Szoftver mély neuronhálók alkalmazásához

4. előadás

Varga Viktor, Kovács Bálint
ELTE IK Mesterséges Intelligencia Tanszék

A Python programozási nyelv

Alap tulajdonságok

+ Tömör, magas szinten absztraktált => könnyen érthető
+ Interpretált => gyors fordítás
+ Dinamikus, erős típusrendszer
+ Többparadigmás
+ Open-source, sok külső csomag elérhető hozzá: https://pypi.org/
- Párhuzamosítási lehetőségek korlátozottak
- Egyes feladatokhoz lassú lehet

Kommentelés

```
# This is a comment

# This is a
# multiline comment
```

```
"""
This is also a
multiline comment
"""
```

Azonosítók

- Első karakter: betű (kis vagy NAGY) vagy aláhúzás (_)
- Többi: Alfanumerikus (betű vagy szám) vagy aláhúzás (_)
- Éķēżéťêķēt ļèħëţ ħäšžṅáłņï... (csak nem ajánlott)
- Kulcsszavak foglaltak:

```
for in and assert import or as except if break else . . .
```

- CaSe SENsITive

Értékadás

```
x = 5
x += 1 # C-style increment operator
x /= 3

a = b = c = 1 # all will have the same value
x, y = 5, 6 # multiple assignment
```

Python objektumok

3 fő jellemzőjük:

- Memóriacímük => lekérhető az **id()** függvénnyel
- Típusuk. Vannak alaptípusok, konténertípusok és felhasználó által definiált típusok => lekérhető a `type()` függvénnyel
  - Ez alapján lehet mutable(változhat az értékük). Ilyen a legtöbb konténertípus és felhasználó által definiált típus
  - Es lehet immutable(nem változhat az értékük). Ilyen az összes alaptípus. Hasheléskor használhatóak kulcsként
- Értékük

Számtípusok és számműveletek

```
x = 6 # x will be an int
y = 5.0 # y will be a float

5 + 3 # result is an int
5 + 3.0 # result is a float
5 / 5 # result is a float

5 % 3 # modulo operator
2 ** 10 # power operator
```

Egyéb alaptípusok

```
i = True # boolean
s = "asd" # string
b = b"\x5Bhexadecimal\x5D \133octal\135" # byte

null_value = None
print(type(null_value)) # output: <class 'NoneType'>
```

Logikai műveletek

- Összehasonlítás: `< > <= >= == !=`
- Műveletek: `a and b, a or b, not a`
- Bitenkénti műveletek: `a & b, a | b, ~a, a^b`
- Komplex kifejezések: `3 < x <= 10`
- Referencia összehasonlítás, ugyanarra az objektumra mutatnak-e:

  `x is None, x is not None`

Iterálható konténertípusok

- Képesek egyesével visszaadni az elemeiket
- Így lehet rajtuk iterálni, például `for` ciklussal
- A gyakorlatban ez azt jelenti, hogy értelmezhető rá az **__iter__()** függvény, mely egy iterátor objektumot ad vissza
- A **`__next__()`** függvénnyel érjük el a következő elemet
- Iterátort át lehet konvertálni szekvencia típussá, mint például a lista
- `StopIteration` exceptiont vált ki, ha elfogytak az iterálható elemek

Adatszerkezetek - lista

- C-ben írt pointer tömbként van megvalósítva
- Elemei lehetnek különböző típusúak
- Dinamikus memóriafoglalás
- ‘Mutable’ típus: módosítható
- Elem elérése/felülírása konstans idejű
- Végéhez való hozzáadás(`append()`) és törlés(`pop()`) szintén konstans idejű
- Középső elem törlése/beillesztése O(n) idejű, mert át kell mozgatni az utána lévő elemeket

Lista inicializáció és elemek elérése

```
basic_list = [1,2,3,4,5,6,7,8,9]

empty_list = []

list_of_list = [[2,3], ['r',[]], 'a', [None]]


# this function returns an int, the length of the given list
length = len(basic_list) # output:9

# Indexing starts with ZERO and goes to length-1
basic_list[0] # output:1

# Negative index ''-k' corresponds to index length-k
basic_list[-2] # output:8
```

Slice-olás

- Egyszerre több elem elérésének egy egyszerű módja
- Szintaxisa: **`List[from:to:step]`**
- Ez a **`from`** indextől(azt is belevéve) a **`to`** indexig(azt nem belevéve) adja vissza az elemeket, **`step`** lépésközzel.
- Slice-olás másolatot ad vissza, viszont ha értékadás bal oldalán szerepel, akkor a slice-olt lista tartalmát írja felül

```
my_list = [0,1,2,3,4,5,6,7,8,9]
my_list[2:len(my_list):1] # From idx 2 to the end
my_list[2:] # default values are the 0:len(list):1, they
can be omitted
```

Slice-olás - példák

```
my_list[2:4] # step is omitted
my_list[:2] # from and step is omitted

my_list[::2] # output: the whole list with step size 2
my_list[::-1] # output: the whole list reversed
my_list[::-2] # Reversed list with step size 2:

my_list[2:8:2] # From idx 2 to idx 8 (exclusive) with step
size 2:

my_list[-2::-2] # "Reversed list with step size 2, starting
from second but last element towards the beginning

my_list[7:20] # When slicing, out of range indices will not
raise an error, so same as my_list[7:]
```

Lista műveletek

```
my_list.append(10) # Append one element to the end

my_list.extend([11, 12]) # Extended the list with two
elements

my_list.insert(-1, None) # Insert an element to the back of
the list (index -1)

item = my_list.pop(3) # Remove 4th item
# other list methods:

my_list.remove(), my_list.clear(), my_list.sort(), ...
```

Tuple(rendezett n-es)

```
tuple1 = (1, "a", [])

tuple1[0] = 6 # immutable
# output: TypeError: 'tuple' object does not support item
assignment

# tuple packing and unpacking
t1 = 1, 2, "asd" # packing 3 elements into a tuple
a1, b1, c1 = t1 # unpacked a 3 long tuple into 3 variables

len(t1) # length of a tuple
```

Halmazok(set-ek)

- A halmazokban minden elem egyszer szerepelhet
- Vegyes típusok is lehetnek, de csak immutable típusok
- Az,hogy egy elem benne van-e a halmazban, O(1) idő alatt eldönthető hash függvényes megvalósítás miatt
- Listánál ugyanez O(n)

```
s1 = {1,2,1,2,"a", "a"}

# Duplicate elements are stored once in the set

# s1 is {1, 2, 'a'}


t1 = (1,2,3)

s2 = set(t1) # A set can be constructed from any iterable


#set operations

s1.add(0) # Adding an element

s1.union(s2) # or s1 | s2

s1.difference(s2)) # or s1 - s2
```

