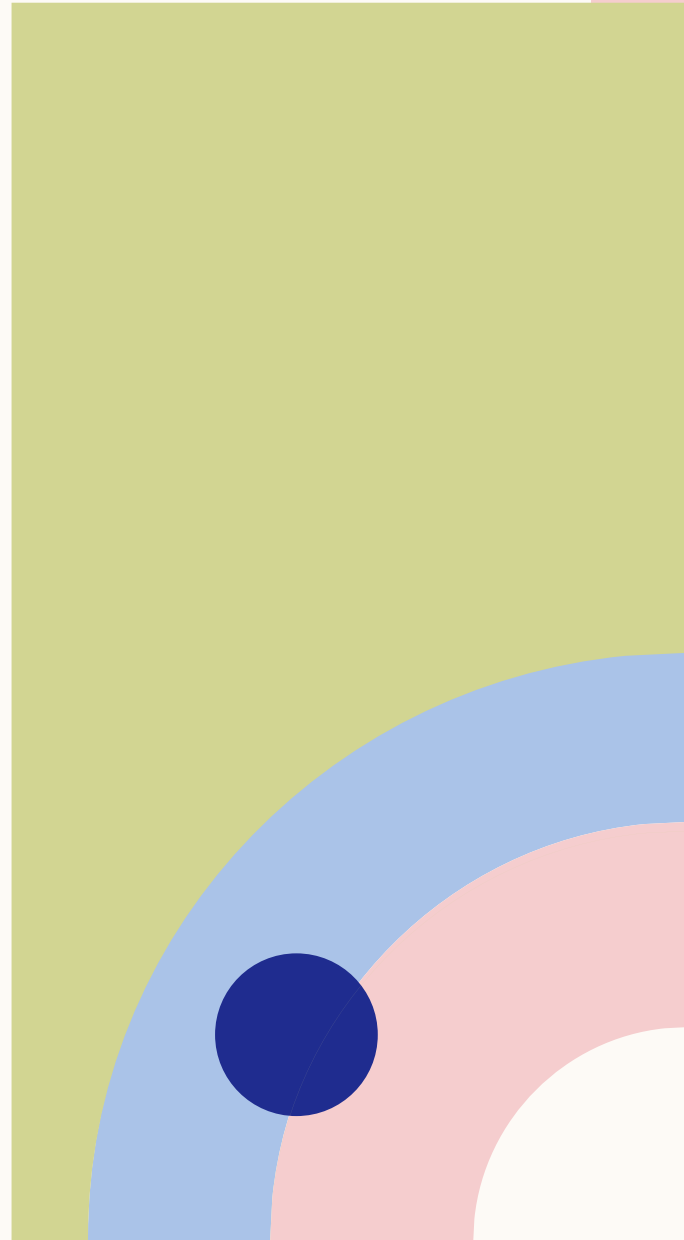
```
rng = np.random.default_rng()
a = rng.integers(1,101,(20,10))
b = rng.integers(1,101,(20,10))
c = a + b
print(np.sum(c > 100))
print(np.mean(c,1).shape)
print(np.where(np.mean(c,1) > 100))
```

Πράξεις γραμμικής άλγεβρας

Πολλαπλασιασμός πινάκων (matrix multiplication):
`np.matmul(a,b)`
Εναλλακτικά: **`a@b`**

Ανάστροφος πίνακας:
`np.transpose(a)`
Εναλλακτικά: **`a.T`**

Αντίστροφος πίνακας:
`np.linalg.inv(a)`



Παράδειγμα 10.4

Επιβεβαιώστε εμπειρικά ότι η συνάρτηση `np.linalg.inv()` λειτουργεί σωστά

Παράδειγμα 10.4

Επιβεβαιώστε εμπειρικά ότι η συνάρτηση **`np.linalg.inv()`** λειτουργεί σωστά

```
a = np.array([[1,5,6],[3,4,2],[4,2,1]])
print(a @ np.linalg.inv(a))
[[1.00000000e+00 0.00000000e+00 2.22044605e-16]
 [1.11022302e-16 1.00000000e+00 2.22044605e-16]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

🤓 Χρησιμοποιώντας πιο πολλές συναρτήσεις της NumPy μπορούμε να τρέξουμε κατευθείαν:

```
print(np.all(np.isclose(a@np.linalg.inv(a),
                        np.eye(3))))
```

Αποθήκευση πίνακα στο δίσκο

Αποθηκεύω τη μεταβλητή **a** στο αρχείο **data.npy**

```
np.save('data.npy', a)
```

Φορτώνω το αρχείο **data.npy** στη μεταβλητή **b**

```
b = np.load('data.npy')
```

🤓 αν θέλω να σώσω πολλούς πίνακες στο ίδιο αρχείο (**.npz**), μπορώ να χρησιμοποιήσω τη **.savez()** ή τη **.savez_compressed()**

Κλείνοντας (;) με τη NumPy

- Τα εργαλεία και οι μέθοδοι της NumPy είναι αρκετά για να γεμίσουν ένα εξάμηνο από μόνα τους
- Σε πολλές περιπτώσεις, η NumPy απαιτεί προσαρμογές στον τρόπο σκέψης για να τα αξιοποιήσουμε
  - Πολυδιάστατα δεδομένα, Views vs Copies, Numbers→Bools →Indexes…
- Το σημαντικότερο είναι να μπορούμε να *καταλάβουμε* κώδικα όταν τον βλέπουμε, και να έχουμε τη διάθεση να συνεχίσουμε να *μαθαίνουμε* όσο χρησιμοποιούμε τα εργαλεία αυτά

Κλείνοντας (not) με την Python

- Παρά τις αλλαγές των τελευταίων χρόνων (*LLMs*...), η παρακάτω αλήθεια φαίνεται να παραμένει σε ισχύ:

Υπάρχουν πάρα πολλά προβλήματα εκεί έξω που μπορούν να λυθούν με ένα μικρό Python script και τις κατάλληλες βιβλιοθήκες

  - Θέλω να ενώσω 100 excel books επιλέγοντας κάποια φύλλα από το καθένα
  - Θέλω να στήσω ένα τοπικό μοντέλο μηχανικής μάθησης που θα ταξινομεί εταιρικά έγγραφα
  - Θέλω να μετασχηματίσω αυτόματα όλες τις καταχωρήσεις του καταλόγου επαφών της εταιρείας στο νέο format που αποφασίσαμε

- Ακόμα κι αν έχω εξωτερική βοήθεια (LLM, documentation, συνεργάτες) πρέπει να μπορώ να καταλάβω τι κάνω!