

Efektivní realizace operací nad n-dimenzionálními poli

za pomoci knihovny NumPy

Vojtěch MRÁZEK

Fakulta informačních technologií, Vysoké učení technické v Brně
Brno, Czech Republic
mrzek@fit.vutbr.cz



Datové typy v Pythonu

- Abychom byli schopni efektivně pracovat s daty a výpočty, musíme vědět, jak jsou data ukládána.
- Největší výhoda (a limitace) jazyka Python je dynamické typování.

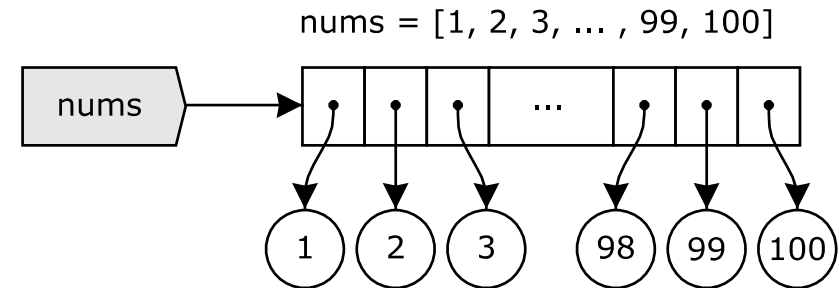
```
# Python  
x = 1  
x = "jedna"
```

```
/* C / C++ */  
int x;  
x = 1;  
x = "jedna"; // neprojde
```

Opakování: jak vlastně je reprezentován Integer v Pythonu

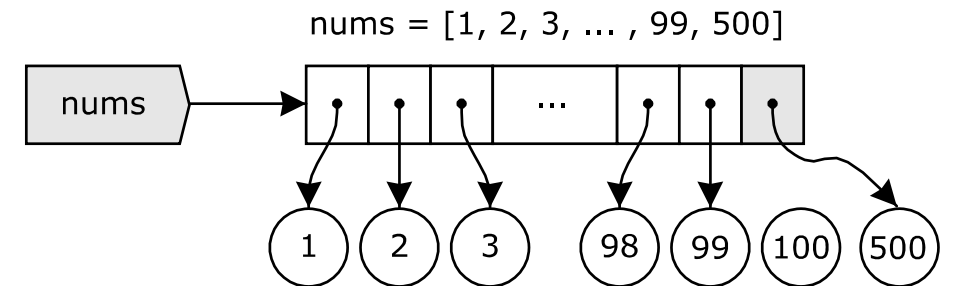
- Všechny proměnné jsou ukazatelem na nějaký objekt. Pro integer např. platí:

```
struct _longobject {  
    long ob_refcnt;  
    PyTypeObject *ob_type;  
    size_t ob_size;  
    long ob_digit[1];  
};
```



- Obsah struktury

- `ob_refcnt` počet odkazů v paměti
- `ob_type` datový typ
- `ob_size` velikost datového typu
- `ob_dig` vlastní data



- Viz zdrojový kód:

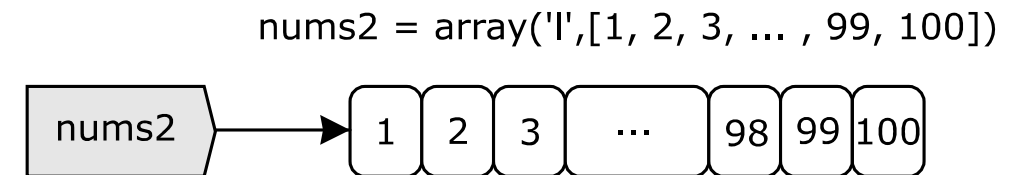
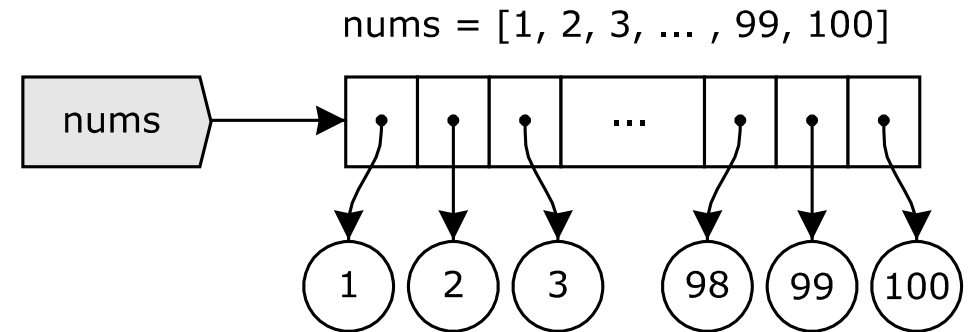
- <https://github.com/python/cpython/blob/master/Include/longintrepr.h#L85-L88>
- <https://github.com/python/cpython/blob/master/Include/floatobject.h>

A co seznam (list)?

- Vzniká tedy *overhead* při práci, zejména se seznamem, který může být nehomogenní.

- Např. při

- Změně prvku (vytváření nového objektu)
- Vyšetřování datového typu
- Pole odkazů => není paměťová lokalita