Szótárak(dict-ek)

```
d = {1: 'bla', 'bb': 42, (): ['hello'], None: 0.5} # unique keys

d.keys() # an iterator over the keys of the dictionary
d.values() # an iterator over the values of the dictionary:
d.items() # an iterator over the (key, value) pairs of the dictionary

d[1] # dictionary element access

d[(3, "a")] = None # setting an element in a dictionary

d = {[1]: "one"} # key must be immutable, for hashing
# output: TypeError: unhashable type: 'list'
```

Függvények

```
def procedure1(my_param): # if no return statement, returns None in the end
    print(my_param)

def function_double(my_param): # type of 'my_param' is not restricted
    return my_param*2

def function_swap(arg1, arg2): # Multiple returns: returns a tuple with a length of 2
    return arg2, arg1
```

Függvények - opcionális paraméterek

```
# paramters with default values must be in the back
def date_to_str(year, month=1, day=1):
    return str(year)+ "."+str(month).zfill(2)+"."+ str(day).zfill(2)

date_to_str(1962) # only year is given
date_to_str(1962, 5) # year and month is given
date_to_str(1962, day=10) # year and day is given

date_to_str(day=10) # argument without default value must be given
# results in: TypeError: date_to_str() missing 1 required positional
argument: 'year'
```

Függvények - függvényparaméterek és lambdák

```
def double(a):
    return a*2

def apply_twice(func, a):
    return func(func(a))

# applying double() twice
apply_twice(double, 3)
```

```
doubleL = lambda x: x*2


apply_twiceL = lambda f, x: f(f(x))


# Same as the other but the
function is defined in lambda form

apply_twiceL(doubleL, 3)
```

Vezérlési szerkezetek

- Kódblokkok indentációval vannak jelölve
- Szülő utasítás(pl: for loop) után whitespace-szel indentált blokk kell, hogy következzen.
- Egy blokk összes utasításának ugyanannyi whitespace karakterrel kell kezdenie a sort.
- Whitespace karakter lehet tabulátor és szóköz is, de a kettő keverése nem lehetséges
- Helytelen tagolás szintaxis hibához vezet.

```
parent statement:
    statement block 1
    ...
    parent statement:
        statement block 2
        ...
statement block 3
```

If-elif-else elagazás

```
if 4 <= feleves_jegy:
  print('Tanulmányi ösztöndíj $_$')
elif 1 < feleves_jegy:
  print('(╯°□°)╯︵ ┻━┻')
else:
  print('Annyira tetszett, hogy jövőre is felveszem (❤ ω ❤)')
```

Vezérlési szerkezetek

A következőkben a vezérlési szerkezeteket egy probléma megoldásával fogom bemutatni.

A probléma: Definiáld a **`numlist_to_numstrdict`** függvényt, ami számok listájából szám-sztring párok szótárát készíti el és adja vissza, az alábbi módon:

Pl.: `[3,3,3,2,-5,0.6] -> {3: '3', 2: '2', -5: '-5', 0.6: '0.6'}`

Vezérlési szerkezetek - Előltesztelős feltételes ciklus(while)

```
def numlist_to_numstrdict_with_while(num_list):
    i = 0
    numstrdict = dict()
    while i < len(num_list):
        curr_item = num_list[i]
        numstrdict[curr_item] = str(curr_item)
        i+=1

    return numstrdict
```

Vezérlési szerkezetek - Számlálós ciklus(for) és range

```
def numlist_to_numstrdict_with_for1(num_list):
    numstrdict = dict()

    ## the counter variable is handled
    for i in range(0, len(num_list)):
        curr_item = num_list[i]
        numstrdict[curr_item] = str(curr_item)

    return numstrdict
```

Range hasonlóan működik a slice-okhoz.
`range(from, to, step)` paraméterekkel hívható

Vezérlési szerkezetek - Számlálós ciklus(for), az elemeken iterálva

```
def numlist_to_numstrdict_with_for2(num_list):
    numstrdict = dict()

    ## iterating through items is handled
    for num in num_list:
        numstrdict[num] = str(num)

    return numstrdict
```

Az **`in`** kulcsszóval bármilyen iterálható típuson végigiterálhatunk, például listán, tuple-ön vagy dicten.

Vezérlési szerkezetek - Comprehensionök

```
def numlist_to_numstrdict_with_comprehension(num_list):
    return {num:str(num) for num in num_list}
```

Tömör, egysoros formája új szekvenciák(listák, dictek,...) létrehozásának.

Szintaktikus cukor (syntactic sugar).

Vezérlési szerkezetek - List comprehension

```
# get numbers between 1 and 20 that are dividable by 3
l1 = [item for item in range(1,21) if item % 3 == 0]

# write 'yeap' if it is dividable by 3 else print 'nope'
l2 = ["yeap" if item % 3 == 0 else "nope"  for item in
range(1,21)]
```

- Ha van else ág és minden ág hozzáad új elemet a szekvenciához =>az if/elif/else megelőzi a ciklus kulcsszavát
- Ha nincs else ág, vagy nem minden elem ad új elemet a listához => az if a ciklus kulcsszava után kell jöjjön

Szekvenciákon értelmezett beépített függvények

- Ezek a függvények iterálható típusokon értelmezettek
- Iterátort adnak vissza általában
- Iterátorokról bővebben: https://colab.research.google.com/drive/1V2aakDvAVd3j8Q2rFcWbn3ToL_FbNXxU
- Beépített függvények dokumentációja: https://docs.python.org/3/library/functions.html

A map() függvény

`map()`: egy szekvencia minden elemére végrehajt egy függvényt

```
def numlist_to_numstrdict_with_map(num_list):
    convert_to_string = lambda num: (num, str(num))

     # map returns an iterator object, needs to be converted
    return dict(map(convert_to_string, num_list))
```

A filter() függvény

`filter()`: egy szekvenciából szelektálja azokat az elemeket, melyekre a megadott függvény igazat ad

```
# get numbers that are dividable by 3 between 1 and 20

new_list = list(filter(lambda x: x%3==0, range(1,21)))
```

A zip() függvény

`zip()`: két szekvencia (lista, iterátor, stb.) összefűzése párok szekvenciájává. Az eredmény egy iterátor.

```
for x in zip([1, 2, 3], ['a', 'b'], range(5, 10)):
    print(x)
# output: (1, 'a', 5) (2, 'b', 6)
```

Az enumerate() függvény

`enumerate()`: egy szekvencián való végigiterálás közben a szekvencia elemei mellett az indexüket is sorban adjuk vissza.

```
# Iterating a sequence, printing the index and the value
for idx, item in enumerate(['bla', 'asd', None]):
  print('    at idx#', idx, ': ', item)
```

