

Mutable en immutable

In Python kan je een variabele conceptueel het beste beschouwen als een verwijzing naar een *object*. Als we schrijven `x=5` vragen we aan Python om variabele `x` naar het object `5` te laten verwijzen. Verder onderscheidt Python twee soorten objecten; *mutable* (wijzigbare) objecten en *immutable* (niet-wijzigbare) objecten. Een voorbeeld van een mutable object is een lijst; als we met `L.append(element)` een element toevoegen aan een lijst, dan is dat nog steeds dezelfde lijst, maar is die lijst wel veranderd.

Waarom is dit belangrijk? Omdat dit erg verwarrend kan zijn als we ons hier niet bewust van zijn. Beschouw bijvoorbeeld volgende regels code (Zie ook [Python tutor](https://pythontutor.com/visualize.html#code=L%3D%5B%5D%20%20%23%20L%20verwijst%20naar%20een%20frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false) (<https://pythontutor.com/visualize.html#code=L%3D%5B%5D%20%20%23%20L%20verwijst%20naar%20een%20frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false>)):

```
In [9]: L=[] # L verwijst naar een nieuwe, lege lijst
        K=L  # K verwijst naar *dezelfde* lijst

        L.append(1)
        print(K)

[1]
```

Wat gebeurt hier? Telkens als we een uitdrukking `variabele=waarde` gebruiken, vergeten we waarnaar variabele verwijst en laten we vanaf dan variabele wijzen naar object waarde. Wanneer we `L=[]` schrijven, maken we een nieuwe lijst aan, en laten we `L` hiernaar verwijzen. Als we in de volgende regel dan `K=L` schrijven, laten we `K` verwijzen naar de lijst `L`. Variabelen `K` en `L` verwijzen dus beiden naar *dezelfde* lijst. Dus, als we vervolgens lijst `L` "muteren" door er een element aan toe te voegen, dan verandert dus ook `K`. Of beter gezegd: aangezien `K` naar dezelfde lijst verwijst als `L`, zal het wijzigen van de lijst waar `L` naar verwijst tot gevolg hebben dat ook de lijst waar `K` naar verwijst, gewijzigd is.

Matrix maken

Bekijk even volgende foutief voorbeeld om een eenheidsmatrix aan te maken, waarbij de interpretatie van mutable en variabelen die naar hetzelfde lijst-object verwijzen, grondig misloopt (Zie ook [Python tutor](https://pythontutor.com/visualize.html#code=n%3D5%0Anulrij%3D%5B0%5D*n%0AM%3D%5B%5D%0Afor%20i' frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false) (https://pythontutor.com/visualize.html#code=n%3D5%0Anulrij%3D%5B0%5D*n%0AM%3D%5B%5D%0Afor%20i' frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)).

```
In [10]: n=5
nulrij=[0]*n
M=[]
for i in range(n):
    rij=nulrij
    rij[i]=1
    M.append(rij)

print(M)
```

```
[[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1,
1, 1, 1]]
```

Het probleem hier is dat `rij=nulrij` geen nieuwe rij aanmaakt, maar `rij` laat verwijzen naar hetzelfde lijst-object als `nulrij`. Willen we dit vermijden kunnen we gebruik maken van `copy` (uit module `copy`), of `nulrij[:]` gebruiken; de slicing operator genereert een nieuwe lijst; hier in dit geval een lijst die toevallig exact even groot is als de oorspronkelijke lijst (omdat begin- en eindindex ontbreken in de notatie `[:]` worden standaard het begin en het einde van de lijst genomen) (Zie ook [Python tutor](https://pythontutor.com/visualize.html#code=n%3D5%0Anulrij%3D%5B0%5D*n%0AM%3D%5B%5D%0Afor%20i%20in%20range(n)%3A%20rij%3Dn%0Arij[i]%3D1%0AM.append(rij)%0Aprint(M)&rawInputLstJSON=%5B%5D&textReferences=false) ([https://pythontutor.com/visualize.html#code=n%3D5%0Anulrij%3D%5B0%5D*n%0AM%3D%5B%5D%0Afor%20i%20in%20range\(n\)%3A%20rij%3Dn%0Arij\[i\]%3D1%0AM.append\(rij\)%0Aprint\(M\)&rawInputLstJSON=%5B%5D&textReferences=false](https://pythontutor.com/visualize.html#code=n%3D5%0Anulrij%3D%5B0%5D*n%0AM%3D%5B%5D%0Afor%20i%20in%20range(n)%3A%20rij%3Dn%0Arij[i]%3D1%0AM.append(rij)%0Aprint(M)&rawInputLstJSON=%5B%5D&textReferences=false))).

```
In [11]: n=5
nulrij=[0]*n
M=[]
for i in range(n):
    rij=nulrij[:]
    rij[i]=1
    M.append(rij)

print(M)
```

```
[[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0], [0, 0,
0, 0, 1]]
```

of ook:

```
In [12]: from copy import copy
```

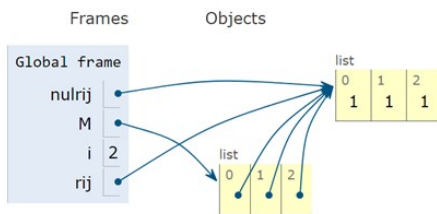
```
n=5
nulrij=[0]*n
M=[]
for i in range(n):
    rij=copy(nulrij)
    rij[i]=1
    M.append(rij)

print(M)
```

```
[[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0], [0, 0,
0, 0, 1]]
```

