Προγραμματιστικά Εργαλεία και Τεχνολογίες για Επιστήμη Δεδομένων

Παράδοση 6/10/2020, Νίκος Παπασπύρου.

NumPy

1. Το NumPy είναι μία εξαιρετική βιβλιοθήκη της Python για επιστημονικούς υπολογισμούς. Μεταξύ άλλων υποστηρίζει:

   - έναν αποδοτικό τύπο για N-διάστατους πίνακες
   - δυνατότητα ενσωμάτωσης συναρτήσεων γραμμένων σε C/C++ ή Fortran
   - χρήσιμες συναρτήσεις και αλγορίθμους (γραμμική άλγεβρα, μετασχηματισμός Fourier, γεννήτρια τυχαίων αριθμών, κ.λπ.)

2. Για να το χρησιμοποιήσουμε, βεβαιωνόμαστε πρώτα ότι υπάρχει εγκατεστημένο στον υπολογιστή μας (το `numpy` δεν είναι πακέτο της βασική βιβλιοθήκης της Python).

   Συνήθως, το κάνουμε `import` και το μετονομάζουμε σε κάτι συντομότερο, π.χ. `np`, για διευκόλυνσή μας.

   ```
   import numpy as np
   ```

3. Το NumPy υποστηρίζει ομογενείς πολυδιάστατους πίνακες (arrays), πολύ αποδοτικότερους από τις λίστες της Python. Παρακάτω φαίνεται εν τάχει ο τρόπος ορισμού και η βασική τους χρήση. Πρώτα ένας μονοδιάστατος πίνακας (διάνυσμα):

   ```
   >>> a = np.array([1, 2, 3])
   >>> print(a)
   [1 2 3]
   >>> print(type(a))
   <class 'numpy.ndarray'>
   >>> print(a.ndim)
   1
   >>> print(a.shape)
   (3,)
   >>> print(a[1])
   2
   ```

   Προσέξτε ότι τα arrays είναι mutable και ότι η αρίθμηση των στοιχείων ξεκινάει από το μηδέν:

   ```
   >>> a[1] = 42
   >>> print(a)
   [ 1 42  3]
   ```

   Στη συνέχεια ένας διδιάστατος πίνακας:

   ```
   >>> b = np.array([[1, 2, 3], [4, 5, 6]])
   >>> print(b)
   [[1 2 3]
    [4 5 6]]
   >>> print(type(b))
   <class 'numpy.ndarray'>
   >>> print(b.ndim)
   2
   >>> print(b.shape)
   (2, 3)
   >>> print(b[0][1])
   2
   >>> print(b[0, 1])
   2
   >>> b[0, 1] = 17
   >>> print(b)
   [[ 1 17  3]
    [ 4  5  6]]
   ```

   Τα πεδία `ndim` και `shape` των arrays μας δίνουν το πλήθος των διαστάσεων και το μέγεθος των πινάκων, αντίστοιχα. Προσέξτε ότι στον διδιάστατο πίνακα, τα `b[0][1]` και `b[0, 1]` είναι ισοδύναμα — αναφέρονται στο δεύτερο στοιχείο

της πρώτης γραμμής. Για την ακρίβεια, το `b[0]` αναφέρεται στην πρώτη γραμμή του πίνακα `b`:

```
>>> print(b[0])
[ 1 17  3]
>>> print(b[0].shape)
(3,)
```

4. Οι πίνακες λειτουργούν ως iterables:

```
>>> for x in a:
...   print(x)
...
1
42
3
```

Προσέξτε όμως ότι σε πολυδιάστατους πίνακες, το iterable αυτό διατρέχει μόνο την πρώτη διάσταση:

```
>>> for x in b:
...   print(x)
...
[ 1 17  3]
[4 5 6]
```

δηλαδή τα δύο στοιχεία που τυπώθηκαν ήταν η πρώτη και η δεύτερη γραμμή αντίστοιχα. Μπορούμε να διατρέξουμε όλα τα στοιχεία ενός πολυδιάστατου πίνακα ως εξής:

```
>>> for x in b.flat:
...   print(x)
...
1
17
3
4
5
6
```