A min() és a max() függvény

```
my_list2 = [15, 25, 1.26, -56, 2., .3]
min(my_list2) # The minimum of a sequence
max(range(3,11)) # The maximum of a range iterator
```

Az any() és az all() függvény

```
my_bools = [2 < 3, 4 < 5 <= 1, True]


# Logical AND operator applied to a sequence of boolean expressions

all(my_bools))


# Logical OR operator applied to a sequence of boolean expressions

any(my_bools))
```

Osztályok

```
class NeptunLogin:
  # constructor
  def __init__(self, targyfelvetel):
    self.targyfelvetel = targyfelvetel

  def login(self, neptun, psw):
    if self.targyfelvetel:
      print("Várakozás szabad helyre...")
    else:
      raise NotImplementedError
```

Osztályok - példányosítás

```
#Példányosítás:
a = NeptunLogin(True)

a.login('alma', 'fa')
# prints Várakozás szabad helyre...

a.targyfelvetel = False
a.login('alma', 'fa')
# raises NotImplementedError
```

Osztályok - privát és protected mezők

```
# Privát és protected mezők:
class NeptunLogin:
    def __init__(self, targyfelvetel):
        self.targyfelvetel = targyfelvetel
        self._protected_field_by_convention = '※'
        self.__class_private_field = 42

a = NeptunLogin(True)
print(a._protected_field_by_convention)      # prints ※
print(a._NeptunLogin__class_private_field)   # prints 42

#output :AttributeError: 'NeptunLogin' object has no
attribute '__class_private_field'
print(a.__class_private_field)
```

Hibakezelés

- Szintaxis hibák: ha az interpreter nem tudja értelmezni a kódot
- Futási idejű excpetion-ök
- Ezeknek hasonlóan hívhatok meg kódból, mint C++-ban vagy Javaban
- Bővebben dokumentáció: https://docs.python.org/3.6/tutorial/errors.html
- Az `assert` kulcsszó használata a debuggolás egyik lehetséges eszköze. Ha az `assert` kulcsszó utáni kifejezés nem igazra értékelődik ki, `AssertionError` kivételt vált ki
- `unittest` modul : `assertTrue()`, és **assertEqual()** függvények

Exception példa

```
class MyException(Exception):
    pass
.
.
.
raise MyException()
```

Python cheat sheet és notebookok

- Python cheat sheet: https://perso.limsi.fr/pointal/_media/python:cours:mementopython3-english.pdf
- Python notebookok: https://colab.research.google.com/drive/1_dti2w4s5D8BVn5QZkPXuLX4-VCeI5xL és https://colab.research.google.com/drive/1TWoFKJAhxCF-gZKLimnF7upWuJaPOnZP

Könyvtárak importálása

Csomag betöltése a memóriába külön névtérként:

```
import numpy
# usage: numpy.add()
```

Alias használata. Hasznos, ha rövidíteni szeretnénk a modulnevet, vagy ha már foglalt:

```
import numpy as np
# usage: np.add()
```

Google Colab

- Interaktív online futtatási környezet a Google által
- Jupyter notebook-on alapul, melyet egy virtuális gép futtat
- Sok python csomag telepítve van
- Szöveg- és kódcellákból áll
- Egy kódblokk mindig kiírja az utolsó utasítás eredményét
- Változók fennmaradnak, míg fut a notebook

Telepítés

- Python: https://www.python.org/downloads/
- Csomagok telepítése: pip csomagkezelővel
- Anaconda disztribúció: Python tudományos csomagokkal és a conda csomagkezelővel kiegészítve
  https://www.anaconda.com/products/individual
- Jupyter Notebook(dokumentumok futtatható kóddal)
  https://jupyter.org/install

NumPy

Miért a NumPy?

- Gyors numerikus számításokra tervezték
- Python interpretáltsága miatt nehezen optimalizálható
- Ez főleg akkor feltűnő, ha nagyon sok utasítást kell végrehajtani egyszerre, például nagy mátrixok szorzásánál
- A NumPy ezen úgy javít, hogy az ilyen műveletekre hatékonyabb nyelveken(pl Fortran, C) írt, párhuzamosítható kódot hív meg
- Optimális cache használat
- Sok neurális hálós könyvtár(pl TensorFlow, PyTorch) épül NumPy-ra

Az ndarray típus

- Homogén típusú elemeket tartalmaz, 0-tól n-1-ig indexelve
- A homogenitás biztosítja, hogy a memóriában az elemek ugyanannyi helyet foglaljanak el, és ugyanúgy legyenek kezelve
- 2 meghatározó attribútuma: az alakja(`shape`) és a tartalmazott elemek típusa(`dtype`)
- Az elemek száma nem változhat, azonban a tömb alakja igen. Példa: egy 6 elemű tömb felvehet 2x3-as, 3x2-es, 6-os, 6x1-es, 6x1x1-es, 1x3x1x1x2x1-es, stb. alakot is

