Integratie Externe Functionaliteiten Les 2 : NumPy datatypes, iteratie, join & split

Datatypes

Via Python kennen we al een aantal veel voorkomende datatypes.

Strings: Reepjes tekst. Aangemaakt tussen quotes.
Integer: Gehele getallen.
Float: Kommagetallen.
Boolean: Geeft True of False weer.
…

NumPy breidt deze lijst voor ons aanzienlijk uit. Een aantal blijven hetzelfde. Naar deze datatypes wordt gerefereerd met een letter. Hierbij komt extra info over de lengte of het aantal bits het datatype inneemt. Hieronder volgen enkele veel voorkomende datatypes.

i: integer. Gevolgd door het aantal bits.
b: boolean
u: unsigned integer, absolute waarde
f: float, kommagetal
M: datum en tijd.
S: string, stukje tekst
U: unicode string. Tekst in unicode geschreven. (Bijna alle tekst.)

dtype

Datatype controleren

NumPy heeft een ingebouwde functie dtype. Die kunnen we gebruiken om het datatype op de vragen.

Maak een 2 nieuwe array's.
1 met 4 cijfers en 1 met 3 soorten bloemen.
Print de datatypes af met volgende methode:
    naamArray.dtype
Waarvoor zou de extra info achter het datatype staan?

Datatype definiëren

We kunnen bij het aanmaken van de array ook het datatype van de elementen bepalen. Dit zagen we al kort bij de les over het aanmaken van array's.

getallenAlsString = np.array([1,2,3,4], dtype='U')
Print het datatype van deze array af.

Voor de types i, u, f, S en U kunnen we ook de grootte bepalen. We geven mee hoeveel bytes of tekens deze mogen innemen.

Beperk de unicode strings van de array met bloemen tot 2 tekens: dtype = 'U2'
Print deze af. Wat merk je op?

Verander een cijfer uit de array met nummers door 500.
Beperk het datatype tot een integer met 1 byte.
Print deze af.

Let op! Een array met nummers kan als een string worden weergegeven, maar niet andersom.

Datatypes omvormen

We kunnen ook een bestaande array omvormen in een nieuwe array met een ander datatype. Hiervoor gebruiken we niet **dtype**, maar **astype**.

```
floats = np.array([1.1, 2.1, 3.1])
ints = floats.astype('i')
print(ints)
print(ints.dtype)
```

Hier kunnen we een array van nummers naar bools omzetten. Alles wordt dan weergegeven door True, buiten 0. Die zal worden weergegeven door False.

```
getallen2 = np.array([1,0,10,4])
bools = getallen2.astype('i')
print(bools)
print(bools.dtype)
```

Iteratie

Loopen met de for-loop
1D – array

Om alle elementen uit een 1D array apart op te vragen kunnen we een gewone for-loop gebruiken. Dit zagen we al in de cursus start to program.

> Maak automatisch een array aan met 16 getallen.
> Print deze getallen afzonderlijk af met een for-loop.

2D – array

Bij een 2D-array wordt het iets complexer. We kunnen niet met een enkelvoudige lus werken.

> Hervorm de array tot een 2D array.
> Loop net zoals bij de 1D array door de array.
> Wat krijgen we als resultaat?

We moeten dus gebruik maken van een geneste array. We spreken 1 voor 1 alle array's in de overkoepelende array aan en lopen daar door.

> Pas een geneste loop toe op de 2D array om alle elementen individueel af te printen.

3D en andere multidimensionale array's

We drijven de “complexiteit” nog iets op met 3D en multidimensionale arrays. Voor elke laag maken we een extra for-loop aan.

> Verdeel de originele array verder op in een 3D array.
> Voeg een extra geneste loop aan om alle elementen apart af te printen.

nditer

We merken al snel dat dit niet de meest flexibele manier is. We moeten weten hoeveel lagen. Als we hervormen moet ook die code aangepast worden. Enzoverder. En in de meeste gevallen willen we gewoon een loop om apart de elementen op de vragen.

Gelukkig heeft NumPy dit voorzien en een methode ingebouwd: **nditer.** Hierbij maken we slechts 1 lus aan en nditer zal de rest doen.

```
for i in np.nditer(loops2D):
    print(i)
```

Nu hebben we alle elementen met 1 loop bereikt.