5. Οι πίνακες είναι **ομογενείς** δηλαδή όλα τους τα στοιχεία είναι του ίδιου τύπου. Αυτό είναι διαφορετικό από τη φιλοσοφία της Python, π.χ. οι λίστες της μπορούν να περιέχουν στοιχεία διαφορετικών τύπων. Το πεδίο `dtype` ενός πίνακα μας δείχνει το data type των στοιχείων του.

```
>>> print(b.dtype)
int64
```

Το `np.int64` είναι ο τύπος των 64-bit ακέραιων αριθμών. Είναι διαφορετικός από τον τύπο των ακεραίων στην Python (`int`), ο οποίος υποστηρίζει αριθμούς οσοδήποτε μεγάλους (bignums). Εξίσου χρήσιμος, αν όχι χρησιμότερος, είναι ο τύπος `np.float64`.

```
>>> c = np.array([3.14, 2.78])
>>> print(c)
[3.14 2.78]
>>> print(c.dtype)
float64
```

6. Πίνακες με ιδιαίτερη μορφή και περιεχόμενο. Ο μηδενικός πίνακας:

```
>>> z = np.zeros((3, 3))
>>> print(z)
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Προσέξτε ότι ο πίνακας αυτός έχει στοιχεία με τύπο `np.float64`. Αν θέλαμε να έχουμε ακέραια μηδενικά, μπορούμε να δώσουμε τιμή στην (προαιρετική) παράμετρο `dtype`:

```
>>> print(np.zeros((3, 3), dtype=np.int64))
[[0 0 0]
 [0 0 0]
```

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

Το ίδιο ακριβώς μπορούμε να πάρουμε με `dtype=int` — ο τύπος ακεραίων της Python αυτόματα μετατρέπεται σε `np.int64` σε αυτή την περίπτωση.

Ένας πίνακας γεμάτος με άσους, ή οποιοδήποτε άλλη σταθερά:

```
>>> n = np.ones((3, 3))
>>> print(n)
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
>>> f = np.full((3, 3), 42)
>>> print(f)
[[42 42 42]
 [42 42 42]
 [42 42 42]]
```

Μοναδιαίος πίνακας:

```
>>> i = np.eye(3)
>>> print(i)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Πίνακες που αντιστοιχεί σε κάποιο `range`:

```
>>> print(np.arange(10))
[0 1 2 3 4 5 6 7 8 9]
>>> print(np.arange(4, 10))
[4 5 6 7 8 9]
>>> print(np.arange(4, 10, 2))
[4 6 8]
```

Πίνακας με τυχαίους αριθμούς, στο διάστημα \([0, 1)\).

```
>>> r = np.random.random((3, 3))
>>> print(r)
[[0.78302488 0.85854877 0.03570157]
 [0.70774711 0.24283631 0.92667939]
 [0.34008778 0.68366237 0.22990968]]
```

7. **Slicing**: Μία από τις χρησιμότερες λειτουργίες που υποστηρίζει το NumPy είναι η δυνατότητα να "κόβουμε φέτες" (slicing) πινάκων. Αυτό γίνεται με παρόμοια σύνταξή όπως αυτή που χρησιμοποιούμε π.χ. για φέτες λιστών στην Python

```
>>> s = i[:2, 1:3]
>>> print(s)
[[0. 0.]
 [1. 0.]]
>>> print(s[1, 0])
1.0
```

Εν αντιθέσει όμως με τις φέτες λιστών στην Python (που κάθε φέτα κατασκευάζει αντίγραφο των στοιχείων της αρχικής λίστας), στο NumPy οι φέτες είναι απλές "όψεις" (views) μέρους ενός πίνακα. Αυτό γίνεται αντιληπτό με το παρακάτω παράδειγμα, στο οποίο αλλάζουμε ένα στοιχείο της φέτας `s` και παρατηρούμε ότι αλλάζει και το αντίστοιχο στοιχείο του αρχικού πίνακα `i`:

```
>>> s[1, 0] = 42
>>> print(s)
[[ 0.  0.]
 [42.  0.]]
>>> print(i)
[[ 1.  0.  0.]
 [ 0. 42.  0.]
 [ 0.  0.  1.]]
```

Αν θέλουμε να αντιγράψουμε μια φέτα (η οποιονδήποτε άλλο πίνακα) μπορούμε να χρησιμοποιήσουμε τη μέθοδο `copy`:

```
>>> c = i[:2, 1:3].copy()
>>> c[1, 0] = 17
>>> print(c)
[[ 0.  0.]
 [17.  0.]]
>>> print(i)
[[ 1.  0.  0.]
 [ 0. 42.  0.]
 [ 0.  0.  1.]]
```