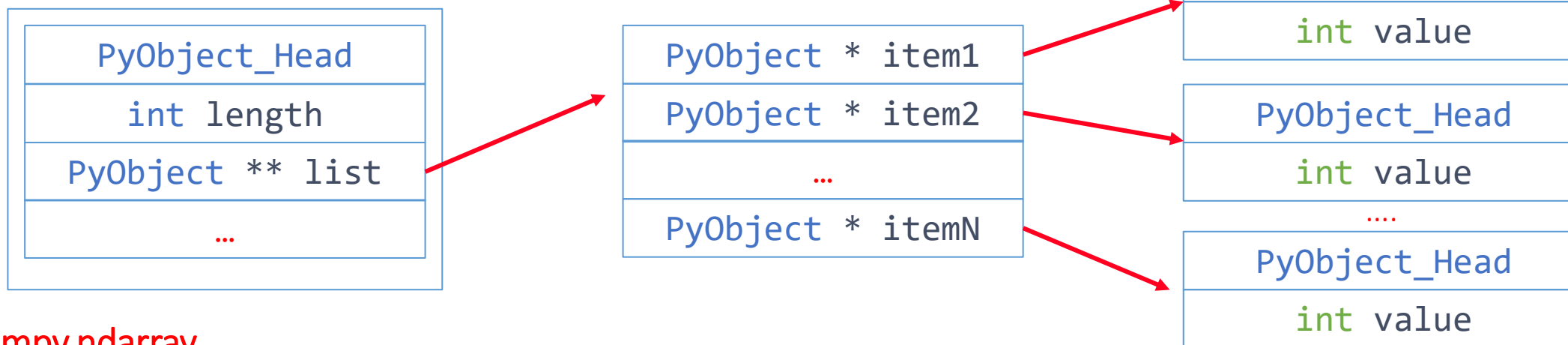


- Potřeba efektivních operací nad homogenními poli vede k novým strukturám

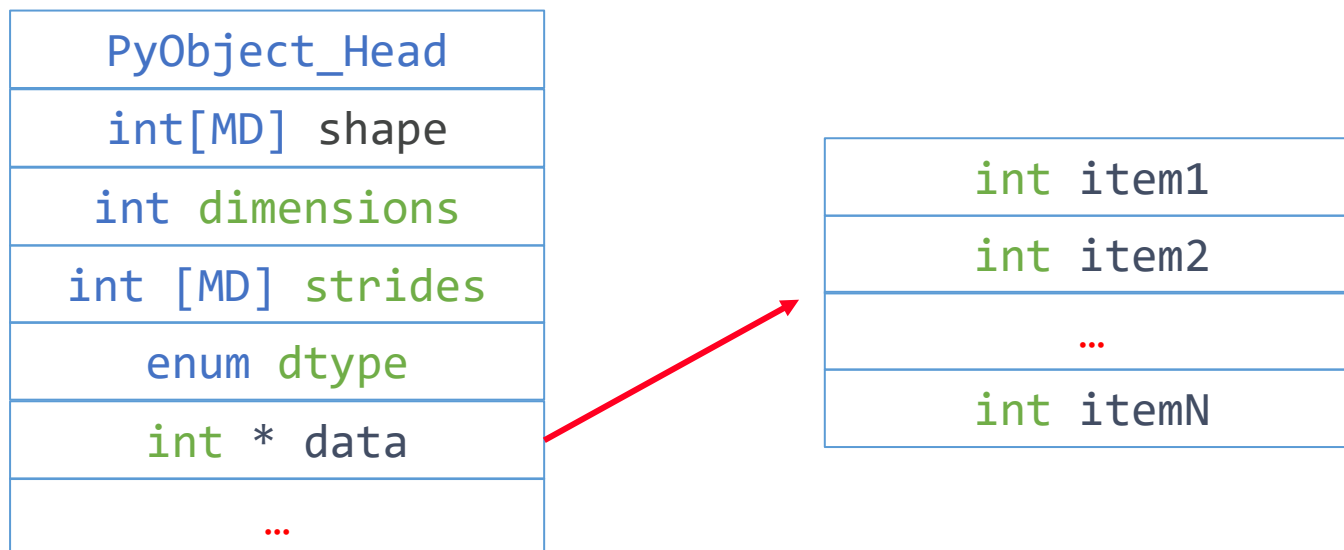
- Knihovna *Array*: <https://docs.python.org/3/library/array.html>
 - Kromě reprezentace obsahuje pouze funkce pro manipulaci s daty (efektivnější paměť), při operacích (procházení) se však zase vytváří Python objekty
- Knihovna *NumPy*: <https://numpy.org/>
 - Koncept z *Array* je rozšířen o rychlé operace přímo nad novou reprezentací (nedochází k transformacím)

Reprezentace polí v paměti

List [int]



numpy.ndarray



Co se musí udělat, když chceme číst konkrétní hodnotu? A když chceme číst parametr pole (např. počet dimenzí)?

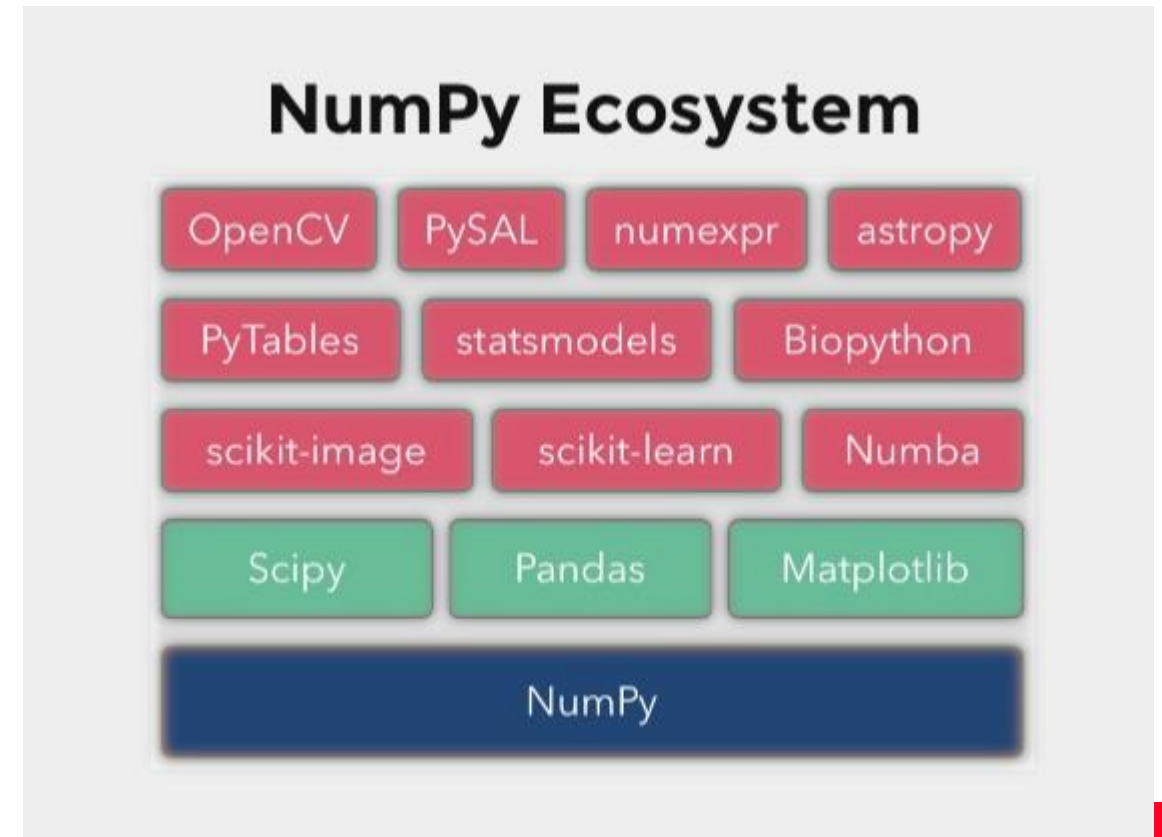
Jak by se změnilo rozložení v paměti, pokud bychom chtěli mít 3D pole?