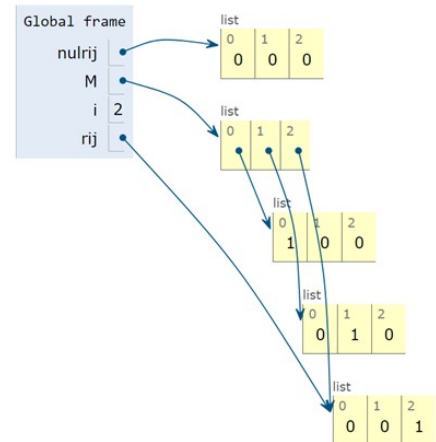
Volgende visualisatie vat het aanmaken van een matrix samen:

```
nulrij=[0]*3
M=[]
for i in range(3):
    rij=nulrij
    rij[i]=1
    M.append(rij)
```



```
#optie 1
nulrij=[0]*3
M=[]
for i in range(3):
    rij=nulrij[:]
    rij[i]=1
    M.append(rij)
```

```
#optie 2
M=[]
for i in range(3):
    rij=[0]*3
    rij[i]=1
    M.append(rij)
```



Kopij maken van een matrix

Een gelijkaardig probleem krijgen we als we een matrix willen kopiëren (Zie ook [Python Tutor](https://pythontutor.com/visualize.html#code=from%20copy%20import%20copy%0A%0Anulrij%3D%5B%5B%5D%5D&textReferences=false) (<https://pythontutor.com/visualize.html#code=from%20copy%20import%20copy%0A%0Anulrij%3D%5B%5B%5D%5D&textReferences=false>)).

$$[[2, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0], [0, 0, 0, 0, 1]]$$

◀ ▶

$$[[2, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0], [0, 0, 0, 0, 1]]$$

◀ ▶

```
In [15]: n=5
nulrij=[0]*n
M=[]
for i in range(n):
    rij=copy(nulrij)
    rij[i]=1
    M.append(rij)

N=[]
for rij in M:
    N.append(copy(rij)) # voeg een kopij van de rij toe
```

Merk op dat we ook een diepe kopij kunnen maken met de functie `deepcopy` van de module `copy` :

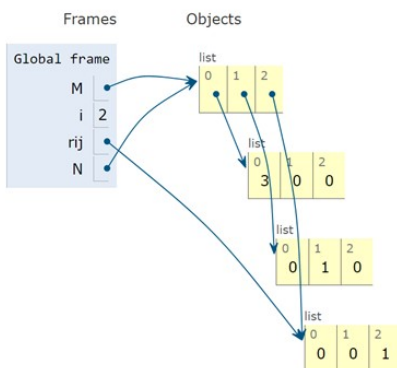
```
In [16]: from copy import deepcopy

n=5
nulrij=[0]*n
M=[]
for i in range(n):
    rij=copy(nulrij)
    rij[i]=1
    M.append(rij)

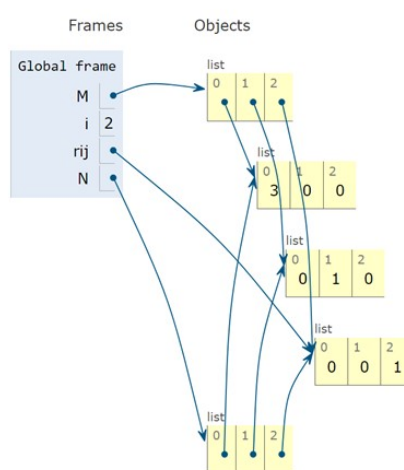
N=deepcopy(M)
```

Volgende visualisatie vat het maken van een kopij van een matrix samen:

```
M=[]
for i in range(3):
    rij=[0]*3
    rij[i]=1
    M.append(rij)
N=M
N[0][0]=3
```



```
M=[]
for i in range(3):
    rij=[0]*3
    rij[i]=1
    M.append(rij)
N=M[:]
N[0][0]=3
```



```
M=[]
for i in range(3):
    rij=[0]*3
    rij[i]=1
    M.append(rij)
N=[]
for i in range(3):
    N.append(M[i][:])
N[0][0]=3
```