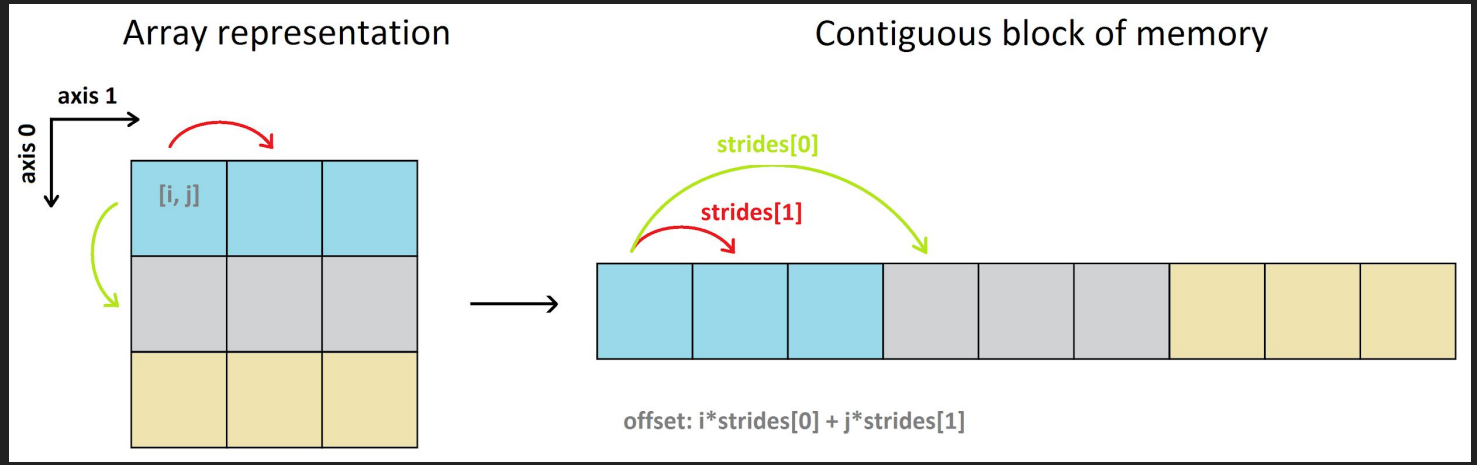
Tömbök elhelyezkedése a memóriában

- Függetlenül attól, hogy a tömbünk hány dimenziós, a memóriában folytonosan, egy dimenzióban helyezkedik el
- Az **`ndarray.strides`**(lépésköz) attribútum felel a konverzióért a tömb memóriabeli elhelyezkedése és a tömb többdimenziós viselkedése között, amit a kódban tapasztalunk
- A **`strides`** attribútum megadja, hogy az egyes tengelyek mentén hány bájtot kell lépnünk a következő eleméig
- Így ha a tömb alakját megváltoztatjuk, a Numpy csak a stride-okat változtatja meg
- Notebook a stride-okról: https://colab.research.google.com/drive/1_eqSAFC-qUY-YTT8TFK2K_XSpL9ZGrjM

Ndarray.strides - példa

```
a = np.arange(9, dtype=np.int32).reshape((3,3))

print(a.itemsize) # size of its items (in byte) - prints 4
print(a.shape) # prints (3,3)
print(a.strides) # prints (12, 4)
```



Ndarray és fő attribútumai

```
a = np.array([[2,3,4],[1,5,8]])   # create array

a.shape # The shape of the array
len(a) == a.shape[0] # always true

a.dtype # The data type of the array

a.ndim # The dimensionality of the array
a.ndim == len(a.shape) # always true
```

Tömbök létrehozása

```
# same as range() python function, but returns ndarray
np.arange(2, 10, 2) # np.arange(start, end, step)

np.zeros((3, 2), dtype=np.uint8) # ndarray of zeros
np.ones((3, 2), dtype=np.uint8) # ndarray of ones
np.full((3, 2), dtype=np.int64, fill_value=-1) # ndarray
filled with fill_value

# instead of explicitly passing shape,
# we can use _like functions and pass an array
# the result has the same shape as the passed array
np.full_like(arr, fill_value=256, dtype=np.int64)
np.zeros_like(arr, dtype=np.uint8)
```

Tömb alakjának változtatása

```
a = np.arange(12, dtype=np.int32).reshape((3,4))
print(a.shape) # prints (3,4)
print(a.itemsize) # prints 4
print(a.strides) # prints (16, 4)

# can only use -1 once, it deduces the possible value
b = a.reshape((-1, 6))
print(b.shape) # prints (2,6)
print(b.strides) # prints (24,4)

# reshape creates a view, so if a is modified, b will change
as well
a[0,0] = 42
```

Transzponálás

```
a = np.arange(6).reshape((2,3))
print(a)          # prints [[0 1 2] [3 4 5]]
print(a.shape)    # prints (2,3)
print(a.T)        # prints [[0 3] [1 4] [2 5]]
print(a.T.shape)  # prints (3,2)

# Tengelyek felcserélése: csak nézet készül
np.swapaxis()     # 2 tengely felcserélése
np.transpose()    # tetszőleges tengelysorrend
```

Típuskonverzió

```
a = np.arange(-1, 4) # a is [-1,0,1,2,3]
print(a.dtype) # prints int64

b = a.astype('float') # b is [-1.  0.  1.  2.  3]
b[2] = 99 # astype creates a copy, so a won't change

print(a.astype(np.uint8)) # prints [255   0   1   2   3]
print(a.astype(np.bool))  # prints [True False True, True,
True]
```

Elemek elérése, egyszerű indexelés

- A python alap slice-olást egészíti ki több dimenzióra
- A ‘`:`‘ egy teljes slice-ot jelöl, ami végigmegy az adott tengelyen
- Az egyszerű indexelésben lehet:
  - Slice objektumok(start:end:step szintaxissal)
  - Egész számok
  - **`np.newaxis`** vagy **`None`** (az eredmény kiterjesztése egy dimenzióval).
  - És egy darab ‘ellipsis’ (...)-tal jelölve. Ez annyi ‘`:`‘-tal egészíti ki a tuple, hogy a teljes tömböt indexelje
- Ha az index nem fedi le az indexelt tömb összes dimenzióját, kiegészítésre kerül a hiányzó számú ‘`:`‘-tal
- Nézetet ad vissza, lehet értékadás bal oldalán is

Egyszerű indexelés - példák

```
a = np.arange(6).reshape(3,2)
# content of a:
# [[0 1]
#  [2 3]
#  [4 5]]

#select last row
print(a[2]) # same as a[3, :], ":" is automatically
prepended
# output: [4 5]

# full slice(:) + integer index
print(a[:, 1]) # from each row select second element
# output: [1 3 5]
```

Egyszerű indexelés - még több példa

```
# two slice objects and newaxis
# newaxis expands the dimnesion of the result
print(a[:, np.newaxis, -2:].shape)
# output: (3, 1, 2)


b = np.arange(12).reshape(3,1,2,2)

# ellipsis, same as b[:,:,:,1]
print(b[..., 1]) # select second element along the last axis
```

Értékadás egyszerű indexeléssel

```
a = np.arange(6).reshape(2,3)

# a before setting:
# [[0 1 2]
#  [3 4 5]]
a[-2:,:2] = -1 # set the elements of the last 2 row in the
first 2 column
# a after setting:
# [[-1 -1  2]
#  [-1 -1  5]]
```

Univerzális függvények(`ufunc`)

- Elemenkénti műveleteket hajtanak végre `ndarray`-eken
- Az ufunc-ok között vannak egyszerű algebrai függvények, mint például összeadás vagy szorzás, trigonometrikus függvények, ...
- A broadcasting megengedi, hogy a műveleteket olyan tömbökön is végrehajtjuk, amelyeknek nem egyezik teljesen az alakjuk
- Ennek módjait a következő slide-okon mutatom be példák segítségével