Numerical Python (NumPy)



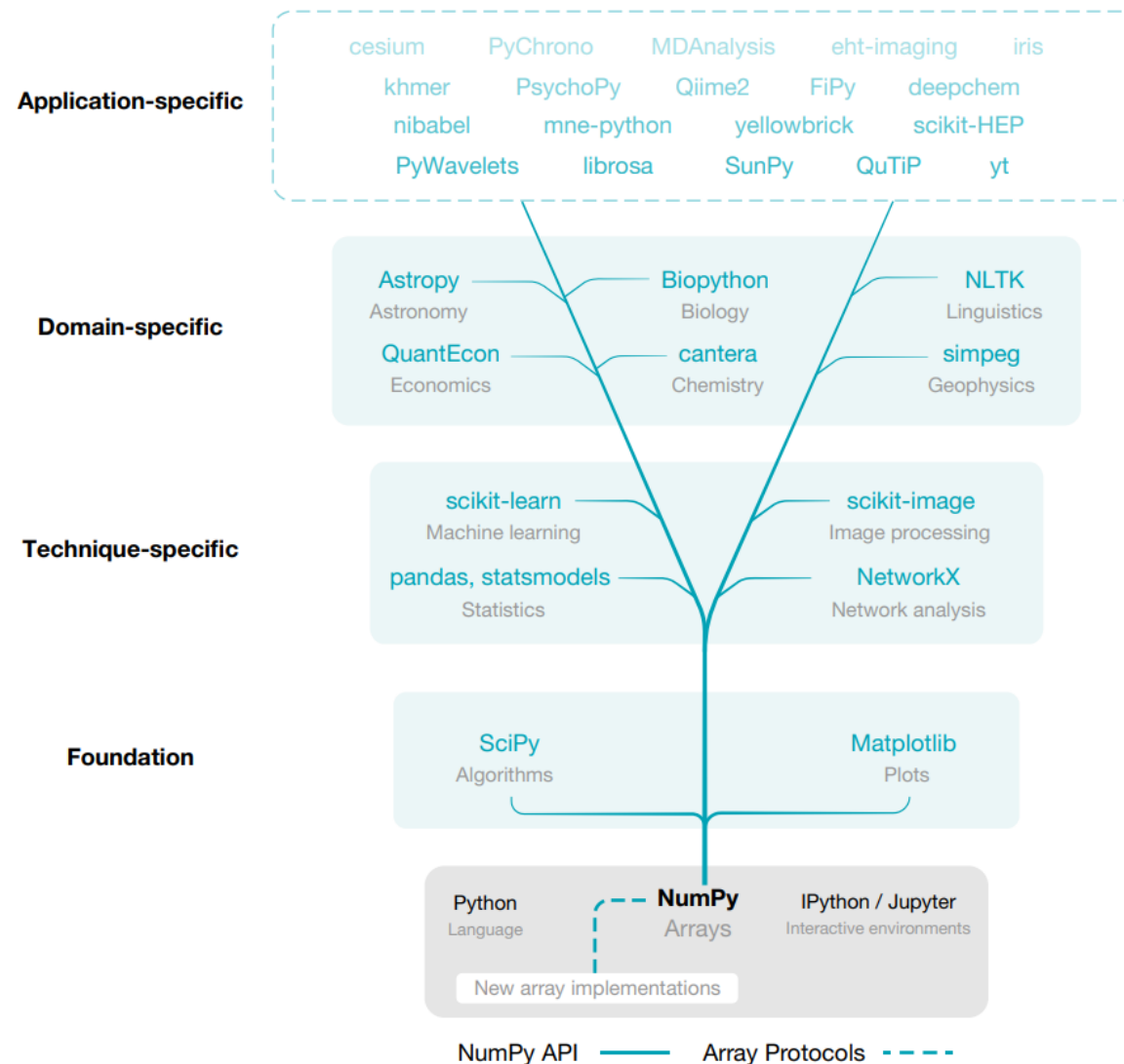
- Externí modul pro práci s poli implementovaný v jazyce C :
 - efektivní
 - in-memory
 - kontinuální uložení v paměti
 - homogenní
- NumPy je vhodné na řadu aplikací
 - Zpracování obrazu
 - Zpracování signálu
 - Lineární algebra
 - Strojové učení
 - a mnoho dalších

```
import numpy as np
```







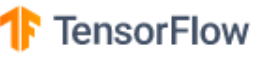








NumPy jako standard

- NumPy je základem celého vědeckého ekosystému v jazyce Python
- Existuje celá řada dalších implementací polí (tzv. [NumPy Array Protocol](#)) rozhraní je však podobné, existujepak kompatibilita s NumPy
- Díky tomu je přechod na nové knihovny snadný
- <https://www.nature.com/articles/s41586-020-2649-2>



Příklady knihoven s NumPy Array Protocol

	Array Library	Capabilities & Application areas
	Dask	Distributed arrays and advanced parallelism for analytics, enabling performance at scale.
	CuPy	NumPy-compatible array library for GPU-accelerated computing with Python.
	JAX	Composable transformations of NumPy programs: differentiate, vectorize, just-in-time compilation to GPU/TPU.
	Xarray	Labeled, indexed multi-dimensional arrays for advanced analytics and visualization
	Sparse	NumPy-compatible sparse array library that integrates with Dask and SciPy's sparse linear algebra.
	PyTorch	Deep learning framework that accelerates the path from research prototyping to production deployment.
	TensorFlow	An end-to-end platform for machine learning to easily build and deploy ML powered applications.
	MXNet	Deep learning framework suited for flexible research prototyping and production.
	Arrow	A cross-language development platform for columnar in-memory data and analytics.
	xtensor	Multi-dimensional arrays with broadcasting and lazy computing for numerical analysis.
	XND	Develop libraries for array computing, recreating NumPy's foundational concepts.
	uarray	Python backend system that decouples API from implementation; unumpy provides a NumPy API.
	TensorLy	Tensor learning, algebra and backends to seamlessly use NumPy, MXNet, PyTorch, TensorFlow or CuPy.

NumPy pole – parametry – datový typ

Numpy type	C type	Description
np.bool_	bool	Boolean (True or False) stored as a byte
np.byte	signed char	Platform-defined
np.ubyte	unsigned char	Platform-defined
np.short	short	Platform-defined
np.ushort	unsigned short	Platform-defined
np.intc	int	Platform-defined
np.uintc	unsigned int	Platform-defined
np.int_	long	Platform-defined
np.uint	unsigned long	Platform-defined
np.longlong	long long	Platform-defined
np.ulonglong	unsigned long long	Platform-defined
np.half/np.float16		Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
np.single	float	Platform-defined single precision float: typically sign bit, 8 bits exponent, 23 bits mantissa
np.double	double	Platform-defined double precision float: typically sign bit, 11 bits exponent, 52 bits mantissa.
np.longdouble	long double	Platform-defined extended-precision float
np.csingle	float complex	Complex number, represented by two single-precision floats (real and imaginary components)
np.cdouble	double complex	Complex number, represented by two double-precision floats (real and imaginary components).
np.clongdouble	long double complex	Complex number, represented by two extended-precision floats (real and imaginary components).

Poznámka: je možné použít i [řetězcovou reprezentaci](#) (b, i, u, f, c, m, ...), používanou např. v strukturovaných datových typech, pro specifikaci little- nebo big-endianu atd.



NumPy pole – parametry – rozměr

- Tvar (shape) – definovaný pomocí n-tice (tuple)

shape = (5)

1	2	3	4	5
---	---	---	---	---

prvek x[3]

shape = (3, 5)

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

prvek x[1, 3]

shape = (2, 3, 5)

	16	17	18	19	20
1	2	3	4	5	
6	7	8	9	10	
11	12	13	14	15	

prvek x[1, 2, 3]

Numpy – rozložení v paměti

■ Z dokumentace

- *An instance of class ndarray consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps N integers into the location of an item in the block.*

shape[1] = 5

shape[0] = 3

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

int64 => 8 B

Parametr **strides** nám říká, o kolik musíme v každé dimenzi přeskakovat. Kolik to bude v tomto případě?