We kunnen ook alle items opvragen en in 1 keer van datatype veranderen. Handig als we getallen willen omzetten naar strings om ergens in een string toe te voegen bijvoorbeeld.
NumPy doet dit echter niet in de originele array dus zal dit even moeten opslaan in een “buffer”.
In plaats van dtype gebruiken we **op_dtypes**
De loop ziet er dan als volgt uit:

```
for i in np.nditer(loops3D, flags=['buffered'], op_dtypes=['S']):
    print(i)
```

iteratie met steps

We passen eenzelfde manier toe om te loopen met tussenstappen als bij het opvragen van de volledige array. We voegen dan tussen vierkante haakjes en met dubbele punten de grootte van de stappen toe. Let wel op, we moeten hierbij wel het aantal lagen kennen.

```
for i in np.nditer(loops2D[:,::2]):
    print(i)
```

ndenumerate

Soms hebben we de overeenstemmende index van de elementen nodig. Ook dit kunnen we via een loop opvragen.

```
for indx, i in np.ndenumerate(loops3D):
    print(indx, i)
```

We krijgen nu een de index in de vorm van een tuple. Net zoals wanneer we de shape zouden opvragen. Met de index van buiten naar binnen toe.

Join & Split

Join

We kunnen aparte arrays samenvoegen tot 1 multidimensionale array. Dit kan met 1D array's en met multidimensionale array's. Er is ook een verschillende aanpak mogelijk.

Concatenate

Met concatenate voegen we simpelweg de arrays samen in 1 array.

> Maak automatisch 2 1D array's. Beide met met 3 getallen.
> In een nieuwe variabele of print voer je volgende methode uit:
> np.concatenate((arr1,arr2))

Met multidimensionale array's hebben we meer opties. We kunnen dan een as, **axis**, toevoegen. Zo bepalen we welke delen samengevoegd worden in de nieuwe array.

> Maak 2 2D array's aan.
> Voeg deze samen met de concatenate methode.
> Voeg nu de extra parameter axis toe.
> Deze staat automatisch op 0. Wat gebeurt er als we deze op 1 of 2 zetten?

Stacking

Stacking of stapelen is hetzelfde als concatenate. Alleen gebeurt dit in de “hoogte”. We moeten hier ook een as meegeven. Anders worden alleen de array's op elkaar gestapeld.

> Neem reeds aangemaakte 1D array's en stapel deze.
> Gebruik dus **stack ipv concatenate**
>
> Voeg nu een axis met waarde 1 toe.
> Merk je het verschil?

hstack, vstack & dstack

hstack: stapelt over de rijen.

vstack: stapelt over de kolommen.

dstack: stapelt in de diepte.

> Probeer deze uit op onze 1D array's om de verschillen te zien.
> Probeer deze ook eens uit op de 2D array's.

Split

Om een op te delen gebruiken we de array_split() of split() methode.
We geven als attributen mee welke array we willen splitten en hoeveel splits we willen.
Let op! Dit is vergelijkbaar met reshape, maar niet hetzelfde.
We maken immers lijsten aan en geen NumPy array's.

Maak een automatische array van 9 cijfers aan.
Splits deze op in lijsten van 2 elementen met de array_split() methode.
Probeer hetzelfde met de split() methode.

Vraag het type of van de nieuwe lijsten.
Doe dit met de oude type() methode en niet met de dtype manier.

Probeer nu de array te verdelen in 4 of 5 stukken.
Wat is het resultaat?
Zijn er verschillen met de reshape() of de split() methodes?

Hsplit, vsplit en dsplit zijn de tegenhangers van hstack, vstack en dstack.
Probeer ze zometeen zelf uit.

We kunnen in plaats van gelijke delen ook de indexnummers geven waar gesplit moet worden. Dit plaatsen we dan tussen vierkante haken.

Splits de vorige array op de 2de en 6de index.

Zelf uitproberen:

Maak een automatische array met 12 getallen.
Vervorm deze tot een 2D array met 3 getallen per array.
Splits deze array op in 3 gelijke delen.

Maak een automatische array met 18 getallen.
Vervorm deze tot een 2D array met 3 getallen per array.
Splits deze array op in 3 gelijke delen, maar voeg ook nog een as van 1 toe.

Probeer ook de hsplit, dsplit en vsplit uit op deze array. Vergeet wel niet mee te geven in hoeveel delen je wil splitten.

Maak een automatische array met 9 cijfers.
Pas hier een split op toe, maar ipv het aantal delen n