Broadcasting - szabályok

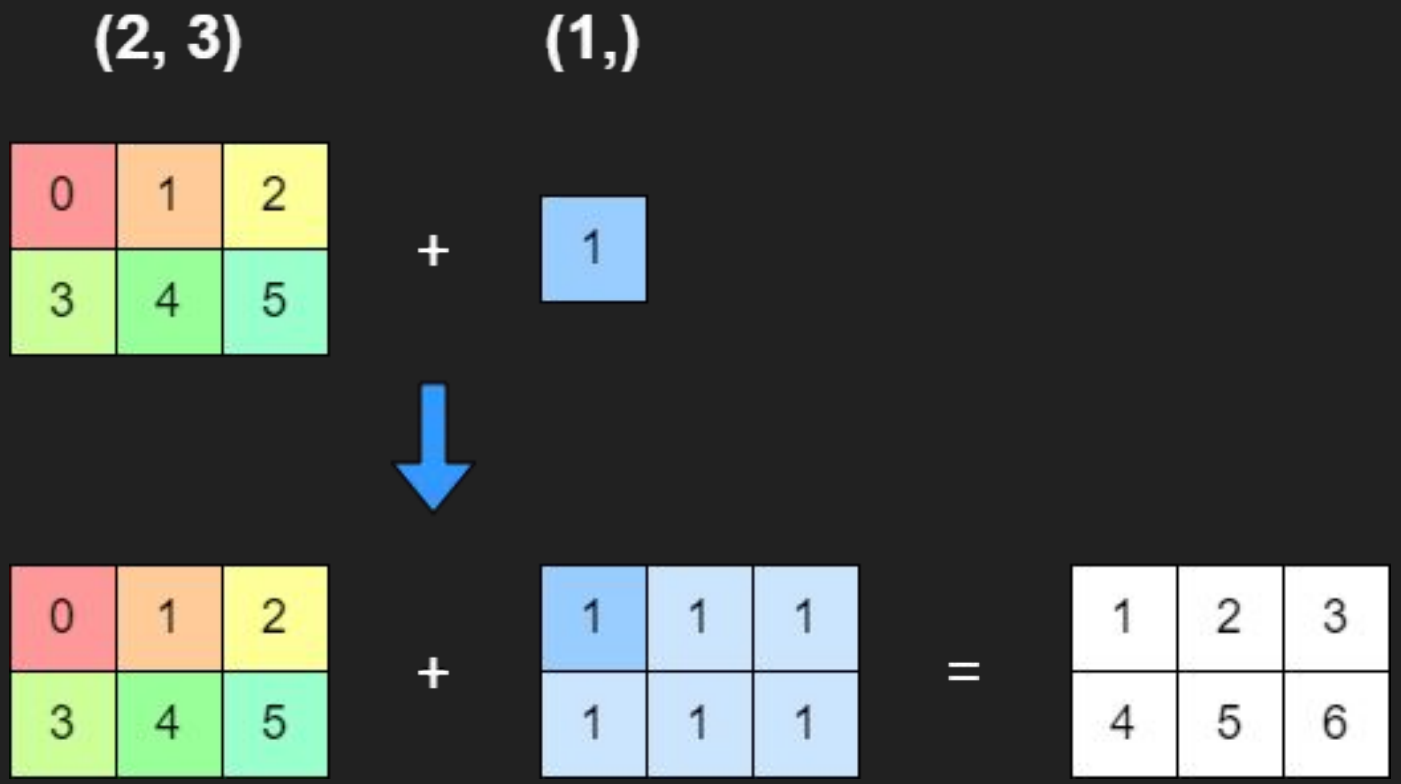
- Ha a két tömb dimenzióinak száma nem egyezik, a kevesebb dimenzióval rendelkező tömb új (1 hosszú) első tengelyeket kap. Éppen annyit, hogy dimenzióik száma egyezzen.
- A második szabály, ha valamelyik tengely mentén az egyik tömb hossza egy, az a tömb azon a tengely mentén annyiszor megismétlődik, hogy a hossza egyenlő legyen a másik tömbével

Broadcasting - egy számliterál

```
a = np.arange(6).reshape((2,3))

# a before the operation:
# [[0 1 2]
#  [3 4 5]]
a+=5
# a after the operation:
# [[ 5  6  7]
#  [ 8  9 10]]
```

Broadcasting - egy számliterál



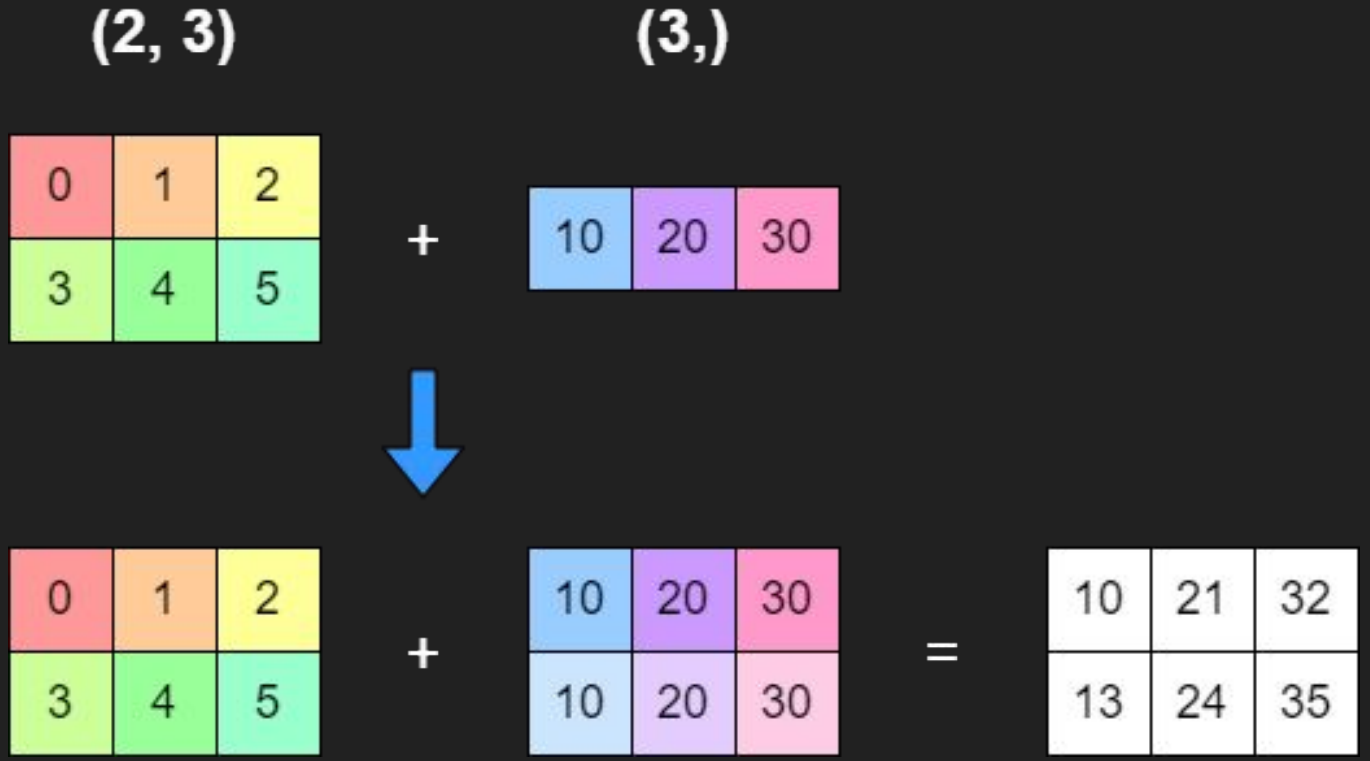
Broadcasting - a sorokon

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
# a is:
# [[0 1 2]
#  [3 4 5]]