Poznámka: Nastavením parametru **order** při vytváření z defaultního „C“ formátu na „Fortran“ chování změníme

0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	2
16	0	0	0	0	0	0	3
24	0	0	0	0	0	0	4
32	0	0	0	0	0	0	5
40	0	0	0	0	0	0	6
48	0	0	0	0	0	0	7
	0	0	0	0	0	0	8
	0	0	0	0	0	0	9
	0	0	0	0	0	0	10
	0	0	0	0	0	0	11
	0	0	0	0	0	0	12
	0	0	0	0	0	0	13
	0	0	0	0	0	0	14
	0	0	0	0	0	0	15

Inicializace

- Numpy pole můžeme vytvořit z N-dimenzionálního seznamu

```
np.array([[1, 2, 3], [4, 5, 6]])
```

- U kterého je možné specifikovat typ přímo

```
np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float)
```

- Nebo je typ odvozen

```
np.array([[1, 2, 3], [4, 5, 6]])  
np.array([[1, 2, 3], [4, 5.000012, 6]])  
np.array([[1, 2, 3], [4, 5.000012+4j, 6]])
```

- Vstupem může být libovolný iterovatelný objekt

%timeit np.array(range(1000000))	216 ms ± 41 ms per loop
%timeit np.arange(1000000)	3.62 ms ± 1.43 ms per loop

Inicializace pole konstantními hodnotami

- Pole typicky vytváříme inicializované nulami, jedničkami, konstantami, nebo neinicializované

```
np.zeros(10, dtype=int)
np.ones(shape=(3,5))
np.full((3,5), 3.14)
```

- Často vytváříme i pole, které má stejné parametry, jako pole jiné (např. pro uložení výsledků)

```
x = np.zeros(shape=(5, 4), dtype="f")
np.ones(shape=x.shape, dtype=x.dtype)
np.ones_like(x)
```

- Můžeme však vytvořit i neinicializované pole

```
np.empty(shape=(3,5))      np.empty_like(x)
```

```
%timeit np.empty(shape=(100000))
%timeit np.zeros(shape=(100000))
```

1.63 μ s \pm 474 ns per loop
23.6 μ s \pm 3.62 μ s per loop



Inicializace pole hodnotami v číselného rozsahu

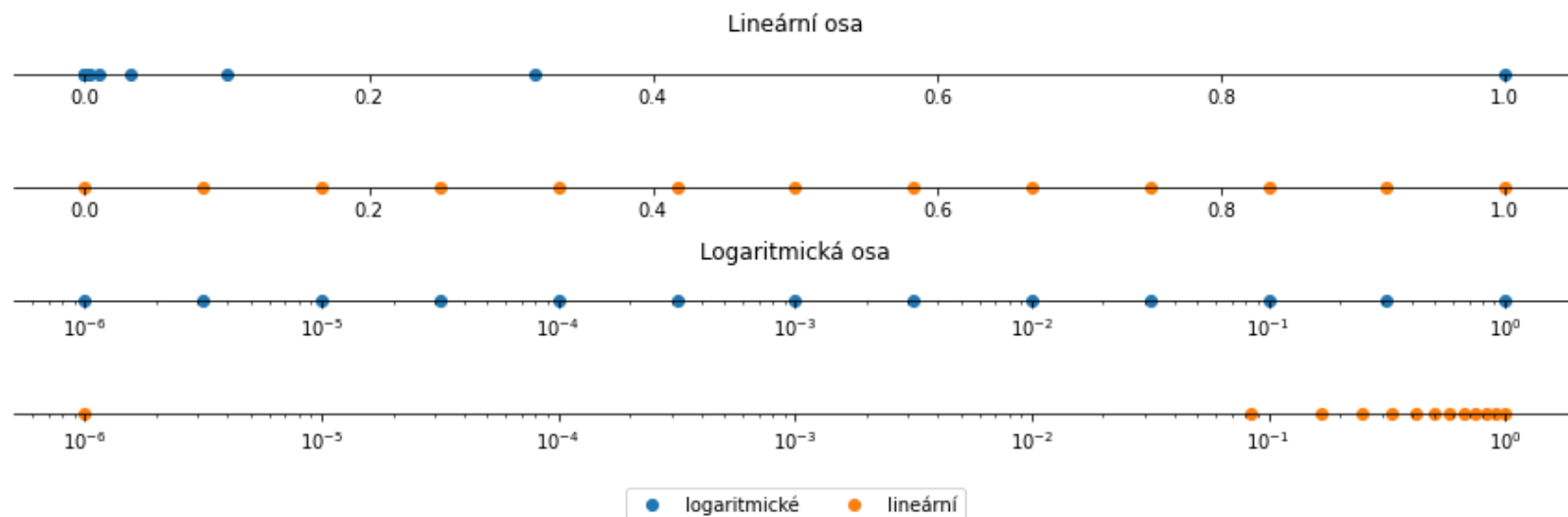
- Pole můžeme rozsahem hodnot definované začátkem, koncem a krokem

```
np.arange(1000)
np.arange(10, 100, 2)
np.arange(10.1, 100, 2) # vytváří float pole
```

- Pokud nechceme uvádět krok, můžeme nechat rozdělit rozsah automaticky (linárně i logaritmicky)
- Tento přístup je výhodný pro zejména pro vizualizaci

```
np.linspace(1, 10, 5)
```

```
np.geomspace(1, 1e6, 13)
np.logspace(0, 6, 13, base=10)
```



Inicializace pole náhodnými hodnotami

- Dalším způsobem plnění je použití náhodných hodnot.

- Ty můžeme vybírat

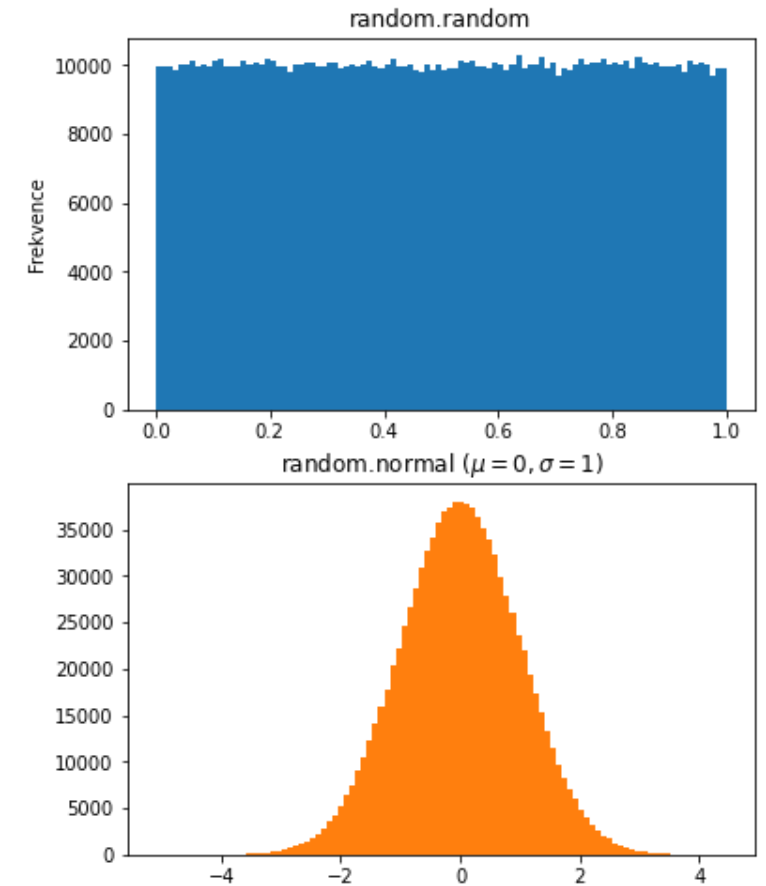
- Ze seznamu hodnot

```
np.random.choice([0, 1, 2], size=(4, 5))
```