Οι φέτες μπορούν να περιορίζονται και σε συγκεκριμένα στοιχεία μίας διάστασης. Προσέξτε τη διαφορά ανάμεσα στα παρακάτω:

```
>>> print(i[2, 1:3])
[0. 1.]
>>> print(i[2:3, 1:3])
[[0. 1.]]
```

Το πρώτο είναι ένας μονοδιάστατος πίνακας, ενώ το δεύτερο ένας διδιάστατος πίνακας \(1 \times 2\).

8. **Mass indexing**: Μπορούν να χρησιμοποιηθούν πίνακες ως indices σε έναν πίνακα. Δείτε το παρακάτω παράδειγμα:

```
>>> i = np.eye(3)
>>> print(i)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>> x = np.arange(3)
>>> print(x)
[0 1 2]
>>> y = np.array([2, 0, 1])
>>> print(y)
[2 0 1]
```

Χρησιμοποιώντας τα `x` και `y` ως indices στον πίνακα `i` παίρνουμε μία μονοδιάστατη όψη των στοιχείων `i[0,2]`, `i[1,0]` και `i[2,1]` (δηλαδή των στοιχείων `i[x[j], y[j]] for j in range(3)`).

```
>>> print(i[x, y])
[0. 0. 0.]
```

Η όψη αυτή είναι mutable:

```
>>> i[x, y] += 5
>>> print(i)
[[1. 0. 5.]
 [5. 1. 0.]
 [0. 5. 1.]]
```

9. **Boolean indexing**: Μπορεί επίσης να χρησιμοποιηθεί ένας πίνακας μέσα σε μία λογική παράσταση. Στην περίπτωση αυτή, τα στοιχεία του πίνακα συμμετέχουν ένα προς ένα στη λογική παράσταση και κατασκευάζεται ένας πίνακας με τις λογικές τιμές:

```
>>> j = i > 0
>>> print(j)
[[ True False False]
 [False  True False]
 [False False  True]]
```

Ο πίνακας `j` έχει ίδιες διαστάσεις με τον `i` και περιέχει `True` στις θέσεις όπου το αντίστοιχο στοιχείο του `i` είναι μεγαλύτερο του μηδέν.

Τέτοιοι πίνακες όπως ο `j` μπορούν επίσης να χρησιμοποιηθούν για mass indexing:

```
>>> print(i[j])
[1. 5. 5. 1. 5. 1.]
```

```
>>> print(i[i == 5])
[5. 5. 5.]
>>> i[i==5] = 42
>>> print(i)
[[ 1.  0. 42.]
 [42.  1.  0.]
 [ 0. 42.  1.]]
```

10. **Αριθμητική πινάκων**: Οι πίνακες μπορούν να χρησιμοποιηθούν σε αριθμητικές πράξεις και στην περίπτωση αυτή οι πράξεις εφαρμόζονται στα στοιχεία τους ένα προς ένα:

```
>>> print(x)
[0 1 2]
>>> print(y)
[2 0 1]
>>> print(x+y)
[2 1 3]
>>> print(x*y)
[0 0 2]
```

Προσέξτε ότι στην τελευταία εντολή υπολογίζεται το γινόμενο των επιμέρους στοιχείων των πινάκων x και y, που προφανώς είναι διαφορετικό τόσο από το γινόμενο πινάκων όσο και από το εσωτερικό γινόμενο. Αν θέλουμε το εσωτερικό γινόμενο των διανυσμάτων x και y μπορούμε να χρησιμοποιήσουμε τη συνάρτηση `np.dot` ή τον τελεστή `@`:

```
>>> print(np.dot(x, y))
2
>>> print(x @ y)
2
```

Πολλές ακόμη συναρτήσεις ορίζονται στο NumPy, π.χ.

```
>>> print(np.sqrt(x))
[0.         1.         1.41421356]
```

Ο ανάστροφος ενός πίνακα κατασκευάζεται με τη συνάρτηση 'np.transpose' ή με την εξής συντομοτραφία:

```
>>> print(np.transpose(i))
[[ 1. 42.  0.]
 [ 0.  1. 42.]
 [42.  0.  1.]]
>>> print(i.T)
[[ 1. 42.  0.]
 [ 0.  1. 42.]
 [42.  0.  1.]]
```