vec = np.arange(10,40,10, dtype=np.int32)
# vec is:
# [10 20 30]

a = a+vec # same as: a = a+vec[np.newaxis, :]
# result
#[[10 21 32]
# [13 24 35]]
```

Broadcasting - a sorokon



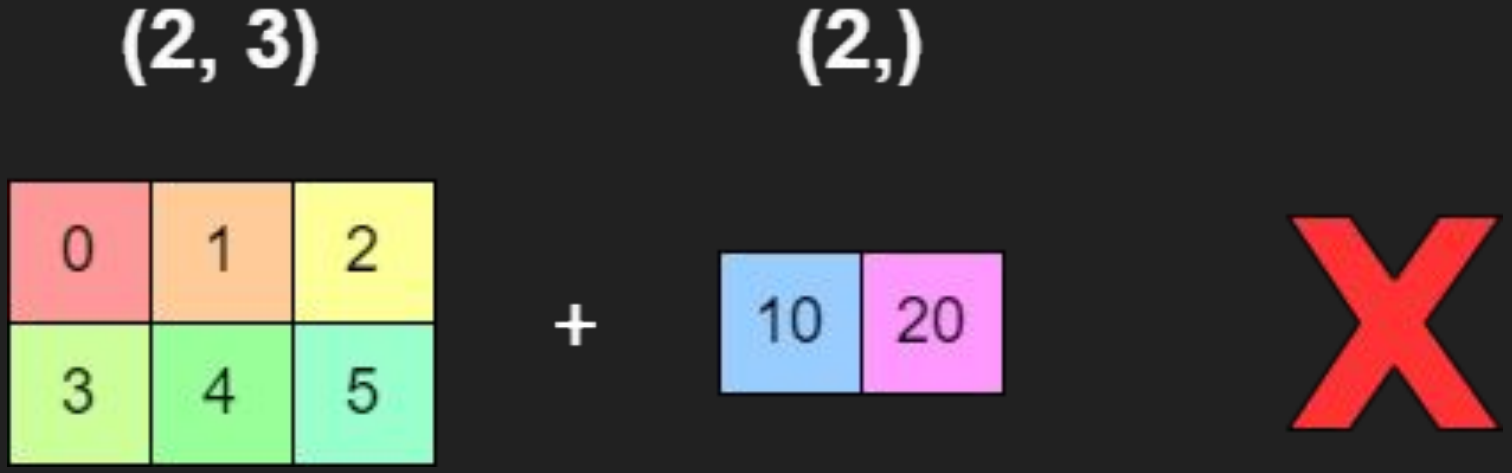
Broadcasting - az oszlopokon, hibásan

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
# a is:
# [[0 1 2]
#  [3 4 5]]

# vec is:
# [10 20]
vec = np.arange(10,30,10, dtype=np.int32)

a = a+vec
# result:
# ValueError: operands could not be broadcast together with
shapes (2,3) (2,)
```

Broadcasting - az oszlopokon, hibásan



Broadcasting - az oszlopokon helyesen

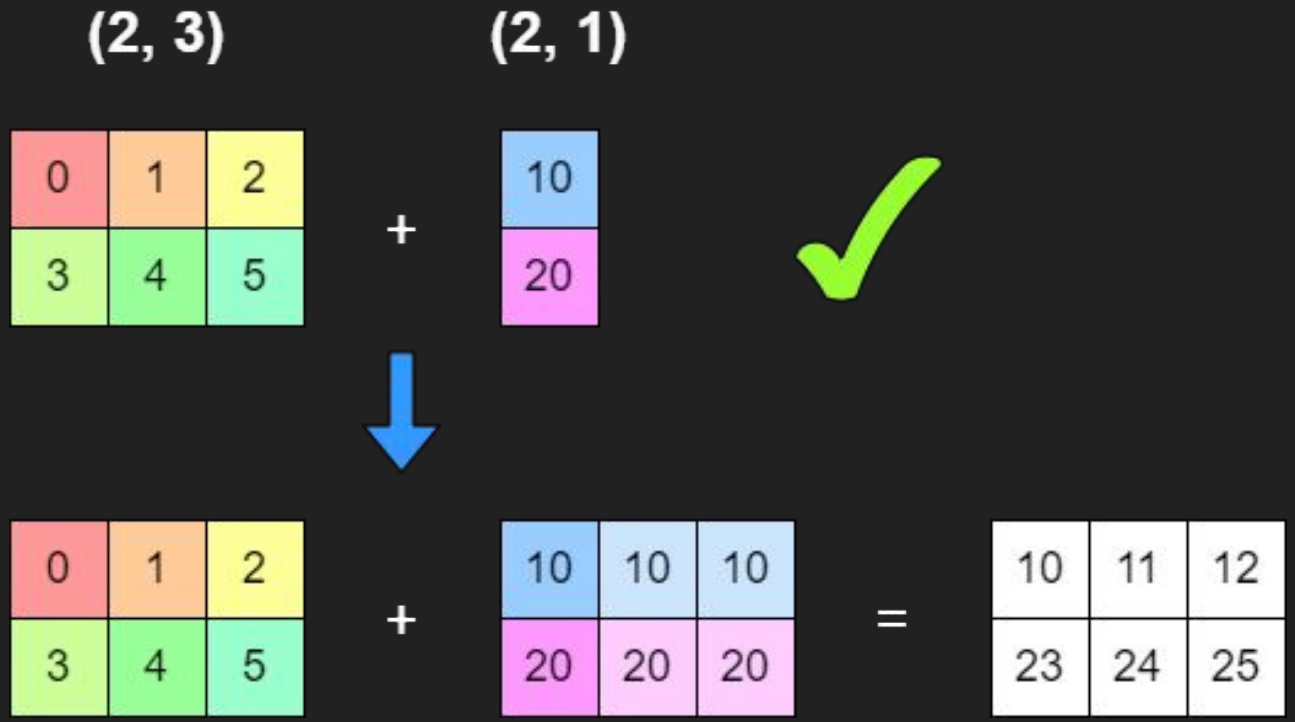
```
a = np.arange(6, dtype=np.int32).reshape((2,3))
# a is:
# [[0 1 2]
#  [3 4 5]]

# vec is:
# [10 20]
vec = np.arange(10,30,10, dtype=np.int32)

print(vec[:, np.newaxis])
# vec[:, np.newaxis] is:
# [[10]
#  [20]]

a = a+vec[:, np.newaxis]
# result is:
# [[10 11 12]
#  [23 24 25]]
```