- Na základě rozdělení

```
# normalni rozdeleni (u=0, s=1)
np.random.normal(0, 1, (3,3))
# uniformní
np.random.random((3, 3))
np.random.randint(0, 10, (3, 3))
```

- Více možností (většinu známých náhodných rozložení) pak vygenerujeme pomocí SciPy (více v příštích přednáškách)



Možnosti vytváření polí

- Pomocí existujících Python objektů
 - `array`, `fromfile`, `fromfunction`, `fromiter`, ...
- Pole konstant
 - `zeros`, `ones`, `empty`, `full`, ...
- Číselná posloupnost
 - `arange`, `linspace`, `logspace`, `geomspace`, ...
- Náhodné hodnoty
 - `random.random`, `random.norma`, `random.randint`, ...
- Více viz [dokumentace](#)

Práce s poli

■ Atributy polí

- dimenze (**ndim**), tvar (**shape**), velikost (**size**)

■ Přístup prvkům

- jednotlivé, rozsahy, zápis do rozsahu
- <https://numpy.org/doc/stable/reference/arrays.indexing.html>

■ Manipulace s poli

- bez změny dat:
 - změna velikosti **reshape**
 - změna typu **view**
- se změnou dat:
 - konverze typu **astype**
- další operace
 - konkatenace, rozdělení
- <https://numpy.org/doc/stable/reference/routines.array-manipulation.html>

```
x.astype(int)
```

Indexace polí

- Operátor `[]` řešen pomocí interních funkcí `__getitem__` pro čtení a `__setitem__` pro zápis. Důsledek?

```
x = np.arange(10)
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

- Indexovat můžeme pomocí n-tice (n je počet dimenzí), kdy každá položka může být jedním z:

- Celočíselnou hodnotou

```
x[0], x[1], x[-2], x[-1]
```

0	1	8	9
---	---	---	---

- Instancí objektu typu `slice`

```
x[slice(None, None, 2)]
```

```
x[::2]
```

0	2	4	6	8
---	---	---	---	---

```
x[1::2]
```

1	3	5	7	9
---	---	---	---	---

```
x[-1:2:-1])
```

9	8	7	6	5	4	3
---	---	---	---	---	---	---

- Polem celočíselných hodnot (*Fancy indexing*)

```
x[[-1, 2, 3]]
```

9	2	3
---	---	---

- Booleovským polem

```
x[[True, True, False, False, True, True, False, False, False, False]]
```

0	1	4	5
---	---	---	---

Indexace polí pro zápis

- Analogicky můžeme stejnou indexaci použít i pro zápis

```
x = np.arange(10)
```

```
x[0] = 12
```

```
x[1:4] = [42, 43, 44]
```

```
x[1::3] = [45, 46, 47]
```

```
x[9::-3] = [48, 49, 50, 51]
```

```
x[[True, True, False, False, True,
    True, False, False, False, False]]
= [52, 53, 54, 55]
```

```
x[[1, 2]] = [56, 57]
```

```
x[[1, 2]] = [58, 59, 60]
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

12	1	2	3	4	5	6	7	8	9
----	---	---	---	---	---	---	---	---	---

0	42	43	44	4	5	6	7	8	9
---	----	----	----	---	---	---	---	---	---

0	45	2	3	46	5	6	47	8	9
---	----	---	---	----	---	---	----	---	---

51	1	2	50	4	5	49	7	8	48
----	---	---	----	---	---	----	---	---	----

51	52	2	3	53	54	6	7	8	9
----	----	---	---	----	----	---	---	---	---

0	56	57	3	4	5	6	7	8	9
---	----	----	---	---	---	---	---	---	---

ValueError: shape mismatch: value array of shape (3,) could not be broadcast to indexing result of shape (2,)



Vícedimenzionální pole

- Indexace je analogická, pouze se jedná o n-tici

`x[2, 1, 0]`

3

`x[[True, False, False], 0, :]`

9	0	2	1	6
---	---	---	---	---

```
array([[[9, 0, 2, 1, 6],
       [3, 7, 7, 9, 3],
       [0, 1, 5, 9, 9],
       [1, 6, 7, 7, 7]],
      [[3, 6, 2, 1, 7],
       [7, 8, 3, 1, 7],
       [8, 0, 6, 5, 2],
       [9, 3, 1, 7, 6]],
      [[0, 1, 8, 9, 6],
       [3, 5, 7, 8, 0],
       [9, 6, 8, 2, 0],
       [1, 0, 8, 3, 6]]])
```

- Kolik bude mít výstup dimenzí?

`x[2,1,0]`

0 – skalární hodnota

`x[2,3,:]`

1 – vektor

`x[2, :, 3]`

1 – vektor

`x[:, :, 3]`

2 – matice

`x[:, :, [3,4]]`

3 – tensor

Práce s pamětí, změna rozměrů

- Kopírování – proč je potřebujeme?

```
y = x
x[0, 0] = -1
print(y[0, 0])
```

```
y = x.copy()
x[0, 0] = -1
print(y[0, 0])
```

- Samostatný typ „view“ – ukazuje na stejná RAW data, ale mění hlavičku
- Konverze rozměrů

```
y = np.arange(64).reshape(4,4,4)
y = np.arange(64).reshape(16, -1)
y = np.arange(64).reshape(-1, 16)
```

- Mění reshape data? Jaké má tento přístup výhody?

```
y = x.reshape((4, -1))
x[-1] = 0
print(y[0, 0])
```

```
x.reshape(-1) } Vytváří pohled (view)
x.ravel()
x.flatten()    } Dělá kopii
```

Kombinace a podvýběry z pole

grid =

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
np.vsplit(grid, [2])
np.split(grid, [2], axis=0)
[grid[:2, :], grid[2:, :]]
```

0	1	2	3
4	5	6	7

8	9	10	11
12	13	14	15

```
np.hsplit(grid, [2, 3])
np.split(grid, [2, 3], axis=1)
[grid[:, :2], grid[:, 2:3], grid[:, 3:]]
```

0	1
4	5
8	9
12	13

2
6
10
14

3
7
11
15

a =

b =


