Obliczenia Naukowe

Wektory i Macierze w pythonie
Czyli wprowadzenie do NumPy

Bartek Wilczyński
bartek@mimuw.edu.pl

5. marca 2018

Szczypta historii

- Pakiety BLAS, ATLAS i LAPACK w Fortranie
- Pakiety numeric i ndarray– pierwsze łączenia z pythona do tych bibliotek
- Scipy – scientific python, zawierał integrację w postaci numpy
- 2006 Numpy wyłączony ze scipy, żeby nie komplikować zależności
- Ciekawa prezentacja nt. Historii:

  http://history.siam.org/presentations.htm

Biblioteka numpy

- Importowanie:
  - `import numpy`,
  - albo **`import numpy as np`**
- Raczej nie ~~`from numpy import *`~~, bo zawiera funkcje (np. max) konfiktujące z biblioteką standardową
- Część większej biblioteki scipy.
- Służy jako efektywna implementacja macierzy i wektorów
- Zawiera bardzo dużo sprawdzonych implementacji operacji na macierzach

Tworzenie wektorów i macierzy

- `a=np.array([1,2,3,4])` → wektor
- `b=np.arange(4)`
- `a.shape` → `(4,)`
- `a.reshape((2,2))` → macierz 2x2
- `a.reshape((4,1))` → wektor “pionowy”
- `a.reshape((1,4))` → wektor “poziomy”
- `np.transpose(a)` ?
- `np.arange(8).reshape((2,2,2))` ?

Tworzenie macierzy c.d.

- `Z= np.zeros((2,2))`
- `Js= np.ones((2,3))`
- `Z2= np.zeros_like(Js)`
- `Fs= np.array([1,2,3], dtype=float)`
- `np.linspace(0,1,3) → array([0,.5,1])`
- `xs=np.linspace(0,1,4)→`
  `array([0,.33,.67,1.])`
- `xs.dtype → float64`
- `a.dtype → int64`

Operacje arytmetyczne (wykonywane “pozycyjnie”)

- `a=np.array([1,2]); b=np.array([3,4])`
- `a+b → array([4,6])`
- `a-b → array([-2,-2])`
- `a/b → array([.33,.5])`
- `a*b → array([3,8]) !!!`
- `a**2 → array([1,4])`
- `np.concatenate((a,b)) → array([1,2,3,4])`
- `c=np.arange(4).reshape((2,2))`
- `a+c ? #”broadcasting”`

Wybieranie elementów tablic

- `m=np.arange(25).reshape((5,5))`
- `v=np.arange(10)`
- `v[1:5]`
- `m[2:4,1:3]`
- `m[::2,::3]`
- `v.take([2,4,6,9])`
- `m.take([3,5,7]) ??`
- `v.put([2,4],[10,20])`
- `np.split(m,5)`

Wpisywanie do macierzy

- `m=np.arange(25).reshape((5,5))`
- `m[1,1]=5`
- `m[2]=6`
- `m[2:4,2:4]=m[0:2,0:2]`
- `m.fill(10)`

Iterowanie po macierzach

- `a=np.arange(10).reshape((5,2))`
-
```
for row in a:
    print row
```

-
```
for x in a.reshape((10,)):
    print x
```

-
```
for x,y in a:
    print x,y
```

Macierze w pamięci

- Macierze w pamięci zachowują się inaczej niż listy
- W momencie tworzenia zajmują ciągły blok pamięci
- Pozwala to na szybkie znajdowanie elementów o zadanym indeksie
- Macierze wielowymiarowe też tak naprawdę są “liniowe” w pamięci (operacja reshape nie “przestawia” danych w pamięci)
- “wstawienie” elementu lub zmiana rozmiaru są operacjami “kosztownymi”

Kopiowanie macierzy

- Pamiętamy, że w pythonie są obiekty mutable i unmutable. Macierze należą do mutable.
- `n0=m #nie kopia, tylko wska?nik`
- `n1=m.copy()`
- `n2=np.copy(m)`
- `n3=np.array(m,copy=True)`
- `n4=m.view() #nie-kopia, dzieli dane`
- `n5=m.view()[1:3,1:3] #wgl?d do fragmentu`

Serializacja macierzy

- Macierze możemy zapisywać na dysku w plikach binarnych .npy
- `np.save(“nazwa.npy”,macierz)`
- `np.save(file_handle,macierz)`
- A potem je wczytywać:
- `m=np.load(“nazwa.npy”)`
- `m=np.load(file_handle)`

Mnożenie macierzy

- m*n # “po pozycjach”
- m.dot(n) # klasyczne mnożenie macierzy
- Dla wektorów mamy też mnożenie tensorowe:
- np.outer(v1,v2)
- i iloczyn wektorowy w R^3:
- np.cross(v1,v2)

Operacje na macierzach

- `M<5 ?`
- `m<n ?`
- `np.any(M<5)`
- `np.all(M>5)`
- `10 in m`
- `m.sort() # domy?lnie w wierszach`
- `m.argsort()`
- `m.min(), m.argmin(), m.max(), m.argmax()`

Co to jest ufunc?

- Funkcje typu max, czy min działają szybko bo korzystają z wektorowej architektury procesora
- Takie wektorowe funkcje w numpy nazywają się ufunc (universal function)
- Jeśli chcemy napisac jakąś “szybką” funkcję, możemy skorzystać z funkcji frompyfunc()
- Np: np.frompyfunc(lambda x:x+3,1,1)(m)