Προσέξτε ότι και ο ανάστροφος είναι απλά μία όψη του πίνακα:

```
>>> i.T[0, 1] = 17
>>> i.T
array([[ 1., 17.,  0.],
       [ 0.,  1., 42.],
       [42.,  0.,  1.]])
>>> i
array([[ 1.,  0., 42.],
       [17.,  1.,  0.],
       [ 0., 42.,  1.]])
```

11. **Tiling**: Έστω ότι έχουμε ένα διδιάστατο πίνακα x και έναν μονοδιάστατο πίνακα v:

```
>>> x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
>>> print(x)
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
>>> v = np.array([1, 0, 1])
```

```
>>> print(v)
[1 0 1]
```

Θέλουμε να κατασκευάσουμε έναν πίνακα y προσθέτοντας τον πίνακα v σε κάθε γραμμή του x. Για να το κάνουμε αυτό, μπορούμε πρώτα να κατασκευάσουμε έναν "άδειο" πίνακα με τις ίδιες διαστάσεις όπως ο x και στη συνέχεια να αναθέσουμε το σωστό άθροισμα σε κάθε γραμμή του:

```
>>> y = np.empty_like(x)
>>> for i in range(4):
...     y[i] = x[i] + v
...
>>> print(y)
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Είναι όμως πολύ αποδοτικότερο, αντί να χρησιμοποιήσουμε ένα `for` loop, να "μεγαλώσουμε" τον πίνακα v αντιγράφοντάς τον σε τέσσερις γραμμές και κατασκευάζοντας έτσι ένα διδιάστατο πίνακα \(4 ×3\):

```
>>> z = np.tile(v, (4, 1))
>>> print(z)
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

Στη συνέχεια, μπορούμε να προσθέσουμε τους πίνακες x και z και να πάρουμε το ίδιο αποτέλεσμα με πριν:

```
>>> y = x + z
>>> print(y)
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Ένα ακόμη παράδειγμα πλακόστρωσης (tiling), σε δύο διαστάσεις αυτή τη φορά:

```
>>> print(x)
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
>>> print(np.tile(x, (2, 3)))
[[ 1  2  3  1  2  3  1  2  3]
 [ 4  5  6  4  5  6  4  5  6]
 [ 7  8  9  7  8  9  7  8  9]
 [10 11 12 10 11 12 10 11 12]
 [ 1  2  3  1  2  3  1  2  3]
 [ 4  5  6  4  5  6  4  5  6]
 [ 7  8  9  7  8  9  7  8  9]
 [10 11 12 10 11 12 10 11 12]]
```

12. Το παραπάνω tiling του πίνακα v μπορεί να αποφευχθεί, χάρη σε ένα μηχανισμό που το NumPy ονομάζει **broadcasting**:

```
>>> y = x + v
>>> print(y)
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Το broadcasting καθορίζει πώς γίνονται οι πράξεις μεταξύ πινάκων με διαφορετικό σχήμα. Αυτό που συμβαίνει συνηθέστερα είναι ότι ο μικρότερος πίνακας "μεγαλώνει" για να φτάσει σε διαστάσεις τον μεγαλύτερο. Είναι η ίδια διαδικασία που ακολουθείται και όταν στις πράξεις συμμετέχουν αριθμοί αντί πινάκων:

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 2, 2])
>>> print(a*b)
[2 4 6]
>>> b = 2
>>> print(a*b)
[2 4 6]
```

Περισσότερα για το broadcasting και τις εφαρμογές του μπορείτε να βρείτε στην αντίστοιχη τεκμηρίωση του NumPy.

13. **Reshaping**: Μπορούμε να φτιάξουμε μία όψη διαφορετικού σχήματος με τη μέθοδο `reshape`:

```
>>> a = np.arange(15)
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
>>> s = a.reshape(3, 5)
>>> print(s)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

Προσέξτε όμως ότι και πάλι πρόκειται για όψη, όχι αντίγραφο:

```
>>> s[1, 1] = 42
>>> print(s)
[[ 0  1  2  3  4]
 [ 5 42  7  8  9]
 [10 11 12 13 14]]
>>> print(a)
[ 0  1  2  3  4  5 42  7  8  9 10 11 12 13 14]
```