Broadcasting - az oszlopokon, helyesen



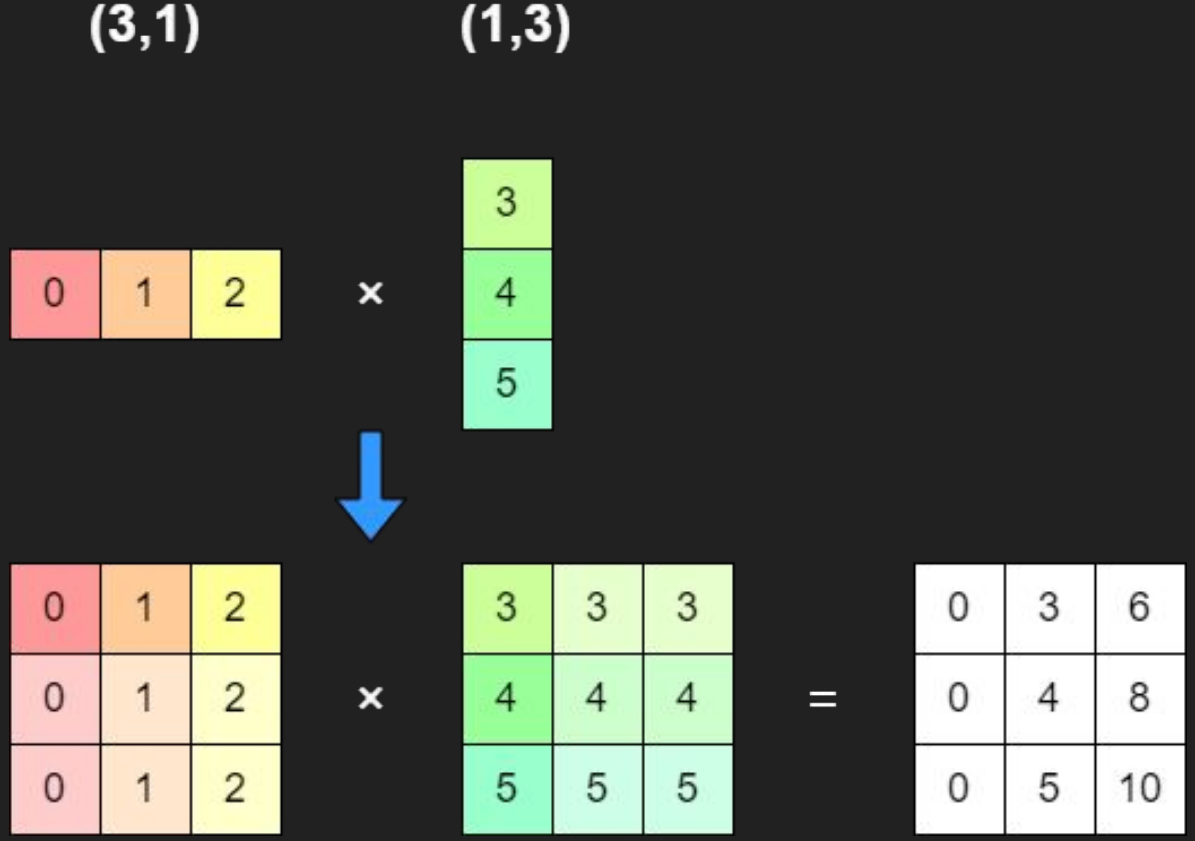
Broadcasting - szorzótábla

```
v1 = np.arange(3, dtype=np.int32)
# v1: [0 1 2]

v2 = np.arange(3, 6, dtype=np.int32)
print(v2[:, np.newaxis])
#: output:
# [[3]
#  [4]
#  [5]]

r = v1 * v2[:, np.newaxis]
# r is:
# [[ 0  3  6]
#  [ 0  4  8]
#  [ 0  5 10]]
```

Broadcasting - szorzótábla



Advanced indexing

- Advanced index esetén slice-ok helyett használhatunk bármilyen szekvenciát, ami integereket vagy logikai változókat tartalmaz
- Az advanced indexing azonban mindig másolatot ad vissza az egyszerű indexeléssel ellentétben
- Állhat advanced indexelt tömb értékadás bal oldalán, ilyenkor nem készül másolat, és az eredeti tömb íródik felül

Advanced indexing - példák

```
a = np.arange(6)
print(a) # [0 1 2 3 4 5]

indices = [2, 1, 1, 4, -1]
print(a[indices])  # [2 1 1 4 5]

mask = a % 2 == 0
print(mask)     # [ True False  True False  True False]
print(a[mask])       # [0 2 4]

a[indices] = 99
print(a) # [0 99 99  3 99 99]
a[mask] *=2
print(a) # [0  99 198   3 198  99]
```

Logikai indexelés - példa

```
# get all numbers from list that are greater than 10
# and not multiples of 3

arr = np.arange(20).reshape((4,5))

greater_than_10_mask = arr > 10
multiple_of_3_mask = arr % 3 == 0
mask = greater_than_10_mask & ~multiple_of_3_mask

arr[mask] # output: [11, 13, 14, 16, 17, 19]
```

Beépített függvények

Tengelyeken végzett műveletek

`axis` paraméter határozza meg, hogy melyik tengely mentén végezzük a műveletet:

- Ha **`axis=None`**(azaz az alapérték), akkor az egész tömbön hajtja végre a műveletet
- Ha **`axis=0`**, akkor a 0-s indexű tengelyen(kétdimenziós esetben a sorokon) hajtja végre a műveletet
- Ha **`axis=1`**, akkor az 1-es indexű tengelyen(kétdimenziós esetben az oszlopokon) hajtja végre a műveletet
- Magasabb dimenziókban hasonlóan, az **`axis`**-ban megadott tengely mentén vett 1 dimenziós szeleteken hajtódik végre a művelet
- Az **`axis`** paraméter lehet tuple is, ekkor a tupleben lévő összes tengelyen hajtódik végre a művelet.