```
np.concatenate([a, b])
```


```
np.concatenate([a, b], axis=1)
```


Manipulace s poli

reshape & ravel

```
a1 = np.arange(1, 13)
```

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

```
a1.reshape(3, 4)
a1.reshape(-1, 4)
a1.reshape(3, -1)
.ravel() # back to 1D
```

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

```
a1.reshape(3, -1, order='F')
.ravel(order='F') # back to 1D
```

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

stack

```
a1 = np.arange(1, 13)
a2 = np.arange(13, 25)
np.stack((a1, a2), axis=1)
np.hstack((a1, a2))
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

3D array from 2D arrays

```
a1 = np.arange(1, 13).reshape(3, 4)
a2 = np.arange(13, 25).reshape(3, -1)
```

1	2	3	4	13	14	15	16
5	6	7	8	17	18	19	20
9	10	11	12	21	22	23	24

```
# stack along axis 2
a3_2 = np.stack((a1, a2), axis=2)
a3_2.shape: (3, 4, 2)
```

1	2	3	4	13	14	15	16
5	6	7	8	17	18	19	20
9	10	11	12	21	22	23	24

```
# retrieve a1
a3_2[:, :, 0]
```

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

```
# stack along axis 0
a3_0 = np.stack((a1, a2))
a3_0.shape: (2, 3, 4)
```

1	2	3	4	13	14	15	16
5	6	7	8	17	18	19	20
9	10	11	12	21	22	23	24

```
# retrieve a1
a3_0[0, :, :]
```

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

```
# stack along axis 1
a3_1 = np.stack((a1, a2), axis=1)
a3_1.shape: (3, 2, 4)
```

1	2	3	4	13	14	15	16
5	6	7	8	17	18	19	20
9	10	11	12	21	22	23	24

```
# retrieve a1
a3_1[:, 0, :]
```

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

flatten 3D array

```
# flatten/ravel
a3_0.ravel()
# flatten/ravel
a3_0.ravel(order='F')
```

1	2	3	4	5	...	20	21	22	23	24
---	---	---	---	---	-----	----	----	----	----	----

reshape 3D array

```
# reshape from (2, 3, 4) to (4, 2, 3)
a3_0.reshape(4, 2, 3)
```

1	2	3	4	13	14	15	16
5	6	7	8	17	18	19	20
9	10	11	12	21	22	23	24

Matematické funkce – sčítání `np.add`

■ Jaké typicky používáme operace?

$$a + b$$

1	2	3	4	5
---	---	---	---	---

+

6	7	8	9	10
---	---	---	---	----

7	9	11	13	15
---	---	----	----	----

`np.add(a, b)`

$$a += b$$

1	2	3	4	5
---	---	---	---	---

+

6	7	8	9	10
---	---	---	---	----

7	9	11	13	15
---	---	----	----	----

`np.add(a, b, out=a)`

1	2	3	4	5
---	---	---	---	---

+

10	11	12
----	----	----

11	13	3	4	17
----	----	---	---	----

`np.add.at(a, [0, 1, 4], c)`

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

`np.add.reduce(a, axis=1)`

15	40	65
----	----	----

18	21	24	27	30
----	----	----	----	----

`np.add.reduce(a, axis=0)``np.add.reduce(a, axis=None)`

120

`np.sum(a)`

1	2	3	4	5
---	---	---	---	---

1	3	6	10	15
---	---	---	----	----

`np.add.accumulate(a)``np.cumsum(a)`

Zobecnění – koncept universálních funkcí (ufunc)

- Univerzální funkce – statická třída realizující následující operace
 - `__call__(a, [b])`
 - `at(a, indices, [b])`
 - `reduce(a, [axis=0])`
 - `reduceat(a, indices, [axis=0])`
 - `outer(a, b)` – pro všechna a v A a b v B vrací $a \text{ op } b$ (A, B jsou zploštělé, výsledek je 2D)
 - `accumulate(a, [axis=0])`
 - Poslední čtyři mají význam pouze u binárních operátorů (dvouvstupých funkcí)
- <https://numpy.org/doc/stable/reference/ufuncs.html>
- <https://numpy.org/doc/stable/reference/ufuncs.html#methods>

Dostupné univerzální funkce

- Matematické funkce
 - `add`, `subtract`, `multiply`, `matmul`, `divide`, `logaddexp`, `true_divide`, `floor_divide`, `negative`, `abs`, `mod`, `power`, `log`, `sqrt`, `square`, ...
- Trigonometrické funkce
 - `sin`, `cos`, `tan`, `arcsin`, `sinh`, `radians`, `degrees`, ...
- Bitové funkce
 - `bitwise_and`, `bitwise_or`, `invert`, `left_shift`, ...
- Komparační funkce – co vracejí za typ?
 - `greater`, `less`, `less_equal`, `logical_or`, `logical_not`, ...
 - Pozor na používání zkrácených výrazů – nikdy se nesmí mezi dvěma bool poli použít operátory `and`, `or`, `not`, ..., musíme používat vždy `&` | `~`
 - Pro použití např. v podmínkách musíme udělat redukci `or` (`np.any`) nebo `and` (`np.all`)
- Float funkce
 - `isinf`, `fabs`, `isnan`, ...
- Poznámky:
 - některé funkce (např. `sum`, `max`, ...) mají i tzv. *nansafe* verzi (`nansum`, `nanmax`)

Řazení, hledání a počítání

■ Řadicí funkce

- `sort`, `lexsort` (stabilní), `ndarray.sort` (in-place), ...

```
np.sort(x) == x[x.argsort()]
```

■ Vyhledávací funkce

- `max`, `min`, `where` („multiplexor“), `argwhere` (vrací indexy, kdy je hodnota v poli True) ...

Někdy použito nevhodně tam, kde by stačila indexace boolovským polem

```
x = np.arange(10)
np.where(x < 5, x*10, x)
#array([ 0, 10, 20, 30, 40,  5,  6,  7,  8,  9])
```

■ Počítací funkce

- `count_nonzero`, `sum`, ...

```
np.sum(a > 5)
```

Operace s řetězci

- Modul [numpy.char](#) obsahuje sadu vektorizovaných řetězových operací pro pole typu `numpy.str_` nebo `numpy.bytes_`
- Obsahuje funkce podobné vestavěným řetězcovým funkcím v Pythonu
Operace: `lower`, `replace`, `split`, `title`, ...
Komparace: `equal`, `less`, ...
Informace: `endswith`, `find`, `index`, ...
- Implementace však [není efektivní](#) – přechází se na `PyObject` a pak zpět. V tomto případě se tomu však nevyhneme ☹
- Příklad nahrazení čárky tečkou

```
a = np.array([f"{i//10},{i%10}" for i in range(10000)])  
# 0,0    0,1 ...  
a.astype("f") # ValueError: could not convert string to float: '0,0'
```

```
a = np.array([f"{i//10},{i%10}" for i in range(10000)])  
# 0,0    0,1 ...  
a = np.char.replace(a, ",", ".").astype("f")
```

Numpy pole PyObjectů

- Datový typ se u pole určí automaticky

```
a = np.array([1, 2, 3])  
a.dtype # int32
```

- V některých případech se však typ určí špatně (typicky výsledek nějaké operace), potom vznikne pole Python objektů <- *proč to je špatně?*

```
a = np.arange(100000).astype("O")  
b = np.arange(100000).astype("f")  
%timeit a + 1000  
# 2.77 ms ± 79.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)  
%timeit b + 1000  
# 14 µs ± 519 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```



Ukázka vylepšeného indexování

Zadání je k dispozici v podpůrných souborech jako Jupyter notebook. Pro otestování správného pochopení problematiky je doporučeno úkoly reprodukovat.

Broadcasting – příklad sčítání

- Jedna z vlastností pro zpříjemnění práce s poli.
- Řeší, co se stane při operacích s poli s různou velikostí.
- Závislosti při sčítání polí A a B:
 - pokud A a B nemají stejný počet dimenzí, menšímu se zvětší (chybějící dimenze se doplní rozměrem 1)
 - porovnávají se jednotlivé dimenze A a B. Ty musí být
 - stejné, nebo
 - jedna větší jak 1 a druhá 1. Ta s rozměrem 1 se „rozkopíruje“ do velikosti větší dimenze.

```
np.arange(3) + 5
```

0	1	2	+	5	5	5	=	5	6	7
---	---	---	---	---	---	---	---	---	---	---

```
np.ones((3,3)) + np.arange(3)
```

1	1	1		0	1	2		1	2	3
1	1	1	+	0	1	2	=	1	2	3
1	1	1		0	1	2		1	2	3

```
np.arange(3).reshape(-1,1) + np.arange(3)
```

0	0	0		0	1	2		0	1	2
1	1	1	+	0	1	2	=	1	2	3
2	2	2		0	1	2		2	3	4

<https://numpy.org/doc/stable/user/theory.broadcasting.html?highlight=broadcasting>



Důsledky broadcastingu (příklady)

- V počítání (např. přičítání skalární hodnoty)

```
a = np.arange(5)
b = np.full(5, 2) # pole dvojek
a + b == a + 2
```

- V indexování (vynechání jedné hodnoty)

```
x = np.arange(12).reshape(4,3)
x[[1, 1, 1], [0, 1, 2]] == x[1, [0, 1, 2]]
```

- Doplnění hodnot přičítání

```
a = np.arange(5)
np.add.at(a, [0, 0, 0, 0, 0], 1) # a = array([5, 1, 2, 3, 4])
```

- Přiřazení do podpole

```
a = np.random.random(size=(2, 3, 4))
a[:, [0, 1]] = 0
# array([[0.          , 0.          , 0.          , 0.          ],
#        [0.          , 0.          , 0.          , 0.          ],
#        [0.41854698, 0.74885643, 0.28944482, 0.34195551]],
#
#        [[0.          , 0.          , 0.          , 0.          ],
#        [0.          , 0.          , 0.          , 0.          ],
#        [0.19944179, 0.797123  , 0.55776715, 0.1056725  ]]])
```

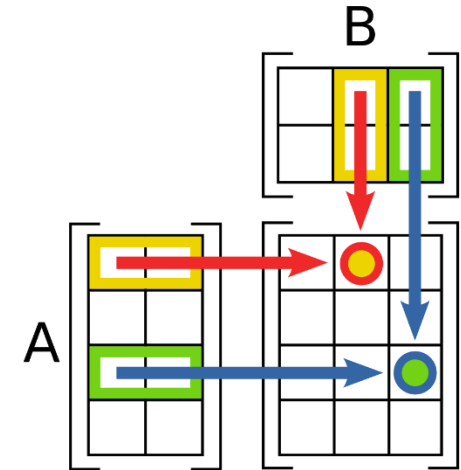

Využití broadcasting v praxi

Zadání je k dispozici v podpůrných souborech jako Jupyter notebook. Pro otestování správného pochopení problematiky je doporučeno úkoly reprodukovat.



Maticové operace

- Dříve vlastní datový typ `np.matrix`, nyní však není doporučen k používání – využívá se `np.ndarray`
- Násobení matic (`np.dot`, operátor `@`)
- Implementace nástrojů pro lineární algebru
 - Implementováno jako rozhraní pro BLAS a LAPACK (např. OpenBLAS, MKL™, ATLAS)
 - Řešení rovnic
 - Vlastní vektory
 - Normální formy
- <https://numpy.org/doc/stable/reference/routines.linalg.html>



Ukázka maticových operací a lineární algebry

- Mějme soustavu rovnic

$$3x_0 + x_1 = 9$$

$$x_0 + 2x_1 = 8$$

- Což můžeme převést na matici c a vektor r

$$c = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}, r = [9 \quad 8]$$

- Pomocí lineární algebry nalezneme řešení

$$res = [2 \quad 3]$$

- neboli $x_0 = 2$ a $x_1 = 3$

- Následně ověříme, že
 $c \cdot res^T = r$

```
c = np.array([[3, 1], [1, 2]])  
r = np.array([9, 8])
```

```
res = np.linalg.solve(c, r)
```

```
np.allclose(np.dot(c, res), r)
```

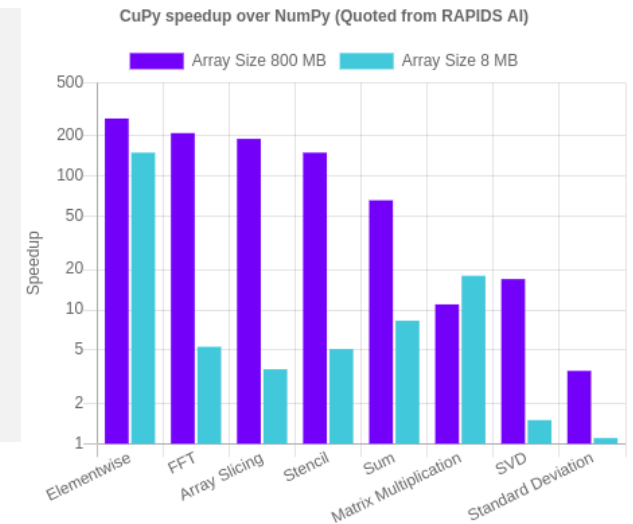
Operace **dot** nevyžaduje transponování



Příklad využití knihovny CuPy pro akceleraci kódu na GPU

CuPy is an open-source array library accelerated with NVIDIA CUDA. CuPy provides GPU accelerated computing with Python. CuPy uses CUDA-related libraries including cuBLAS, cuDNN, cuRand, cuSolver, cuSPARSE, cuFFT and NCCL to make full use of the GPU architecture.

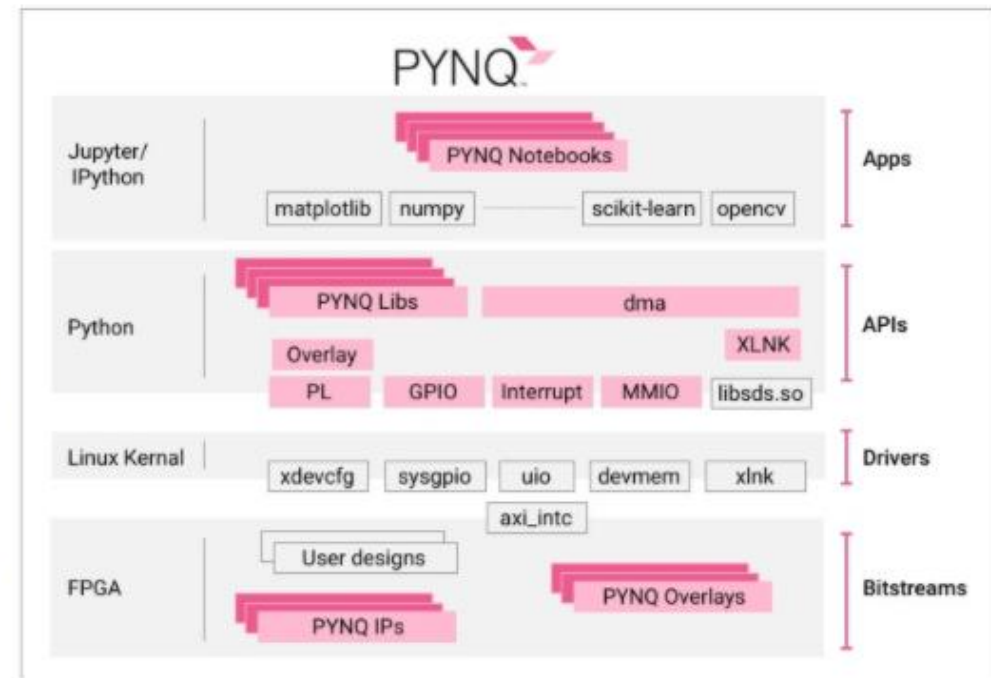
```
In [1]: import numpy as np
In [2]: import cupy as cp
In [3]: a_np = np.ones((10**4, 10**4))
In [6]: a_cp = cp.ones((10**4, 10**4))
In [7]: %timeit np.sum(a_np)
68.4 ms ± 216 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [8]: %timeit cp.sum(a_cp)
48 µs ± 23.6 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```



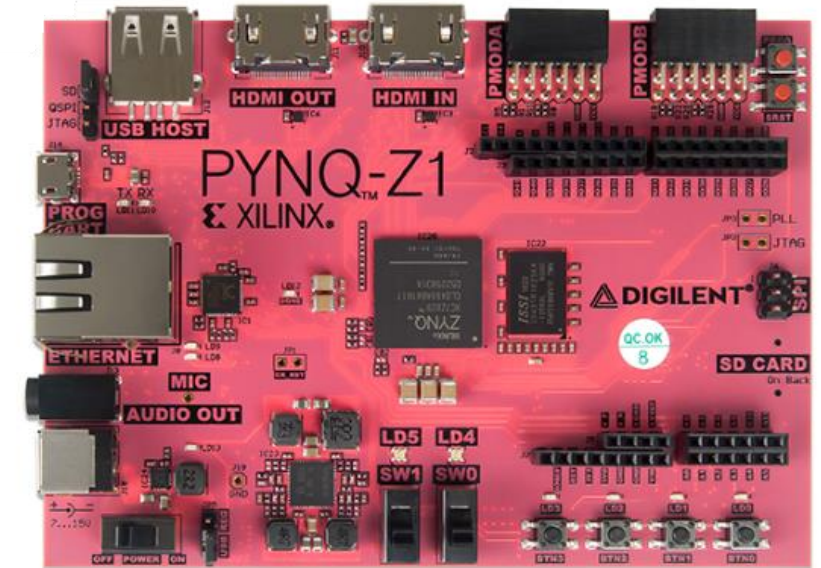
CuPy

Akcelerace NumPy v FPGA

- V high-end systémech [Amazon](#)
- I v low-end systému Zynq (FPGA + ARM)



Domain / Topic	Title / Author / DOI	Improvement vs CPU+GPU	Improvement vs CPU-Only
Digital Signal Processing Sliding Windows	A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding Window Applications, Fowers, http://dx.doi.org/10.1145/2145694.2145704	11x	57x
Graph Processing Tree-reweighted Message Passing (TRW-S)	GraphGen for CoRAM: Graph Computation on FPGAs, Weisz, http://dx.doi.org/10.1109/FCCM.2014.15	10.3x	14.5x
Monte Carlo Simulation Random Number Generation	A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation, Thomas, http://dx.doi.org/10.1145/1508128.1508139	3x	30x
Machine Vision Moving Average with Local Difference (MALD)	CPU, GPU and FPGA Implementations of MALD: Ceramic Tile Surface Defects Detection Algorithm, Hocenski, http://dx.doi.org/10.7305/automatika.2014.01.317	14x	35x
Bioinformatics <i>De Novo</i> Genome Assembly	Hardware Accelerated Novel Optical <i>De Novo</i> Assembly for Large-Scale Genomes, Kastner, http://dx.doi.org/10.1109/FPL.2014.6927499	8.5x	11.9x
Atmospheric Modelling Solvers for Global Atmospheric Equations	Accelerating Solvers for Global Atmospheric Equations through Mixed-Precision Data Flow Engine, Gan, http://dx.doi.org/10.1109/FPL.2013.6645508	4x	100X



Co jsme přeskočili

■ Strukturované datové typy

```
record_type = np.dtype([("a", "<i8"), ("b", np.float_), ("c", np.bool_)])
d = np.array([(1, 3.4, True)]*8, dtype=record_type)
d
# array([(1, 3.4,  True), (1, 3.4,  True), (1, 3.4,  True), (1, 3.4,  True),
#        (1, 3.4,  True), (1, 3.4,  True), (1, 3.4,  True), (1, 3.4,  True)],
#        dtype=[('a', '<i8'), ('b', '<f8'), ('c', '?')])
d["a"]
# array([1, 1, 1, 1, 1, 1, 1, 1], dtype=int64)
d[0]
# (1, 3.4, True)
```

■ Ukládání dat

- np.load, np.save, np.savez
- <https://numpy.org/doc/stable/reference/routines.io.html>

■ Komplikované příklady

- <https://www.labri.fr/perso/nrougier/from-python-to-numpy/>

TIP!

■ Další funkce

- FFT
- Finanční funkce
- C-funkce
- Práce s řídkými maticemi (sparse matrices)
- Akcelerace pomocí překládaného kódu => 13. přednáška

- <https://numpy.org/doc/stable/reference/>

Rekapitulace (co si zapamatovat)

- NumPy řeší výkonnostní problémy, musíme však přemýšlet nad ukládáním dat
- Pozor na neustálé vytváření nových objektů
 - Vytváří přechodné (temporary) pole
$$a = a + 1$$
 - Rovnou zapisuje do pole a
$$a += 1$$
- Vektorizace přináší novou abstrakci, ale pokud zpracujeme velké množství dat, tak jsou výsledky výborné
- Pozor na čitelnost kódu
- [Cheatsheet](#)

TIP!

Děkuji za pozornost, dotazy?