Tengelyeken végzett műveletek - példa

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
# a is:
# [[0 1 2]
#  [3 4 5]]

# Summing along the whole array
np.sum(a)  # output: 15
# Summing array along axis #0
np.sum(a, axis=0) # output: [3 5 7]
# Summing array along axis#1
np.sum(a, axis=1) # output: [ 3, 12]
```

Tengelyeken végzett műveletek

```
# Példafeladat: egy 3x3-as tömbben mik a sorokban a
legkisebb elemek?

a = np.array([[-10, 2, 3], [80, 0, 70], [1,1,2]])

np.min(a, axis=1)
#output: array([-10,   0,   1])
```

Tengelyeken végzett műveletek

- Szélsőértékek:
  - `np.max()`, `np.min()`
  - `np.argmax()`, `np.argmin()`
- Aritmetikai műveletek:
  - `np.sum()`, `np.cumsum()`
  - `np.prod()`, `np.cumprod()`
  - `np.mean()`, `np.var()`, `np.std()`
- Logikai műveletek:
  - `np.all()`
  - `np.any()`

Shape függvények - flatten()

`ndarray.flatten()`: tömbpéldányon lehet meghívni. 1 dimenziósra ‘lapított’ másolatát adja vissza a tömbnek.

```
a = np.zeros((3,2,2))
b = a.flatten()
b.shape # output: (12,)
```

Shape függvények - Tömbök konkatenálása

- Ezek a műveletek tömbök szekvenciáját várják(lehet lista, tuple…)
- Új tömböt hoznak létre
- **`np.concatenate`**: a tömböket egy létező tengely mentén köti össze. Ezt az axis paraméterben kell átadni. A kokatenáláshoz használt tengelyt leszámítva egyalakúnak kell lennie a tömböknek
- **`np.stack`**: a tömböket egy új tengely mentén köti össze. A tömböknek egyalakúaknak kell lennie

Shape függvények - a concatenate függvény

```
a = np.zeros((2,3))
b = np.zeros((4,3))
np.concatenate((a,b), axis=0) # output: (6, 3)

np.concatenate((a,b), axis=-1) # defaults to last axis
# output : ValueError: all the input array dimensions for
the concatenation axis must match exactly, but along
dimension 0, the array at index 0 has size 2 and the array
at index 1 has size 4
```

Shape függvények - a stack függvény

```
a = np.zeros((4,3))
b = np.ones((4,3))

np.stack([a,b], axis=0)
# result:
# [[[0., 0., 0.],
# [0., 0., 0.],
# [0., 0., 0.],
# [0., 0., 0.]],

# [[1., 1., 1.],
# [1., 1., 1.],
# [1., 1., 1.],
# [1., 1., 1.]]]
```

Shape függvények - a stack függvény

```
a = np.zeros((4,3))
b = np.ones((4,3))

np.stack([a,b], axis=1)
# result:
# [[[0., 0., 0.],
# [1., 1., 1.]],

# [[0., 0., 0.],
# [1., 1., 1.]],

# [[0., 0., 0.],
# [1., 1., 1.]],

# [[0., 0., 0.],
# [1., 1., 1.]]]
```

Shape függvények - a stack függvény

```
a = np.zeros((4,3))
b = np.ones((4,3))

np.stack([a,b], axis=2)
# result:
# [[[0., 1.],
# [0., 1.],
# [0., 1.]],

# [[0., 1.],
# [0., 1.],
# [0., 1.]],

# [[0., 1.],
# [0., 1.],
# [0., 1.]],

# [[0., 1.],
# [0., 1.],
# [0., 1.]]]
```

Alapfüggvények - linspace, logspace

`np.linspace`: **`num`** paraméter számú, egyenlő távolságra lévő elemeket ad vissza az első 2 paraméter által meghatározott intevalumon.

```
np.linspace(2.0, 3.0, num=5)
# result:
# array([2.  , 2.25, 2.5 , 2.75, 3.  ])
```

`np.logspace`: **`num`** paraméter számú, logaritmikus skálán egyenlő távolságra lévő elemeket ad vissza **`base**start`** és **`base**stop`** intervalumon

```
np.logspace(2.0, 3.0, num=4, base=2.0)
# result:
# array([4.        , 5.0396842 , 6.34960421, 8.        ])
```

Alapfüggvények - unique

`numpy.unique:` visszaadja a tömb egyedi elemeit rendezve. Vannak opcionális paraméterei, melyekkel az elemek gyakoriságát, vagy a gyakori elemek indexét kérhetjük le. Az **`axis`** paramétert ha nem adjuk meg, a kilapított tömbön fut le. Ha szám, a számnak megfelelő tengely mentén fut le, és a résztömbök egyediségét vizsgálja.

```
arr = np.array([1,1,1,2,3,3])
unique_items, frequencies = np.unique(arr, return_counts = True)
# unique_items: array([1, 2, 3])
# frequencies: array([3, 1, 2])
```

Alapfüggvények - bincount

`numpy.bincount`: nemnegatatív egész számok gyakoriságát adja meg a tömbben. Az inputnak integerek tömbjének kell lennie. Az output tömb értékei az adott indexnek megfelelő szám gyakorisága. A kosarak számának (azaz az ouput tömb hosszának) az alapértéke a tömb maximuma.

```
arr = np.array([1,1,1,2,4,5,5])
counts = np.bincount(arr)
# counts: array([0, 3, 1, 0, 1, 2])
```

Alapfüggvények - diff

`numpy.diff(arr, n=1)`: szomszédos elemeinek különbségeit adja vissza egy tömbben. **`n`** paraméter megadja, hogy hányszor alkalmazzuk rekurzívan ezt a műveletet.

```
x = np.array([1, 2, 4, 7, 0])
diff = np.diff(x)
# diff: array([ 1,  2,  3, -7])
```

Apply_along_axis és vectorize problémái

- **`numpy.apply_along_axis(func1d, axis, arr): func1d`** függvényt **`arr axis`** mentén vett 1 dimenziós szeletein alkalmazza. A ciklus Pythhonban fut, nem C-ben, tehát kevésbé hatékony egy broadcastolt művelethez képest
- **`numpy.vectorize(pyfunc): pyfunc`** függvényt vektorizálja, így alkalmazhatóak rá a broadcasting szabályok. Kényelmi függvény, az implementáció lényegében egy Python for ciklus
- Használatukat kerüljétek a beadandókban!

Hasznos linkek

- Numpy notebookok:
  - https://drive.google.com/open?id=19Fc5mpRqUpDDEktg5xN_wrHd29Y3Usc1
  - https://drive.google.com/open?id=1zLn0dHVaKVTbMtaa1KqBD-WAd-JGOTut
- Numpy cheat sheet: https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf